34th International Symposium on Distributed Computing

DISC 2020, October 12–16, 2020, Virtual Conference

Edited by

Hagit Attiya

Editors

Hagit Attiya
Technion, Haifa, Israel
hagit@cs.technion.ac.il

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

ISBN 978-3-95977-168-9

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at https://www.dagstuhl.de/dagpub/978-3-95977-168-9.

Publication date
October, 2020

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

License
This work is licensed under a Creative Commons Attribution 3.0 Unported license (CC-BY 3.0): https://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.2020.0

ISBN 978-3-95977-168-9 **ISSN 1868-8969** **https://www.dagstuhl.de/lipics**

LIPIcs – Leibniz International Proceedings in Informatics

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

Editorial Board

- Luca Aceto (*Chair*, Gran Sasso Science Institute and Reykjavik University)
- Christel Baier (TU Dresden)
- Mikolaj Bojanczyk (University of Warsaw)
- Roberto Di Cosmo (INRIA and University Paris Diderot)
- Javier Esparza (TU München)
- Meena Mahajan (Institute of Mathematical Sciences)
- Dieter van Melkebeek (University of Wisconsin-Madison)
- Anca Muscholl (University Bordeaux)
- Luke Ong (University of Oxford)
- Catuscia Palamidessi (INRIA)
- Thomas Schwentick (TU Dortmund)
- Raimund Seidel (Saarland University and Schloss Dagstuhl – Leibniz-Zentrum für Informatik)

ISSN 1868-8969

https://www.dagstuhl.de/lipics

Contents

Preface
Hagit Attiya 0:ix–0:x

Symposium Organization
.... 0:xi–0:xiv

2020 Edsger W. Dijkstra Prize in Distributed Computing
.... 0:xv

2020 Principles of Distributed Computing Doctoral Dissertation Award
.... 0:xvii

Regular Papers

Improved Bounds for Distributed Load Balancing
Sepehr Assadi, Aaron Bernstein, and Zachary Langley 1:1–1:15

Intermediate Value Linearizability: A Quantitative Correctness Criterion
Arik Rinberg and Idit Keidar 2:1–2:17

The Splay-List: A Distribution-Adaptive Concurrent Skip-List
Vitaly Aksenov, Dan Alistarh, Alexandra Drozdova, and Amirkeivan Mohtashami . 3:1–3:18

Efficient Multi-Word Compare and Swap
Rachid Guerraoui, Alex Kogan, Virendra J. Marathe, and Igor Zablotchi 4:1–4:19

LL/SC and Atomic Copy: Constant Time, Space Efficient Implementations Using Only Pointer-Width CAS
Guy E. Blelloch and Yuanhao Wei 5:1–5:17

Message Complexity of Population Protocols
Talley Amir, James Aspnes, David Doty, Mahsa Eftekhari, and Eric Severson 6:1–6:18

Distributed Computation with Continual Population Growth
Da-Jung Cho, Matthias Függer, Corbin Hopper, Manish Kushwaha, Thomas Nowak, and Quentin Soubeyran 7:1–7:17

Who Started This Rumor? Quantifying the Natural Differential Privacy of Gossip Protocols
Aurélien Bellet, Rachid Guerraoui, and Hadrien Hendrikx 8:1–8:18

Spread of Information and Diseases via Random Walks in Sparse Graphs
George Giakkoupis, Hayk Saribekyan, and Thomas Sauerwald 9:1–9:17

Spiking Neural Networks Through the Lens of Streaming Algorithms
Yael Hitron, Cameron Musco, and Merav Parter 10:1–10:18

Communication Efficient Self-Stabilizing Leader Election
Xavier Défago, Yuval Emek, Shay Kutten, Toshimitsu Masuzawa, and Yasumasa Tamura 11:1–11:19

34th International Symposium on Distributed Computing (DISC 2020).
Editor: Hagit Attiya
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Gathering on a Circle with Limited Visibility by Anonymous Oblivious Robots
Giuseppe A. Di Luna, Ryuhei Uehara, Giovanni Viglietta, and Yukiko Yamauchi . 12:1–12:17

Tight Bounds for Deterministic High-Dimensional Grid Exploration
Sebastian Brandt, Julian Portmann, and Jara Uitto 13:1–13:16

Distributed Dispatching in the Parallel Server Model
Guy Goren, Shay Vargaftik, and Yoram Moses 14:1–14:18

Distributed Dense Subgraph Detection and Low Outdegree Orientation
Hsin-Hao Su and Hoa T. Vu 15:1–15:18

Local Conflict Coloring Revisited: Linial for Lists
Yannic Maus and Tigran Tonoyan 16:1–16:18

Classification of Distributed Binary Labeling Problems
Alkida Balliu, Sebastian Brandt, Yuval Efron, Juho Hirvonen, Yannic Maus, Dennis Olivetti, and Jukka Suomela 17:1–17:17

The Complexity Landscape of Distributed Locally Checkable Problems on Trees
Yi-Jun Chang 18:1–18:17

Improved Hardness of Approximation of Diameter in the CONGEST Model
Ofer Grossman, Seri Khoury, and Ami Paz 19:1–19:16

Twenty-Two New Approximate Proof Labeling Schemes
Yuval Emek and Yuval Gil 20:1–20:14

Distributed Constructions of Dual-Failure Fault-Tolerant Distance Preservers
Merav Parter 21:1–21:17

Singularly Optimal Randomized Leader Election
Shay Kutten, William K. Moses Jr., Gopal Pandurangan, and David Peleg 22:1–22:18

Making Byzantine Consensus Live
Manuel Bravo, Gregory Chockler, and Alexey Gotsman 23:1–23:17

Leaderless State-Machine Replication: Specification, Properties, Limits
Tuanir França Rezende and Pierre Sutra 24:1–24:17

Not a COINcidence: Sub-Quadratic Asynchronous Byzantine Agreement WHP
Shir Cohen, Idit Keidar, and Alexander Spiegelman 25:1–25:17

Expected Linear Round Synchronization: The Missing Link for Linear Byzantine SMR
Oded Naor and Idit Keidar 26:1–26:17

Asynchronous Reconfiguration with Byzantine Failures
Petr Kuznetsov and Andrei Tonkikh 27:1–27:17

Improved Extension Protocols for Byzantine Broadcast and Agreement
Kartik Nayak, Ling Ren, Elaine Shi, Nitin H. Vaidya, and Zhuolun Xiang 28:1–28:17

From Partial to Global Asynchronous Reliable Broadcast
Diana Ghinea, Martin Hirt, and Chen-Da Liu-Zhang 29:1–29:16

Fast Agreement in Networks with Byzantine Nodes
Bogdan S. Chlebus, Dariusz R. Kowalski, and Jan Olkowski 30:1–30:18

Scalable and Secure Computation Among Strangers: Message-Competitive Byzantine Protocols
John Augustine, Valerie King, Anisur Rahaman Molla, Gopal Pandurangan, and Jared Saia 31:1–31:19

Byzantine Lattice Agreement in Synchronous Message Passing Systems
Xiong Zheng and Vijay Garg 32:1–32:16

Fast Distributed Algorithms for Girth, Cycles and Small Subgraphs
Keren Censor-Hillel, Orr Fischer, Tzlil Gonen, François Le Gall, Dean Leitersdorf, and Rotem Oshman 33:1–33:17

Improved MPC Algorithms for MIS, Matching, and Coloring on Trees and Beyond
Mohsen Ghaffari, Christoph Grunau, and Ce Jin 34:1–34:18

Improved Distributed Approximations for Maximum Independent Set
Ken-ichi Kawarabayashi, Seri Khoury, Aaron Schild, and Gregory Schwartzman .. 35:1–35:16

Models of Smoothing in Dynamic Networks
Uri Meir, Ami Paz, and Gregory Schwartzman 36:1–36:16

Distributed Maximum Matching Verification in CONGEST
Mohamad Ahmadi and Fabian Kuhn 37:1–37:18

Distributed Planar Reachability in Nearly Optimal Time
Merav Parter 38:1–38:17

Coloring Fast Without Learning Your Neighbors' Colors
Magnús M. Halldórsson, Fabian Kuhn, Yannic Maus, and Alexandre Nolin 39:1–39:17

Brief Announcements

Brief Announcement: Efficient Load-Balancing Through Distributed Token Dropping
Sebastian Brandt, Barbara Keller, Joel Rybicki, Jukka Suomela, and Jara Uitto .. 40:1–40:3

Brief Announcement: Distributed Graph Problems Through an Automata-Theoretic Lens
Yi-Jun Chang, Jan Studený, and Jukka Suomela 41:1–41:3

Brief Announcement: Phase Transitions of the k-Majority Dynamics in a Biased Communication Model
Emilio Cruciani, Hlafo Alfie Mimun, Matteo Quattropani, and Sara Rizzo 42:1–42:3

Brief Announcement: Distributed Quantum Proofs for Replicated Data
Pierre Fraigniaud, François Le Gall, Harumichi Nishimura, and Ami Paz 43:1–43:3

Brief Announcement: Optimally-Resilient Unconditionally-Secure Asynchronous Multi-Party Computation Revisited
Ashish Choudhury 44:1–44:3

Brief Announcement: Polygraph: Accountable Byzantine Agreement
Pierre Civit, Seth Gilbert, and Vincent Gramoli 45:1–45:3

Brief Announcement: What Can(Not) Be Perfectly Rerouted Locally
Klaus-Tycho Foerster, Juho Hirvonen, Yvonne-Anne Pignolet, Stefan Schmid, and Gilles Tredan 46:1–46:3

Brief Announcement: Byzantine Agreement, Broadcast and State Machine Replication with Optimal Good-Case Latency
Ittai Abraham, Kartik Nayak, Ling Ren, and Zhuolun Xiang 47:1–47:3

Brief Announcement: Multi-Threshold Asynchronous Reliable Broadcast and Consensus
Martin Hirt, Ard Kastrati, and Chen-Da Liu-Zhang 48:1–48:3

Brief Announcement: Game Theoretical Framework for Analyzing Blockchains Robustness
Paolo Zappalà, Marianna Belotti, Maria Potop-Butucaru, and Stefano Secci 49:1–49:3

Brief Announcement: Jiffy: A Fast, Memory Efficient, Wait-Free Multi-Producers Single-Consumer Queue
Dolev Adas and Roy Friedman 50:1–50:3

Brief Announcement: Concurrent Fixed-Size Allocation and Free in Constant Time
Guy E. Blelloch and Yuanhao Wei 51:1–51:3

Brief Announcement: Building Fast Recoverable Persistent Data Structures with Montage
Haosen Wen, Wentao Cai, Mingzhe Du, Benjamin Valpey, and Michael L. Scott . 52:1–52:3

Brief Announcement: Reaching Approximate Consensus When Everyone May Crash
Lewis Tseng, Qinzi Zhang, and Yifan Zhang 53:1–53:3

Brief Announcement: On Decidability of 2-Process Affine Models
Petr Kuznetsov and Thibault Rieutord 54:1–54:3

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 2020, the 34th International Symposium on Distributed Computing, held as a virtual event online on October 12–16, 2020. It also includes the citations for two awards jointly sponsored by DISC and the ACM Symposium on Principles of Distributed Computing (PODC):

- The *2020 Edsger W. Dijkstra Prize in Distributed Computing*, presented at PODC 2020, to Dana Angluin, James Aspnes, Zoë Diamadi, Michael J. Fischer, and René Peralta for their paper "Computation in networks of passively mobile finite-state sensors", *Distributed Computing*, volume 18, number 4, 2006, pages 235–253.
- The *2019 Principles of Distributed Computing Doctoral Dissertation Award*, presented at DISC 2020, to Yannic Maus for his dissertation "The Power of Locality: Exploring the Limits of Randomness in Distributed Computing", written under the supervision of Prof. Fabian Kuhn at the University of Freiburg and to Yi-Jun Chang for his dissertation "Locality of Distributed Graph Problems", written under the supervision of Prof. Seth Pettie at the University of Michigan.

Despite the COVID-19 pandemic, we received this year a record number of submissions in response to the call for papers: 170 regular paper submissions, and 15 brief announcement submissions. We had a program committee with 32 members, and the committee was assisted by 150 external reviewers. All submissions were evaluated by at least three reviewers; in total, 616 reviews were collected. The program committee used double-blind peer review: the submissions were anonymous and the PC members and external reviewers did not see the names of the authors. The program committee decided to accept 39 regular submissions (an acceptance rate of 23%) and 15 brief announcements for presentation at DISC 2020.

The committee selected the following paper as the recipient of the *DISC 2020 Best Paper Award*:

Improved Bounds for Distributed Load Balancing,
by Sepehr Assadi, Aaron Bernstein and Zachary Langley

and the following paper as the recipient of the *DISC 2020 Best Student Paper Award*:

Intermediate Value Linearizability: A Quantitative Correctness Criterion,
by Arik Rinberg and Idit Keidar

DISC 2020 Best Reviewer Award was presented to Michal Dory and Laurent Feuilloley.

Three workshops were held in conjunction with DISC 2020:

- ADGA: Workshop on Advances in Distributed Graph Algorithms
  (chair: Jara Uitto)
- CELLS: Computing among Cells
  (chairs: Matthias Fuegger, Manish Kushwaha and Thomas Nowak)
- Blockchain Sharding and Interoperability among Shards
  (chairs: Fatemeh Shirazi and Eleftherios Kokoris Kogias)

A tutorial on TLA+ was given by Leslie Lamport and Stephan Merz.

I would like to thank all conference participants and everyone who contributed to DISC 2020: the authors of the submitted papers, PC members and external reviewers, keynote speakers, members of the organizing committee, workshop organizers, and members of the award committees. I would also like to thank the members of the steering committee, former chairs and many other members of the community for their valuable assistance and suggestions, EATCS for their financial support, and the staff at Schloss Dagstuhl – Leibniz-Zentrum für Informatik for the hard work they did with preparing these proceedings.

October 2020

Hagit Attiya
DISC 2020 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

Hagit Attiya — Technion, Israel

Program Committee

James Aspnes — Yale (USA)
Hagit Attiya (chair) — Technion (Israel)
Leonid Barenboim — Open University of Israel (Israel)
Petra Berenbrink — Universität Hamburg (Germany)
Armando Castañeda — UNAM (Mexico)
Keren Censor-Hillel — Technion (Israel)
Tudor David — Oracle Labs (Switzerland)
Carole Delporte-Gallet — Université Paris Diderot (France)
David Doty — University of California, Davis (USA)
Aleksandar Dragojevic — Microsoft (UK)
Constantin Enea — Université Paris Diderot (France)
Matthias Fitzi — IOHK (Switzerland)
Paola Flocchini — University of Ottawa (Canada)
Sebastian Forster — University of Salzburg (Austria)
Luisa Gargano — Università di Salerno (Italy)
Eric Goubault — École Polytechnique (France)
Guy Gueta — VMware Research (Israel)
Joe Izraelevitz — University of Colorado, Boulder (USA)
Anne-Marie Kermarrec — EPFL (Switzerland)
Alex Kogan — Oracle Labs (USA)
Dariusz R. Kowalski — Augusta University (USA)
Moti Medina — Ben-Gurion University of the Negev (Israel)
Alessia Milani — University of Bordeaux (France)
Adam Morrison — Tel Aviv University (Israel)
Lata Narayanan — Concordia University (Canada)
Thomas Nowak — Université Paris-Saclay (France)
Dominik Pająk — Wroclaw University of Science and Technology (Poland)
Boaz Patt-Shamir — Tel Aviv University (Israel)

34th International Symposium on Distributed Computing (DISC 2020).
Editor: Hagit Attiya
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Andrzej Pelc	Université du Québec en Outaouais (Canada)
Christian Scheideler	University of Paderborn (Germany)
Jennifer Welch	Texas A&M University (USA)
Moti Yung	Google Inc. and Columbia University (USA)

Steering Committee

Hagit Attiya	Technion (Israel)
Fabian Kuhn	U. Freiburg (Germany)
Yoram Moses (chair)	Technion (Israel)
Merav Parter (treasurer)	Weizmann Institute (Israel)
Andréa Richa (vice chair)	Arizona State U. (USA)
Ulrich Schmid	TU Wien (Austria)
Jukka Suomela	Aalto U. (Finland)

Organizing Committee

Fabian Kuhn (general chair)	U. Freiburg (Germany)
Moti Medina (workshops and tutorials chair)	Ben-Gurion University of the Negev (Israel)
Jukka Suomela (virtuality chair)	Aalto U. (Finland)

External Reviewers

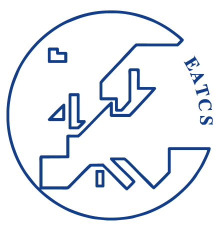
Ittai Abraham
Udit Agarwal
Emmanuelle Anceaume
Sepehr Assadi
Chen Avin
Christian Badertscher
Philipp Bamberger
Joffroy Beauquier
Luca Becchetti
Soheil Behnezhad
Ohad Ben Baruch
Subhash Bhagat
Martin Bohm
Dominik Bojko
Silvia Bonomi
Sebastian Brandt
Trevor Brown
Christian Cachin
Bogdan Chlebus
Jules Chouquet
Michele Ciampi
Nachshon Cohen
Sarel Cohen
Gennaro Cordasco
Juan Antonio Cordero Fuertes
Sandro Coretti
Paolo D'Arco
Sanjoy Dasgupta
Gianluca De Marco
Antonella Del Pozzo
Mahsa Derakhshan
Giuseppe Antonio Di Luna
Michael Dinitz
Stephan Dobrev
Michal Dory
Dana Drachsler Cohen
Andrew Drucker
Fabien Dufoulon
Mahsa Eftekhari
Michael Elkin
Ulrich Fahrenberg
Michael Feldmann
Laurent Feuilloley
Yuval Filmus
Arnold Filtser
Manuela Fischer
Orr Fischer
Matthias Fuegger
Chaya Ganesh
Álvaro García-Pérez

Cyril Gavoille
Paweł Gawrychowski
Peter Gaži
George Giakkoupis
Seth Gilbert
Gramoz Goranci
Alexey Gotsman
Thorsten Götte
Vincent Gramoli
Rachid Guerraoui
Andreas Haas
Magnus M. Halldorsson
Marc Heinrich
Danny Hendler
Eshcar Hillel
Kristian Hinnenthal
Martin Hirt
Yufan Huang
David Ilcinkas
Damien Imbs
Taisuke Izumi
Burman Janna
Colette Johnen
Omri Kahalon
Nikolaos Kallimanis
Idit Keidar
Artem Khyzha
Juno Kim
Ralf Klasing
Peter Kling
Marek Klonowski
Sven Köhler
Christian Konrad
Danny Krizanc
Petr Kuznetsov
François Le Gall
Jeremy Ledent
Dean Leitersdorf
Christoph Lenzen
Chen-Da Liu Zhang
Julian Loss
Tzalik Maimon
Frederik Mallmann-Trenn
Virendra Marathe
Yannic Maus
Othon Michail
William Moses, Jr.
Achour Mostefaoui
Miguel A. Mosteiro
Kartik Nayak
Krzysztof Nowicki
Dennis Olivetti
Jan Olkowski
Jaroslav Opatrny
Gal Oren
Gopal Pandurangan
Denis Pankratov
Matej Pavlovic
Ami Paz
Matthieu Perrin
Mor Perry
Giuseppe Persiano
Seth Pettie
Volodymyr Polosukhin
Nuno Preguica
Adele Rescigno
Stephane Rovedakis
Joel Rybicki
Jared Saia
Hayk Saribekyan
Thomas Sauerwald
Stefan Schmid
Gregory Schwartzman
Michael L. Scott
Michele Scquizzato
Siddhartha Sen
Adi Serendinschi
Alexander Setzer
Eric Severson
Shay Solomon
David Soloveichik
Alexander Spiegelman
Grzegorz Stachowiak
Patryk Stopyra
Aikaterini-Panagiota Stouka
Hsin-Hao Su
Lili Su
Edward Talmage
Tigran Tonoyan
Amitabh Trehan
Sara Tucci Piergiovanni
Schmid Ulrich
Przemysław Uznański
Giovanni Viglietta
Koichi Wada
Daniel Warner

Roger Wattenhofer
Haosen Wen
Josef Widder
Marcin Witkowski
Cong Xie
Yukiko Yamauchi
Avishay Yanai
Igor Zablotchi

Sponsoring Organizations

DISC 2020 acknowledges the use of **HotCRP** for handling submissions and managing the review process, **LIPIcs** for producing and publishing the proceedings, and **Zulip** for providing virtual interaction space for conference participants.



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

Bronze sponsors for DISC 2020:

2020 Edsger W. Dijkstra Prize in Distributed Computing

The Edsger W. Dijkstra Prize in Distributed Computing is awarded for 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. It is sponsored jointly by the ACM Symposium on Principles of Distributed Computing (PODC) and the EATCS Symposium on Distributed Computing (DISC). The prize is presented annually, with the presentation taking place alternately at PODC and DISC.

The committee decided to award the 2020 Edsger W. Dijkstra Prize in Distributed Computing to

Dana Angluin, James Aspnes, Zoë Diamadi, Michael J. Fischer, and René Peralta

for their paper:

Computation in networks of passively mobile finite-state sensors,
Distributed Computing 18(4): 235-253 (2006)

A preliminary version of this paper appeared in the proceedings of the *Twenty-Third Annual ACM Symposium Principles of Distributed Computing* (PODC), 2004, pages 290–299.

This seminal paper introduces and initiates the study of *population protocols*. The computational setting consists of agents whose resources are limited to a very small amount of memory and computational power. When two agents interact, they change their local states according to their previous local states through a simple transition function. This is arguably one of the simplest models of distributed computing yet, surprisingly, non-trivial predicates on the inputs of the agents can be computed. The paper defined the concept of stable computation of a function or predicate in the population model, showed that any predicate definable in Presburger arithmetic can be computed by a population protocol, and provided constructions for fundamental objects such as counters and timers in the probabilistic interaction model.

Through its novelty and technical quality, the paper of Angluin, Aspnes, Diamadi, Fischer and Peralta initiated an expansive new sub-field in distributed computing. Population protocols have revealed a rich landscape of algorithmic and lower bound techniques in the context of fundamental distributed computing tasks. An powerful aspect of this paper is in providing an elegant and concise model that faithfully captures a variety of real-world processes, ranging from wireless sensor networks, to gene regulatory networks and chemical reaction networks. This abstracts away much of the complexity of distributed computation in these settings, without shedding its non-trivial technical essence.

This foundational paper led to extensive follow-up work regarding the computation power of population protocols, their complexity costs, their application in dynamic networks, and their usage in chemical and biological systems. Closest to the thrust of the original paper is the investigation of the *computational power* of the population model. Extensive work studied how the amount of memory available to each agent affects the system's computational power, and whether *efficient* general computational constructions exist.

The paper also paved the way to investigating bounds for the computation of specific fundamental predicates, in terms of natural cost measures such as time and space. A flurry of progress in this area yielded tight or almost tight bounds for tasks such as majority, plurality,

or leader election in population protocols. Several of these protocols relied on *epidemic* propagation of information. This work led to the discovery of new tools, in particular, at the boundary between distributed computing, randomized algorithms, and probability theory. In addition, the paper was one of the first to study computation in a dynamic network, whose topology is highly variable, and generally unpredictable. The resulting algorithmic theory of dynamic networks has since become a new and active research area in theoretical computer science.

Finally, population protocols also has had significant impact outside the traditional boundaries of distributed computing. For instance, the population protocol model has been shown in follow-up work to be formally equivalent, under technical conditions, to the classic *chemical reaction network* (CRN) model, while providing a much simpler interface. This has been a key contributing factor to the significant progress in this area in recent years. Moreover, the use of population protocols has had impact in the context of *DNA Computing and Molecular Programming.* In this recent and flourishing research area, population protocols are used to express molecular programs that are coded in synthetic DNA.

In summary, the pioneering work on population protocols introduced in this paper provided a general and elegant concept, created a new sub-field in distributed computing and has had an impact outside the distributed computing area.

Hagit Attiya (chair), *Technion*
Christian Cachin, *University of Bern*
Rachid Guerraoui, *EPFL*
Nancy Lynch, *MIT*
Yoram Moses, *Technion*
Paul Spirakis, *University of Liverpool* and *University of Patras*
Alex Schwarzmann, *Augusta University*

2020 Principles of Distributed Computing Doctoral Dissertation Award

The winners of the 2019 Principles of Distributed Computing Doctoral Dissertation Award are

Dr. Yi-Jun Chang for his dissertation *Locality of Distributed Graph Problems*, written under the supervision of Prof. Seth Pettie at the University of Michigan.

and

Dr. Yannic Maus for his dissertation *The Power of Locality: Exploring the Limits of Randomness in Distributed Computing*, written under the supervision of Prof. Fabian Kuhn at the University of Freiburg.

The theses of Dr. Chang and Dr. Maus have played a key role in the recent, rapid development of the theory of distributed graph algorithms and network computing. Both of the theses have significantly advanced our understanding of the distributed computational complexity of many key problems (e.g. graph coloring and splitting). In addition, they have made groundbreaking contributions to the development of distributed computational complexity theory in general.

These theses have introduced highly insightful concepts (e.g. the SLOCAL model) and new intriguing graph problems (e.g. hierarchical coloring), they have developed novel proof techniques (e.g. pumping arguments), and they have proved surprising results (e.g. gap theorems and completeness). Put together, they have dramatically changed the way in which researchers working in this field reason about distributed computational complexity. The theses have been a driving force in this research area, and the concepts and ideas introduced in these theses have already led to major breakthroughs.

The work presented in both dissertations has been published in a remarkably large number of papers, has been presented at top conferences, and has received wide recognition, including multiple best paper awards.

Due to the highly significant role these two theses have played in the development of the field of distributed computing, the award committee unanimously selected them as the winners of the 2020 Principles of Distributed Computing Doctoral Dissertation Award, presented at DISC 2020.

The award is sponsored jointly by the ACM Symposium on Principles of Distributed Computing (PODC) and the EATCS Symposium on Distributed Computing (DISC). It is presented annually, with the presentation taking place alternately at PODC and DISC.

2020 Award Committee:
Faith Ellen, *University of Toronto*
Pierre Fraigniaud, *CNRS and University Paris Diderot*
David Peleg (chair), *The Weizmann Institute*
Jukka Suomela, *Aalto University*

34th International Symposium on Distributed Computing (DISC 2020).
Editor: Hagit Attiya
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Improved Bounds for Distributed Load Balancing

Sepehr Assadi
Rutgers University, New Brunswick, NJ, USA
sepehr.assadi@rutgers.edu

Aaron Bernstein
Rutgers University, New Brunswick, NJ, USA
bernstei@gmail.com

Zachary Langley
Rutgers University, New Brunswick, NJ, USA
zach.langley@rutgers.edu

Abstract

In the load balancing problem, the input is an n-vertex bipartite graph $G = (C \cup S, E)$ – where the two sides of the bipartite graph are referred to as the clients and the servers – and a positive weight for each client $c \in C$. The algorithm must assign each client $c \in C$ to an adjacent server $s \in S$. The load of a server is then the weighted sum of all the clients assigned to it. The goal is to compute an assignment that minimizes some function of the server loads, typically either the maximum server load (i.e., the ℓ_∞-norm) or the ℓ_p-norm of the server loads. This problem has a variety of applications and has been widely studied under several different names, including: scheduling with restricted assignment, semi-matching, and distributed backup placement.

We study load balancing in the distributed setting. There are two existing results in the CONGEST model. Czygrinow et al. [DISC 2012] showed a 2-approximation for unweighted clients with round-complexity $O(\Delta^5)$, where Δ is the maximum degree of the input graph. Halldórsson et al. [SPAA 2015] showed an $O(\log n / \log \log n)$-approximation for unweighted clients and $O(\log^2 n / \log \log n)$-approximation for weighted clients with round-complexity polylog(n).

In this paper, we show the first distributed algorithms to compute an $O(1)$-approximation to the load balancing problem in polylog(n) rounds:

- In the CONGEST model, we give an $O(1)$-approximation algorithm in polylog(n) rounds for unweighted clients. For weighted clients, the approximation ratio is $O(\log n)$.
- In the less constrained LOCAL model, we give an $O(1)$-approximation algorithm for weighted clients in polylog(n) rounds.

Our approach also has implications for the standard sequential setting in which we obtain the first $O(1)$-approximation for this problem that runs in near-linear time. A 2-approximation is already known, but it requires solving a linear program and is hence much slower. Finally, we note that all of our results *simultaneously* approximate all ℓ_p-norms, including the ℓ_∞-norm.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Load Balancing, Distributed Algorithms, Matching, Semi-Matching

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.1

Related Version A full version of the paper is available at https://arxiv.org/abs/2008.04148.

Funding *Aaron Bernstein*: This work was done while funded by NSF CAREER Grant 1942010 and Simons Collaboration on Algorithms and Geometry.

© Sepehr Assadi, Aaron Bernstein, and Zachary Langley;
licensed under Creative Commons License CC-BY

1 Introduction

In this paper, we study the *load balancing* problem. The input is a bipartite graph $G = (C \cup S, E)$, where we refer to the sets C and S the *clients* and *servers*, respectively. The goal is to find an assignment of clients to servers such that no server is assigned too many clients. To be more precise, we define the load of a server in an assignment to be the number of clients assigned to it, and we are interested in finding an assignment that minimizes the maximum load of any server (or minimizes the ℓ_p-norm of the server loads – we will discuss this objective more later in the introduction).

The load balancing problem has a rich history in the scheduling literature as the *job scheduling with restricted assignment problem* [14, 6, 19, 13], in the distributed computing literature as the *backup placement problem* [12, 22, 3], in sensor networks [23, 21] and peer-to-peer systems [18, 26, 25] as the *load-balanced data gathering tree construction problem*, and more generally as a relaxation of the bipartite matching problem known as the *semi-matching problem* [13, 8, 9, 17]. We refer the reader to [13, 9, 12] for more background.

Our primary focus in this paper is on the load balancing problem in a distributed setting, where clients and servers correspond to separate nodes in a network. Communication through the network happens in synchronous rounds, where in each round, every node can send $O(\log n)$ bits to its neighbors over any of its incident edges (formally, we work in the CONGEST model – see Section 2 for more details). In the distributed setting, the load balancing problem generalizes the distributed backup placement problem with replication factor one (introduced in [12]), where the nodes (corresponding to clients) in a distributed network may have memory faults and therefore wish to store backup copies of their data at neighboring nodes (corresponding to servers). Since backup-nodes may incur faults as well, the number of nodes that select the same backup-node should be minimized. See the full version of this paper [1] for the exact formulation of the distributed backup placement problem and for how some of our results extend to the more general version of the problem with arbitrary replication factor.

A simple distributed algorithm for the load balancing problem in which the nodes myopically reassign themselves to a server with smaller load eventually converges to an $O(\frac{\log n}{\log \log n})$-approximation, where n is the number of nodes in the network [10, 16], but as was shown in Halldórsson et al. [12], the algorithm requires $\Omega(\sqrt{n})$ rounds. The same paper [12] shows a way of circumventing this costly process and gives a distributed algorithm that achieves the same $O(\frac{\log n}{\log \log n})$-approximation in only $\mathrm{polylog}(n)$ rounds. On the other hand, Czygrinow et al. [8] show a distributed $O(1)$-approximation (precisely, a 2-approximation) that requires $O(\Delta^5)$ rounds, where Δ is the maximum degree of a node in the network. This algorithm is highly efficient for low-degree networks but is again too expensive for high-degree graphs.

This state-of-affairs is the starting point of our work: *Can we obtain the best of both worlds, namely, an $O(1)$-approximation algorithm in* $\mathrm{polylog}(n)$ *rounds?*

Our first contribution. Our first main contribution in this paper is an affirmative answer to this question.

▶ **Result 1** (Formalized in Theorem 6)**.** *We give an $O(1)$-approximate randomized distributed algorithm for load balancing in the* CONGEST *model that runs in $O(\log^5 n)$ rounds.*

At the core of our algorithm is a new structural lemma for the load balancing problem. Informally speaking, we show that eliminating all "short augmenting paths" of length $O(\log n)$ is sufficient to assign *all* clients to servers with load a constant factor as much as

the optimum (**Lemma 3**). In conjunction with ideas from [12], this effectively reduces the load balancing problem to that of finding a matching with no short augmenting paths, which can be solved using the by-now standard algorithm of Lotker et al. [20].

Our second contribution. Next, we consider the *weighted* load balancing problem in which every client comes with a weight. The load of a server is then the total weight of the clients assigned to it. The goal, as before, is to minimize the maximum load of any server. Halldórsson et al. [12] also studied the weighted problem and gave an $O(\frac{\log^2 n}{\log\log n})$-approximation in polylog(n) rounds using a simple reduction to the unweighted case.

Using the same weighted-to-unweighted reduction, our algorithm in Result 1 also implies an $O(\log n)$-approximation for the weighted load balancing problem in polylog(n) rounds of the CONGEST model. Our main technical contribution in this paper is a new algorithm for this problem that achieves an $O(1)$-approximation in the less constrained LOCAL model, in which communication over edges in each round is unbounded.

▶ **Result 2** (Formalized in Theorem 15). *We give an $O(1)$-approximate randomized distributed algorithm for weighted load balancing in $\log^3 n$ rounds of the* LOCAL *model.*

Our LOCAL algorithm consists of two main components: a distributed algorithm for (approximately) solving a relaxed version of the problem where each client c with weight $w(c)$ should be assigned to $w(c)$ adjacent servers with multiplicity – a *split assignment* – and a novel distributed rounding procedure. Using our structural result in Lemma 3, we can find a split assignment by approximately solving (or rather, eliminating short augmenting paths in) a generalized b-matching problem with *edge capacities*. We are not aware of any efficient algorithm for this problem in the CONGEST model, but we can show that a simple extension of the work of [20] can solve this problem in polylog(n) rounds in the LOCAL model. The rounding step is also based on a new application of our Lemma 3 that allows us to circumvent the typical use of "cycle canceling" procedures for rounding fractional matching LP solutions into integral ones, which do not translate to efficient distributed algorithms.

We now turn to two important extensions of Results 1 and 2. The first is the more general problem of all-norm load balancing, and the second is a fast sequential algorithm.

Approximating all norms. Recall that our goal in the load balancing problem has been to minimize the maximum load of any server. Assuming we denote the loads of servers under some assignment A by a vector $L_A := [L_A(s_1), L_A(s_2), \ldots, L_A(s_n)]$ for all $s_i \in S$, minimizing the maximum server load is equivalent to minimizing $\|L_A\|_\infty$, i.e., the ℓ_∞-norm of L_A. Depending on the application, however, miniziming this norm may not be the most natural notion of a "balanced" assignment; if some server requires vastly more load than the other servers, an ℓ_∞-norm-minimizing assignment may put needlessly large load on those other servers.

As a result, it is natural to consider minimizing some other ℓ_p-norm of L_A for some $p \geq 1$. This is done, for instance, in [13, 9, 17], which considered ℓ_2-norms. An even more general objective is the *all-norm* problem, studied in [2, 5, 7, 13], where the goal is to *simultaneously* optimize with respect to every ℓ_p-norm. These results compute an assignment which is an $O(1)$-approximation (or even optimal) simultaneously with respect to all ℓ_p-norms, including $p = \infty$ (*a priori*, even the existence of such an assignment is not clear).

All of our results extend to the all-norm problem without any increase in approximation factor or round-complexity. In particular, in the CONGEST model, we give randomized distributed $O(1)$- and $O(\log n)$-approximation algorithms for all-norm load balancing

in polylog(n) rounds, in the unweighted and weighted variant of the problem, respectively (**Theorem 11**). In the LOCAL model, the approximation ratio for the weighted problem can be reduced to $O(1)$ as well (**Theorem 15**).

Faster sequential algorithms. Finally, we show that our new approach to weighted load balancing can also be used to design a near-linear time algorithm for this problem in the *sequential* setting. We give a deterministic $O(m \log^3(n))$ time algorithm for the $O(1)$-approximate all-norm load balancing problem in the sequential setting (**Theorem 20**).

Previously, a deterministic $O(m\sqrt{n} \log n)$ time for the exact problem in case of unweighted graphs was given in [9]. The weighted variant of the problem is NP-hard [2]; 2-approximate algorithms were shown in [2] and [7], but they are based on solving, respectively, the linear and convex programming relaxations of the problem exactly using the ellipsoid algorithm, and thus are much slower than the algorithm we present.

2 Preliminaries

Notation. For any function $f : A \to \mathbb{N}$ and $B \subseteq A$, we use the notation $f(B) = \sum_{b \in B} f(b)$ to sum f over all elements in B. For any integer $t \geq 1$, we denote $[t] := \{1, \ldots, t\}$.

Throughout, we assume $G = (C \cup S, E)$ is a bipartite graph. We refer to C and S as the *clients* and the *servers*, respectively. We let uv denote the edge between vertices u and v and let $\delta(v)$ denote the set of edges incident to a vertex v. We use n as number of vertices in G and m as the number of edges in G.

Load balancing. In the load balancing problem, the input is a bipartite graph $G = (C \cup S, E)$ together with a client weight function $w : C \to [W]$. The output is an assignment $A : C \to S$ mapping every client to one of its adjacent servers. The *load* $L_A(s)$ of a server $s \in S$ under assignment A is the sum of the weights of the clients assigned to it: $L_A(s) = w(A^{-1}(s))$. The *maximum load* of an assignment A is the maximum load of any server under A. We refer to the problem of computing an assignment of minimum load as the *(weighted) min-max load balancing problem.*

As mentioned in the introduction, the min-max objective can be generalized by considering any ℓ_p-norm of L_A, defined as $\|L_A\|_p = \left(\sum_{s \in S} (L_A(s))^p\right)^{1/p}$. For brevity, we also use the notation $\|A\|_p := \|L_A\|_p$. In the language of norms, the min-max objective corresponds to minimizing the load vector's ℓ_∞-norm. When the goal is to find an assignment A that simultaneously minimizes $\|A\|_p$ for all $p \geq 1$, including $p = \infty$, the problem is called the *(weighted) all-norm load balancing problem.* Prior results in [2, 5, 7, 13] show the *existence* of an assignment that can (approximately) minimize all these norms simultaneously. In particular, we use the following result due to Harvey et al. [13] in our proofs (see also [5]).

▶ **Lemma 1** ([13]). *Given any instance of the unweighted load balancing problem, there exists an assignment A^* that simultaneously minimizes $\|A^*\|_p$ for all $p \geq 1$, including $p = \infty$.*

***b*-matchings.** In addition to assignments, we will also work with *b-matchings.* For a vertex capacity function $b : V \to \mathbb{Z}^+$, a b-matching is an assignment $x : E \to \mathbb{Z}^+$ of integer multiplicities to edges so that for every vertex v, the sum $x(\delta(v))$ of the multiplicities of the edges incident to v does not exceed $b(v)$.

Since we will focus solely on the case when G is bipartite and $V = C \cup S$, it will be convenient to split b into two separate capacity functions, one for the clients and one for the servers. We use $\kappa : C \to \mathbb{Z}^+$ to denote the *client capacities* and $\tau : S \to \mathbb{Z}^+$ to denote the

server capacities. A (κ, τ)-*matching* is then a function $x : E \to \mathbb{Z}^+$ assigning multiplicities to edges such that

$$\sum_{s \in N(c)} x(cs) \leq \kappa(c) \tag{1}$$

for every client c and

$$\sum_{c \in N(s)} x(cs) \leq \tau(s)$$

for every server s. A (κ, τ)-matching is *client-perfect* if (1) holds with equality for all $c \in C$. We say that a server s (resp. client c) is x-*saturated* if $x(\delta(s)) = \tau(s)$ (resp. $x(\delta(c)) = \kappa(c)$). If a vertex is not x-saturated, then it is x-*unsaturated*. An x-*augmenting path* is a path $v_1, \ldots, v_{2k+1}$ such that v_1 and v_{2k+1} are x-unsaturated and $x(v_{2i+1}v_{2i+2}) > 0$ for all $0 \leq i < k$.

We will make repeated use of the following simple remark.

▶ Remark 2. When all client weights are one (the unweighted case), a client-perfect $(1, \tau)$-matching induces an assignment of maximum load at most $\max_{s \in S} \tau(s)$, and vice versa.

Note that the remark does not generalize to weighted clients; under a (w, τ)-matching, a client may be split across multiple servers, which does not correspond to a proper assignment.

The LOCAL and CONGEST models. In both the LOCAL and the CONGEST models of distributed computation, each vertex of the input graph hosts a processor that initially only knows its neighbors and its weight. Following a standard assumption, we assume that all vertices know n and the maximum weight W. Computation proceeds in synchronous rounds; in each round, vertices may send messages to their neighbors and then receive messages from their neighbors in lockstep. Local computation is free – we are only interested in the *round complexity*, the number of rounds required by the algorithm.

The LOCAL and CONGEST models differ in that in the LOCAL model, vertices can send and receive arbitrarily large messages to and from their neighbors, while in the CONGEST model, the communication between adjacent vertices in each round is capped at $O(\log n)$.

3 A Structural Lemma

A crucial component of our results is a structural observation about approximate (κ, τ)-matchings in the context of the load balancing problem, which is inspired by results from online load balancing [11, 4]: if a graph contains *some* client-perfect (κ, τ)-matching, then *every* $(\kappa, 2\tau)$-matching is either client-perfect or can be augmented via an augmenting path of logarithmic length. Formally, and more generally, we have the following lemma.

▶ **Lemma 3.** *If G contains a client-perfect (κ, τ)-matching and x is a $(\kappa, \alpha\tau)$-matching for $\alpha > 1$, then either x is client-perfect or there is an x-augmenting path of length at most $2\lceil \log_\alpha \tau(S) \rceil + 1$.*

Proof. Suppose G contains a client-perfect (κ, τ)-matching x^*. To simplify the discussion, we define a directed multigraph D on $V(G)$ whose arcs are oriented edges in the support of x and x^* as follows. For every $cs \in E(G)$ with $c \in C$ and $s \in S$, D has $x(cs)$ copies of the arc (s, c) and $x^*(cs)$ copies of the arc (c, s) and no other arcs. Notice that every directed path in D starting at an x-unsaturated client and ending at an x-unsaturated server corresponds to an x-augmenting path in G.

Suppose x is not client-perfect and let $c \in C$ be an x-unsaturated client. Let $k \in \mathbb{N}$ be fixed and define U_k to be the set of vertices reachable via a walk of length k from c in D. Call U_k *full* if u is x-saturated for all $u \in U_k$.

The lemma follows from two simple claims:

1. if U_{2k+1} is not full, then G contains an x-augmenting path of length at most $2k+1$; and
2. if U_{2k+1} is full, then $\tau(U_{2k+3}) \geq \alpha\tau(U_{2k+1})$.

The first claim follows from the fact that a directed walk contains a directed path with the same endpoints and from the correspondence noted earlier between directed paths with unsaturated endpoints in D and augmenting paths in G.

We proceed to the second claim. If $s \in U_{2k+1}$ and U_{2k+1} is full, then s is x-saturated and the out-degree of s is $\alpha\tau(s)$. Thus, the total out-degree of U_{2k+1} – and also the total in-degree of U_{2k+2} – is $\alpha\tau(U_{2k+1})$. Now we use the fact that the out-degree of a client $c \in U_{2k+2}$ is at least as large as its in-degree. This follows simply from the fact that the in-degree must be at most $\kappa(c)$, and since x^* is client-perfect, the out-degree is exactly $\kappa(c)$. Following the arcs once more, the total in-degree of U_{2k+3} is at least $\alpha\tau(U_{2k+1})$. Finally, since the in-degree of U_{2k+3} is also point-wise less than τ, we have $\alpha\tau(U_{2k+1}) \leq \tau(U_{2k+3})$.

Now we show how the two claims together imply the lemma. If any U_{2i+1} for $i \leq \lceil\log_\alpha(\tau(S))\rceil$ is not full, we are done by the first claim. Otherwise, the sums of capacities grow exponentially starting with $\tau(U_1) \geq 1$. Inductively, $|U_{2i+1}| \geq \alpha^i$ for all $i \in \mathbb{N}$. For $k = \lceil\log_\alpha \tau(S)\rceil$, therefore, we have $\tau(U_{2k+3}) \geq \alpha\tau(S)$, a contradiction. Thus, not all $\{U_{2i+1}\}$ are full. ◀

4 Unweighted Load Balancing

Assuming an algorithm to eliminate augmenting paths up to a certain length efficiently, the structural lemma from the previous section almost immediately implies an algorithm for the unweighted load balancing problem. To obtain an algorithm for eliminating short augmenting paths, we use the following lemma which is implied by Lemma 24 in [12].

▶ **Lemma 4** ([12]). *There exists an $O(k^3 \log n)$-round randomized algorithm in the* CONGEST *model that, with high probability, given a graph $G = (C \cup S, E)$, a positive integer k, and server capacity function τ, computes a $(1,\tau)$-matching with no augmenting paths of length less than k.*

The proof of Lemma 4 combines two existing results. The algorithm of Lotker et al. [20] computes a (1,1)-matching with no augmenting paths of length $\leq k$ in $O(k^3 \log n)$ rounds. Halldórsson et al. [12] then show a black-box extension from (1,1)-matching to (1, τ)-matching which does not increase the round-complexity; see [12] for more details.

▶ Remark 5. Both our algorithm and the algorithm of [12] use the above lemma as a starting point. But the algorithm of [12] only removes short augmenting paths to ensure that the $(1,\tau)$-matching is approximately optimal. Since a near-optimal matching is still not an assignment (it is not client-perfect), they then use a different set of tools to convert an approximate $(1,\tau)$-matching to an $O(\log n/\log\log n)$-approximate assignment.

Our analysis, by contrast, directly exploits the non-existence of short augmenting paths via Lemma 3. We thus avoid the additional conversion of [12], which leads to a better approximation ratio, as well as a simpler algorithm.

Approximating the ℓ_∞-norm (the min-max load balancing problem). Let B^* be the optimum ℓ_∞-norm. We will first describe an algorithm that assumes as input some $B \geq 2B^*$. The algorithm begins by using Lemma 4 to compute a $(1, B)$-matching x with no augmenting paths of length $4\lceil \log_2 n \rceil + 1$. The sum of the server capacities is at most nB, and since clients have unitary weight, we can assume $B \leq n$. Therefore, by Lemma 3, since there are no x-augmenting paths of length $2\lceil \log_2(nB) \rceil + 1 \leq 4\lceil \log_2 n \rceil + 1$, we know that x is necessarily client-perfect. A client c can now assign itself to the vertex it is matched to under x.

To remove the assumption that we are given a $B \geq 2B^*$, we run the algorithm above $\log n$ times with $B = 1, 2, 4, \ldots, n$. For every run where $B \geq 2B^*$, the algorithm will successfully assign every client. Note, however, that in the distributed setting, there is no efficient way for the clients to determine the smallest B for which the algorithm successfully matched every client. Instead, each client c *locally* assigns itself according to the run with smallest B that succeeded—i.e., according to the first run in which c was matched. We show that the resulting assignment has maximum load at most $8B^*$. See Algorithm 1 for a concise treatment.

Algorithm 1 Approximate unweighted load balancing in the CONGEST model.

```
1 for B ∈ {1, 2, 4, ..., n} do
2 |  compute a (1, B)-matching x_B with no augmenting paths of length 4⌈log n⌉ + 1
3 end
4 each client c locally finds the minimum B such that c is matched in x_B and assigns
    itself to the server it is matched to in x_B
```

▶ **Theorem 6.** *In the CONGEST model, there is a randomized algorithm (Algorithm 1) that with high probability computes an 8-approximation to the min-max load balancing problem in $O(\log^5 n)$ rounds.*

Proof. First observe that since augmenting paths with respect to a $(1, n)$-matching have length at most 1, all clients are assigned in x_n. Therefore the algorithm always outputs an assignment of all clients.

Let B^* be the optimal maximum load and B be the smallest power of 2 that is at least $2B^*$ (we have $B \leq 4B^*$). The $(1, B)$-matching x_B computed in the main loop of Algorithm 1 will be client-perfect by Lemma 3, and so no client will assign itself according to x'_B for $B' > B$. A single server is only assigned at most i clients from x_i, and since $x_1, x_2, x_4, \ldots, x_B$ are the only assignments in play, the total load of a server under any combination of these assignments is at most $1 + 2 + 4 + \cdots + B < 2B \leq 8B^*$. Thus every server has load at most $8B^*$.

Finally, each x_B is computed in $O(\log^4 n)$ rounds with high probability by Lemma 4. We compute them sequentially, resulting in total round complexity of $O(\log^5 n)$. ◀

▶ Remark 7. In the LOCAL model, the round complexity of Algorithm 1 is $O(\log^3 n)$. One log factor is shaved off of Lemma 4 because the algorithm of [20] for finding a (1,1)-matching is faster in the LOCAL model. The second log factor is shaved off by running the for-loop of Algorithm 1 in parallel for every B.

Approximating all ℓ_p-norms simultaneously (the all-norm load balancing problem). The assignment produced by Algorithm 1 in fact does more than approximate the optimal ℓ_∞-norm; it also simultaneously approximates every ℓ_p-norm for $p \geq 1$, as we will now show.

Recall that by Lemma 1, when clients are unweighted, there is an all-norm optimal assignment that simultaneously minimizes the ℓ_p-norm for all $p \geq 1$, including $p = \infty$. Let $\mathcal{A}^*$ be the set of all all-norm optimal assignments; by Lemma 1, the set $\mathcal{A}^*$ is non-empty. We need the following key definition.

▶ **Definition 8.** *The* level $\ell(s)$ *of a server s is the maximum load of s over all assignments in $\mathcal{A}^*$, i.e., $\ell(s) := \max_{A \in \mathcal{A}^*} L_A(s)$. The* level $\ell(c)$ *of a client c is $\ell(c) := \max_{A \in \mathcal{A}^*} L_A(A(c))$.*

▶ **Lemma 9.** *For $i \in [n]$, let $C_i \subseteq C$ be the set of clients at level i and let $S_i \subseteq S$ be the set of servers at level i. There are no edges in G from C_i to S_j for any $j < i$.*

Proof. Fix a client $c \in C_i$ and let A be an all-norm optimal assignment such that $L_A(A(c)) = i$. Suppose to obtain a contradiction that $N(c) \cap S_j \neq \emptyset$ and let $s \in N(c) \cap S_j$. Consider the assignment A' which maps c to s and is otherwise identical to A. Comparing the load vector of A to A', one entry of the load vector moves from i to $i-1$, another entry moves from j to $j+1$, and the rest remain unchanged. Since $j < i$, it follows that $\|A'\|_p \leq \|A\|_p$ for all p. If $\|A'\|_p < \|A\|_p$ for some p, then A is not all-norm optimal, a contradiction. Otherwise, the norms are equal for all p. But then there is an all-norm optimal assignment, namely A', in which s has level $j+1$, contradicting that the level of s is j. ◀

▶ **Lemma 10.** *If a client $c \in C$ has level at most ℓ and x is a $(1, B)$-matching with $B \geq 2\ell$ such that there are no x-augmenting paths of length at most $4\lceil \log n \rceil + 1$ in the graph, then c is matched under x.*

Proof. Consider the graph G' constructed by removing all servers of load larger than ℓ from G and let x' be x restricted to G'. Since every augmenting path in G' is an augmenting path in G, there are no x'-augmenting paths of length $\leq 4\lceil \log n \rceil + 1$ in G'. Note that any optimal assignment in G restricted to G' has maximum load at most ℓ. Since x' is a $(1, B)$-matching, G' has a client-perfect $(1, \ell)$-matching, and $B \geq 2\ell$, it follows that x' is client-perfect in G' by Lemma 3. As x' is the restriction of x to G', it follows that x matches all clients of level at most ℓ in G. ◀

▶ **Theorem 11.** *In the* CONGEST *model, there is a randomized algorithm (Algorithm 1) that with high probability computes an 8-approximation to the all-norm load balancing problem in $O(\log^5 n)$ rounds.*

Proof. For each client c, let B_c be the smallest power of two that is at least $2\ell(c)$. By Lemma 10, each client c will be assigned in x_{B_c}. Fix a server s. Lemma 9 implies that the load of s is only determined by clients whose level is at most $\ell(s)$ as no other clients can be adjacent to s. Since each x_B contributes at most B to the load of s and only contributes for $B \leq 4\ell(s)$, we obtain that $L_A(s) \leq 1 + 2 + 4 + \cdots + 4\ell(s) \leq 8\ell(s)$. Thus,

$$\|A\|_p = \left(\sum_{s \in S} (L_A(s))^p \right)^{1/p} \leq \left(\sum_{s \in S} (8\ell(s))^p \right)^{1/p} = 8 \|A^*\|_p .$$ ◀

5 Weighted Load Balancing

In this section, we describe our algorithms for the weighted load balancing problem. We start by showing that with the simple reduction in [12] from unweighted to weighted load balancing, our unweighted algorithm (Algorithm 1) also implies an $O(\log n)$-approximate polylog(n)-round CONGEST algorithm for weighted instances. We then turn to the main result of

this section: an $O(1)$-approximate polylog(n)-round LOCAL algorithm for the weighted load balancing problem. We conclude this section with an $O(1)$-approximate sequential algorithm that runs in near-linear time.

As our goals in this section are to obtain, at best, an $O(1)$-approximation, we may assume that all client weights are powers of two. If not, rounding weights up to the nearest power of two will at most double the approximation ratio. We can assume similarly that the maximum weight $W \leq n$. Indeed, clients with load less than W/n can collectively distribute at most W weight across the servers and can therefore be assigned arbitrarily. Thus, when $W > n$, clients can simply rescale their own weight by n/W (and round it up to the nearest integer).

Throughout this section, we denote by C_i the set of clients whose weight is exactly 2^i. (By our previous assumption, the sets $\{C_i\}$ partition C.) We let $G_i := G[C_i, N(C_i)]$ be the induced graph on C_i and its neighborhood.

An $O(\log n)$-approximation in the CONGEST model. We begin with an easy corollary of our unweighted algorithm following a simple reduction in [12].

▶ **Theorem 12.** *In the* CONGEST *model, an $O(\log n)$-approximation to the all-norm weighted load balancing problem can be computed with high probability in $O(\log^5 n)$ rounds.*

Proof. Consider the following algorithm: For each weight class i, compute an assignment A_i of G_i using Algorithm 1 by treating all clients as having weight 1. Then, have each client in C_i assign itself according to A_i.

Since all A_i's can be computed in parallel (as the graphs G_i are edge-disjoint, only one of the parallel copies need to communicate over an edge), the algorithm runs in $O(\log^5 n)$ rounds. We now show that the resulting assignment A is $O(\log n)$-approximate for all norms.

Fix any $p \geq 1$ including $p = \infty$; let A^* be an assignment for G with minimum ℓ_p-norm, and let A_i^* be an assignment for G_i with minimum ℓ_p-norm. Clearly $\|A_i^*\|_p \leq \|A^*\|_p$ for all i. By Theorem 11, $\|A_i\|_p \leq 8\|A_i^*\|_p \leq 8\|A^*\|_p$. It follows that

$$\|A\|_p = \|A_1 + \cdots + A_{\log n}\|_p \leq \|A_1\|_p + \cdots + \|A_{\log n}\|_p \leq 8\log(n)\|A^*\|_p . \quad ◀$$

Preliminaries for the weighted algorithms. Though they use entirely different techniques, the LOCAL and sequential algorithms of the next two subsections both follow the same high-level approach: first compute a split assignment, then round it into an integral one.

▶ **Definition 13.** *Let $G = (C \cup S, E)$ be a bipartite graph with client weights $w : C \to \mathbb{Z}^+$. A* split assignment *y_f in G is a client-perfect (w, ∞)-matching (so servers have unbounded capacity). For every server s, the* load *$L_{y_f}(s)$ is the sum of edge-multiplicities incident to s.*

Notice that split assignments are a relaxation of standard assignments by allowing clients to be assigned to several different servers at once, contributing an integral load to each server, provided that the total load distributed by the client does not exceed its weight.

We will also need the following notion. Define the *client-expanded graph* $\widetilde{G}$ of G as the graph formed by making $w(c)$ copies of each client c. Formally, for each $c \in C$, the client-expanded graph has vertices $c_1, \ldots, c_{w(c)}$ and an edge between c_i and s for all $i \in [w(c)]$ if and only if G has an edge between c and s.

▶ **Observation 14.** *A split assignment y_f in G corresponds to an integral assignment in the client-expanded graph $\tilde{G}$ with the same server loads. Thus, since $\tilde{G}$ is unweighted, by Lemma 1 there exists an all-norm optimal split assignment y_f^*.*

5.1 An $O(1)$-approximation in the LOCAL model

Our main result in this section is the following theorem.

▶ **Theorem 15.** *In the* LOCAL *model, there is a randomized algorithm (Algorithm 2) that with high probability computes an $O(1)$-approximation to the weighted all-norm load balancing problem in $O(\log^3 n)$ rounds.*

We will need the next rounding lemma to describe our algorithm; the proof is standard.

▶ **Lemma 16.** *If $G = (C \cup S, E)$ contains a client-perfect (κ, τ)-matching x, then there exists an assignment $A : C \to S$ such that for all servers $s \in S$,*

$$L_A(s) \leq \tau(s) + \max_{c \in A^{-1}(s)} \kappa(c).$$

Proof of Lemma 16. Consider the set of edges F in the support of x. If $C \subseteq F$ is a cycle, we can alternately increase and decrease the value of $x(e)$ on each edge e of the cycle by $\min_{f \in C} x(f)$ to break the cycle without changing $x(\delta(v))$ for any $v \in V$ (this cycle can only be of even length as the input graph is bipartite). Thus, we may assume that the support of x has no cycles and thus is a forest.

We can next turn F into a collection of stars centered on servers. This done by rooting each tree T in the support of F arbitrarily, picking each server s which has a client parent-node c, and setting the edge $x(cs) = \kappa(c)$ and $x(cs') = 0$ for all other $s' \in N(c)$. This clearly satisfies the requirement of client c and the load on server s can only ever be increased by $\max_{c \in A^{-1}(s)} \kappa(c)$ as each server can only have one parent client. At this point, in F, any client is assigned to exactly one server and thus we obtain an integral solution in which the load of any server s is at most $\tau(s) + \max_{c \in A^{-1}(s)} \kappa(c)$, finalizing the proof. ◀

Our LOCAL algorithm consists of two main parts, an algorithm for solving the split load balancing problem and a rounding procedure, which we describe now in turn.

Computing a split assignment. The first step of the LOCAL algorithm is to compute an assignment $\tilde{A}$ in the client-expanded graph $\widetilde{G}$ of G using Algorithm 1. Note that in the LOCAL model, each client c can simulate all "new" clients $c_1, \ldots, c_{w(c)}$ in Algorithm 1 without any overhead in the round complexity.[1] As mentioned in Observation 14, the assignment $\tilde{A}$ corresponds to a split assignment with the same server loads. To limit the amount of notation in the algorithm description, we will sometimes refer to $\tilde{A}$ as a split assignment in G, although formally it is an assignment in $\tilde{G}$.

The guarantees of Algorithm 1 tell us that $\tilde{A}$ has small ℓ_p-norm. The next step is to use $\tilde{A}$ to find an integral assignment without much loss in the norm.

A "rounding" procedure. We would now ideally round the split assignment $\tilde{A}$ into an integral assignment, but even in the LOCAL model we cannot afford to run such a procedure directly. The fact that a good rounding exists, however, is enough for us to apply Lemma 3 to obtain a similarly good assignment, as we show below.

For each i, let $\tilde{A}_i$ be $\tilde{A}$ restricted to G_i. Lemma 16 states that there is a way to round $\tilde{A}_i$ into an assignment with load $\tau_i(s) = L^i_{\tilde{A}_i}(s) + 2^i$ for servers s assigned to by $\tilde{A}_i$ and $\tau_i(s) = 0$ for the remaining servers. Treating the clients as unweighted, $\tilde{A}_i$ corresponds to

[1] We remark that computing this assignment is the only step of our weighted algorithm that does not run efficiently in the CONGEST model, precisely because this simulation not possible in the CONGEST model in polylog(n) rounds.

Algorithm 2 Approximate weighted (all-norm) load balancing in the LOCAL model.

1 emulate Algorithm 1 on $\widetilde{G}$ to compute an assignment $\tilde{A}$
2 **for** $i \in \{1, 2, 4, \ldots, n\}$ *in parallel* **do**
3 let $\tilde{A}_i$ be $\tilde{A}$ restricted to $\widetilde{G}_i$
4 let $\tau_i(s) = \begin{cases} L_{\tilde{A}_i}(s) + 2^i, & \text{if } L_{\tilde{A}_i}(s) > 0 \\ 0, & \text{otherwise} \end{cases}$
▷ *by Lemma 16, an assignment with load vector point-wise less than τ_i exists in G_i*
▷ *therefore, scaling clients in G_i to weight 1, a $(1, \lceil 2^{-i}\tau_i \rceil)$-matching exists*
5 treating G_i as unweighted, compute a $(1, 2\lceil 2^{-i}\tau_i \rceil)$-matching x_i in G_i with no augmenting paths of length $4\lceil \log n \rceil + 1$
▷ *by Lemma 3, x_i is client-perfect*
6 let A_i be the assignment induced by x_i
7 assign each $c \in C_i$ to $A_i(c)$
8 **end**

a $(1, \lceil 2^{-i}\tau_i \rceil)$-matching. We now compute a $(1, 2\lceil 2^{-i}\tau_i \rceil)$-matching x_i with no augmenting paths of length $4\lceil \log n \rceil + 1$ or smaller. By Lemma 3, each x_i is client-perfect, inducing an (integral) assignment A_i in G_i. Lastly, each client in C_i assigns itself in accordance with A_i to produce the global assignment A. See Algorithm 2.

To formalize the logic of the algorithm, we make a few claims that together will imply the algorithm's correctness. The first claim ensures that the algorithm produces a proper assignment.

▷ Claim 17. Algorithm 2 assigns every client to some server.

Proof. We need to show that the matching x_i computed in Line 5 of Algorithm 2 is client-perfect. Consider τ_i from Line 4 of Algorithm 2. Viewing $\tilde{A}_i$ as a client-perfect $(w, L_{\tilde{A}_i})$-matching, Lemma 16 guarantees that there is an assignment wherein each server s has load at most $\tau_i(s)$.

Because all clients in G_i have the same weight, we can interpret the assignment from Lemma 16 as a client-perfect $(1, \lceil 2^{-i}\tau_i \rceil)$-matching in the unweighted graph G_i. When treating clients as unweighted, server capacities are always bounded by n, and so by Lemma 3, if x_i has no augmenting paths of length $\leq 4\lceil \log n \rceil + 1$, it follows that x_i is client-perfect. ◁

The next claim shows that the assignment produced is $O(1)$-approximate.

▷ Claim 18. There is a universal constant C such that for all $p \geq 1$, including $p = \infty$, the assignment A produced by Algorithm 2 satisfies $\|A\|_p \leq C \|A^*\|_p$, where A^* is an ℓ_p-norm-minimizing assignment.

Proof. Fix $p \geq 1$ (including $p = \infty$). Let A^* and $\tilde{A}^*$ be assignments for G and $\tilde{G}$, respectively, that minimize the ℓ_p-norm. For brevity, we omit the subscript p when writing norms with the understanding that all norms in this proof are ℓ_p-norms. We will also treat the client weight function w as a vector so that we can write its norm as $\|w\|$.

Our strategy is to decompose the final assignment A into two parts and bound the norms of those parts separately. First, we decompose each assignment A_i of Line 6. We define the first part, ρ_i, by $\rho_i(s) = 2^i$ if s is assigned to by A_i and $\rho_i(s) = 0$ otherwise. In other words, ρ_i has the same support as the load vector L_{A_i} of A_i, but all of its nonzero entries are 2^i. The second part, μ_i, docks 2^{i+1} from the support of L_{A_i}: $\mu_i = L_{A_i} - 2\rho_i$. Letting $\mu = \sum_i \mu_i$

and $\rho = \sum_i \rho_i$, we have that $\|A\| = \|\mu + 2\rho\| \le \|\mu\| + 2\|\rho\|$. It therefore suffices to show that $\|\mu\|$ and $\|\rho\|$ both $O(1)$-approximate $\|A^*\|$.

Let us first bound $\|\mu\|$. For any server s assigned to by A_i, we have

$$\begin{aligned}
\mu_i(s) &= L_{A_i}(s) - 2^{i+1} \\
&\le 2^{i+1}\lceil 2^{-i}\tau_i(s)\rceil - 2^{i+1} \\
&\le 2^{i+1}\lceil 2^{-i}L_{\tilde{A}_i}(s) + 1\rceil - 2^{i+1} \\
&\le 2^{i+1}2^{-i+1}L_{\tilde{A}_i}(s) + 2^{i+1} - 2^{i+1} && (\lceil x\rceil \le 2x \text{ for all } x \ge 1) \\
&\le 4L_{\tilde{A}_i}(s).
\end{aligned}$$

For any server s *not* assigned to by A_i we have $\mu_i(s) = 0$, and so trivially $\mu_i(s) \le 4L_{\tilde{A}_i(s)}$ for such s. Therefore, $\mu(s) = \sum_i \mu_i(s) \le \sum_i 4L_{\tilde{A}_i}(s) = 4L_{\tilde{A}}(s)$. Using Theorem 6, it follows that $\|\mu\| \le 4\|\tilde{A}\| \le 32\|\tilde{A}^*\| \le 32\|A^*\|$.

We now bound $\|\rho\|$. Define $\rho^*(s) = \max_{c \in A^{-1}(s)} w(c)$. Note that ρ^* is the load vector of a "partial" assignment (not all clients are assigned) that assigns to each server at most once. Since w can be interpreted as the load vector of an assignment that assigns *every* client to a unique server, we have $\|\rho^*\| \le \|w\|$. Now observe that $\rho(s) = \sum_{i=1}^{\log \rho^*(s)} \rho_i(s) \le 2\rho^*(s)$ simply because $\rho_i(s)$ is either 0 or 2^i for each i. To complete the bound, notice that $\|w\| \le \|A^*\|$; the best (hypothetical) assignment would assign every client to a unique server, resulting in value $\|w\|$. Putting things together, we have shown that $\|\rho\| \le 2\|A^*\|$. ◁

It remains to bound the round-complexity of the algorithm.

▷ Claim 19. Algorithm 2 takes $O(\log^3 n)$ rounds in the LOCAL model.

Proof. In the LOCAL model, we can easily emulate Algorithm 1 (or any algorithm) on the client-expansion $\tilde{G}$ at no extra cost; any communication across an edge cs simply needs to specify which c_i in the expansion the message is to/from. Since $W \le n$, Algorithm 1 still runs in $O(\log^3 n)$ rounds in the LOCAL model (see Remark 7). The main for-loop is run in parallel, and so we only need to bound the round-complexity of its body. Line 5 is the only line inside the loop that requires (additional) communication, and this again only takes $O(\log^3 n)$ rounds. The total round-complexity is therefore $O(\log^3 n)$. ◁

This concludes the proof of Theorem 15.

5.2 An $O(1)$-approximate $O(m \log^3 n)$-time sequential algorithm

We now show that our approach can also be used to compute an $O(1)$-approximation to the weighted all-norm load balancing problem in near-linear time in the standard sequential setting, proving the following theorem.

▶ **Theorem 20.** *In the standard sequential model, there is a deterministic algorithm to compute an $O(1)$-approximate solution to the weighted all-norm load balancing problem that runs in $O(m \log^3 n)$ time.*

Previously, Azar et al. [2] showed a 2-approximate algorithm for this problem, which runs in two phases: (1) compute an optimal fractional assignment and (2) round the fractional assignment, which incurs a 2-approximation. But their algorithm computes the optimal fractional assignment using the ellipsoid method to solve a linear program with exponentially many constraints, and hence incurs a large polynomial runtime.

Our algorithm uses the same rounding procedure as [2], but instead of computing an exact fractional assignment, we compute an $O(1)$-approximate split assignment in near-linear time by simulating our distributed approach in the sequential setting. To this end, we will need the following subroutine:

▶ **Lemma 21.** *Given any bipartite graph $G = (C \cup S, E)$ and capacity functions κ, τ, it is possible to compute a (κ, τ)-matching with no augmenting paths of length $\leq 8\log(n)$ in $O(m \log^2 n)$ time in the sequential setting.*

Proof. Note that a (κ, τ)-matching corresponds to the following flow problem. Every edge in E gets infinite capacity; there is a dummy source v_s and for every client $c \in C$ there is an edge (v_s, c) of capacity $\kappa(c)$; there is also a dummy sink v_t and for every server $s \in S$ there is an edge from s to v_t of capacity $\tau(s)$. It is immediate to verify that any v_s-v_t flow in this network corresponds to a (κ, τ)-matching and vice versa.

We now show how to compute a solution to this flow problem that contains no augmenting paths of length $9\log(n) \geq 8\log(n) + 2$ which corresponds to the desired (κ, τ)-matching.

The algorithm simply runs $9\log(n)$ successive iterations of blocking flow. A blocking flow in a capacitated graph can be computed in $O(m \log n)$ time using the dynamic tree structure of Sleator and Tarjan [24]. ◀

We are now ready to show our algorithm to compute a split assignment.

▶ **Lemma 22.** *Let $G = (C \cup S, E)$ be a bipartite graph with client-weights $w(C)$. There exists a sequential algorithm that in $O(m \log^3 n)$ time computes a split assignment y_f such that $\left\|L_{y_f}\right\|_p \leq 8 \left\|L_{y_f^*}\right\|_p$ for every $p \geq 1$ (including $p = \infty$).*

Proof. Recall from Observation 14 that the optimal split assignment y_f^* corresponds to an optimal (integral) assignment $\tilde{A}^*$ in the client-expanded graph $\tilde{G}$; server loads in the two solutions are the same, so $\left\|L_{y_f^*}\right\|_p = \|L_{\tilde{A}^*}\|_p$. We obtain our split assignment y_f by simulating Algorithm 1 on the graph $\tilde{G}$: by Theorem 11, this yields the desired 8-approximation. We now describe how to execute the simulation in the sequential model and how to convert between the perspectives of split assignment in G and integral assignment in $\tilde{G}$.

Firstly, in Line 2 of Algorithm 1, we need to a compute a $(1, B)$-matching in $\tilde{G}$ with no short augmenting paths. This is equivalent to a (w, B)-matching in G, which we compute in $O(m \log^2 n)$ time using Lemma 21.

Secondly, in Line 4 of Algorithm 1, each client-copy $\tilde{c}$ in $\tilde{G}$ must find the minimum B such that $\tilde{c}$ is matched in x_B. We need to convert this line to the language of split assignments. In particular, note that in our sequential simulation, x_B is a (w, B)-matching in G rather than a $(1, B)$-matching in $\tilde{G}$. It is easy to see that the following simulates Line 4. For each client c in G, let $s_B(c)$ be the set of servers incident to c in x_B: if an edge cs has multiplicity α in x_B, then s appears α times in $s_B(c)$. To construct the split assignment y_f, first assign c to the server in $s_1(c)$ (if any). Then assign c to an arbitrary $|s_2(c)| - |s_1(c)|$ servers from $s_2(c)$, an arbitrary $|s_4(c)| - |s_2(c)|$ servers from $s_4(c)$, and more generally an arbitrary $|s_B(c)| - |s_{B/2}(c)|$ servers from $s_B(c)$. It is not hard to check that the resulting split assignment is equivalent to some integral assignment in $\tilde{G}$ formed by executing Line 4 of Algorithm 1 in $\tilde{G}$. It is also easy to see that for each x_B the assignments can be performed in $O(m)$ time, for a total of $O(m \log n)$ time.

The running time of the algorithm is thus dominated by the time for computing matchings x_B. Each takes $O(m \log^2 n)$ time to compute (Lemma 21), and there are $O(\log(nW)) = O(\log n)$ values of B, so the total run-time is $O(m \log^3 n)$. ◀

Finally, we round the split assignment to an integral assignment using the rounding procedure of [2], which is described in the proof of Lemma 16. The rounding procedure has two steps: cycle cancelling and computing a matching in a tree. The second can clearly be done in $O(m)$ sequential time. Cycle cancelling can be done deterministically in $O(m \log n)$ time (see, e.g., [15]). The total time for rounding is thus $O(m \log n)$. (Note that in the distributed setting we only relied on the *existence* of such a rounding procedure, because it is unclear how to implement cycle canceling efficiently in the LOCAL model.)

Following the exact same argument as in [2] or in the proof of Lemma 15 of this paper, since our split assignment was an 8-approximation (Theorem 22), the integral assignment formed by rounding yields a 9-approximation. This concludes the proof of Theorem 20.

References

1. Sepehr Assadi, Aaron Bernstein, and Zachary Langley. Improved bounds for distributed load balancing, 2020. arXiv:2008.04148. URL: `https://arxiv.org/abs/2008.04148`.
2. Yossi Azar, Leah Epstein, Yossi Richter, and Gerhard J. Woeginger. All-norm approximation algorithms. *J. Algorithms*, 52(2):120–133, 2004. `doi:10.1016/j.jalgor.2004.02.003`.
3. Leonid Barenboim and Gal Oren. Distributed backup placement in one round and its applications to maximum matching and self-stabilization. In *Proc. 3rd Symposium on Simplicity in Algorithms (SOSA 2020)*, pages 99–105, 2020. `doi:10.1137/1.9781611976014.14`.
4. Aaron Bernstein, Jacob Holm, and Eva Rotenberg. Online bipartite matching with amortized $O(\log^2 n)$ replacements. *J. ACM*, 66(5):Art. 37, 23, 2019. `doi:10.1145/3344999`.
5. Aaron Bernstein, Tsvi Kopelowitz, Seth Pettie, Ely Porat, and Clifford Stein. Simultaneously load balancing for every p-norm, with reassignments. In *Proc. 8th Innovations in Theoretical Computer Science Conference (ITCS 2017)*, volume 67 of *LIPIcs. Leibniz Int. Proc. Inform.*, pages Art. No. 51, 14. Schloss Dagstuhl. Leibniz-Zent. Inform., Wadern, 2017. `doi:10.4230/LIPIcs.ITCS.2017.51`.
6. J. Bruno, E. G. Coffman, Jr., and R. Sethi. Scheduling independent tasks to reduce mean finishing time. *Comm. ACM*, 17:382–387, 1974. `doi:10.1145/361011.361064`.
7. Deeparnab Chakrabarty and Chaitanya Swamy. Simpler and better algorithms for minimum-norm load balancing. In *Proc. 27th Annual European Symposium on Algorithms (ESA 2019)*, volume 144 of *LIPIcs. Leibniz Int. Proc. Inform.*, pages Art. No. 27, 12. Schloss Dagstuhl. Leibniz-Zent. Inform., Wadern, 2019. `doi:10.4230/LIPIcs.ESA.2019.27`.
8. Andrzej Czygrinow, Michal Hanćkowiak, Edyta Szymańska, and Wojciech Wawrzyniak. Distributed 2-approximation algorithm for the semi-matching problem. In *Proc. 26th International Symposium on Distributed Computing (DISC 2012)*, volume 7611 of *LNCS*, pages 210–222. Springer, Heidelberg, 2012. `doi:10.1007/978-3-642-33651-5_15`.
9. Jittat Fakcharoenphol, Bundit Laekhanukit, and Danupon Nanongkai. Faster algorithms for semi-matching problems. *ACM Trans. Algorithms*, 10(3):Art. 14, 23, 2014. `doi:10.1145/2601071`.
10. Martin Gairing, Thomas Lücking, Marios Mavronicolas, and Burkhard Monien. The price of anarchy for restricted parallel links. *Parallel Process. Lett.*, 16(1):117–131, 2006. `doi:10.1142/S0129626406002514`.
11. Anupam Gupta, Amit Kumar, and Cliff Stein. Maintaining assignments online: matching, scheduling, and flows. In *Proc. 25th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2014)*, pages 468–479. ACM, 2014. `doi:10.1137/1.9781611973402.35`.
12. Magnús M. Halldórsson, Sven Köhler, Boaz Patt-Shamir, and Dror Rawitz. Distributed backup placement in networks. *Distrib. Comput.*, 31(2):83–98, 2018. `doi:10.1007/s00446-017-0299-x`.
13. Nicholas J. A. Harvey, Richard E. Ladner, László Lovász, and Tami Tamir. Semi-matchings for bipartite graphs and load balancing. *J. Algorithms*, 59(1):53–78, 2006. `doi:10.1016/j.jalgor.2005.01.003`.

14 W. A. Horn. Minimizing average flow time with parallel machines. *Oper. Res.*, 21(3), 1973. doi:10.1287/opre.21.3.846.

15 Donggu Kang and James Payor. Flow rounding, 2015. arXiv:1507.08139. URL: https://arxiv.org/abs/1507.08139.

16 Sven Köhler, Volker Turau, and Gerhard Mentges. Self-stabilizing local k-placement of replicas with minimal variance. In *Proc. 14th Stabilization, Safety, and Security of Distributed Systems (SSS 2012)*, pages 16–30, 2012. doi:10.1016/j.tcs.2015.04.019.

17 Christian Konrad and Adi Rosén. Approximating semi-matchings in streaming and in two-party communication. *ACM Trans. Algorithms*, 12(3):Art. 32, 21, 2016. doi:10.1145/2898960.

18 Anshul Kothari, Subhash Suri, Csaba D. Tóth, and Yunhong Zhou. Congestion games, load balancing, and price of anarchy. In *Proc. 1st Combinatorial and Algorithmic Aspects of Networking*, volume 3405 of *LNCS*, pages 13–27. Springer, Berlin, 2005. doi:10.1007/11527954_3.

19 Yixun Lin and Wenhua Li. Parallel machine scheduling of machine-dependent jobs with unit-length. *European J. Oper. Res.*, 156(1):261–266, 2004. doi:10.1016/S0377-2217(02)00914-1.

20 Zvi Lotker, Boaz Patt-Shamir, and Seth Pettie. Improved distributed approximate matching. *J. ACM*, 62(5):Art. 38, 17, 2015. doi:10.1145/2786753.

21 Renita Machado and Sirin Tekinay. A survey of game-theoretic approaches in wireless sensor networks. *Comput. Networks*, 52(16):3047–3061, 2008. doi:10.1016/j.gaceta.2008.07.003.

22 Gal Oren, Leonid Barenboim, and Harel Levin. Distributed fault-tolerant backup-placement in overloaded wireless sensor networks. In *Proc. 9th International Conference on Broadband Communications, Networks, and Systems (BROADNETS 2018)*, pages 212–224, 2018. doi:10.1007/978-3-030-05195-2_21.

23 Narayanan Sadagopan, Mitali Singh, and Bhaskar Krishnamachari. Decentralized utility-based sensor network design. *Mobile Networks and Applications*, 11(3):341–350, 2006. doi:10.1007/s11036-006-5187-8.

24 Daniel D. Sleator and Robert E. Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, 1983. doi:10.1016/0022-0000(83)90006-5.

25 Subhash Suri, Csaba D. Tóth, and Yunhong Zhou. Uncoordinated load balancing and congestion games in P2P systems. In *Proc. 3rd International Workshop on Peer-to-Peer Systems (IPTPS 2004)*, pages 123–130, 2004. doi:10.1007/978-3-540-30183-7_12.

26 Subhash Suri, Csaba D. Tóth, and Yunhong Zhou. Selfish load balancing and atomic congestion games. *Algorithmica*, 47(1):79–96, 2007. doi:10.1007/s00453-006-1211-4.

Intermediate Value Linearizability: A Quantitative Correctness Criterion

Arik Rinberg
Technion – Israel Institute of Technology, Haifa, Israel
ArikRinerg@campus.technion.ac.il

Idit Keidar
Technion – Israel Institute of Technology, Haifa, Israel
idish@ee.technion.ac.il

Abstract

Big data processing systems often employ batched updates and data sketches to estimate certain properties of large data. For example, a *CountMin sketch* approximates the frequencies at which elements occur in a data stream, and a *batched counter* counts events in batches. This paper focuses on correctness criteria for concurrent implementations of such objects. Specifically, we consider *quantitative* objects, whose return values are from a totally ordered domain, with a particular emphasis on (ϵ, δ)*-bounded* objects that estimate a numerical quantity with an error of at most ϵ with probability at least $1 - \delta$.

The de facto correctness criterion for concurrent objects is linearizability. Intuitively, under linearizability, when a read overlaps an update, it must return the object's value either before the update or after it. Consider, for example, a single batched increment operation that counts three new events, bumping a batched counter's value from 7 to 10. In a linearizable implementation of the counter, a read overlapping this update must return either 7 or 10. We observe, however, that in typical use cases, any *intermediate* value between 7 and 10 would also be acceptable. To capture this additional degree of freedom, we propose *Intermediate Value Linearizability (IVL)*, a new correctness criterion that relaxes linearizability to allow returning intermediate values, for instance 8 in the example above. Roughly speaking, IVL allows reads to return any value that is bounded between two return values that are legal under linearizability. A key feature of IVL is that we can prove that concurrent IVL implementations of (ϵ, δ)-bounded objects are themselves (ϵ, δ)-bounded. To illustrate the power of this result, we give a straightforward and efficient concurrent implementation of an (ϵ, δ)-bounded CountMin sketch, which is IVL (albeit not linearizable).

Finally, we show that IVL allows for inherently cheaper implementations than linearizable ones. In particular, we show a lower bound of $\Omega(n)$ on the step complexity of the update operation of any wait-free linearizable batched counter from single-writer objects, and propose a wait-free IVL implementation of the same object with an $O(1)$ step complexity for update.

2012 ACM Subject Classification Theory of computation → Semantics and reasoning; Computer systems organization → Parallel architectures; Computing methodologies → Concurrent computing methodologies

Keywords and phrases concurrency, concurrent objects, linearizability

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.2

Related Version A full version of the paper is available at [31], `https://arxiv.org/abs/2006.12889`.

Acknowledgements We thank Hagit Attiya and Gadi Taubenfeld for their insightful comments, and the anonymous reviewers for their detailed feedback.

© Arik Rinberg and Idit Keidar;
licensed under Creative Commons License CC-BY
34th International Symposium on Distributed Computing (DISC 2020).
Editor: Hagit Attiya; Article No. 2; pp. 2:1–2:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

1.1 Motivation

Big data processing systems often perform analytics on incoming data streams, and must do so at a high rate due to the speed of incoming data. Data sketching algorithms, or *sketches* for short [6], are an indispensable tool for such high-speed computations. Sketches typically estimate some function of a large stream, for example, the frequency of certain items [7], how many unique items have appeared [9, 13, 14], or the top-k most common items [26]. They are supported by many data analytics platforms such as PowerDrill [18], Druid [11], Hillview [19], and Presto [30] as well as standalone toolkits [10].

Sketches are quantitative objects that support UPDATE and QUERY operations, where the return value of a QUERY is from a totally ordered set. They are essentially succinct (sublinear) summaries of a data stream. For example, a sketch might estimate the number of packets originating from any IP address, without storing a record for every packet. Typical sketches are *probably approximately correct (PAC)*, estimating some aggregate quantity with an error of at most ϵ with probability at least $1 - \delta$ for some parameters ϵ and δ. We say that such sketches are (ϵ, δ)*-bounded*.

The ever increasing rates of incoming data create a strong demand for parallel stream processing [8, 18]. In order to allow queries to return fresh results in real-time without hampering data ingestion, it is paramount to support queries concurrently with updates [32, 33]. But parallelizing sketches raises some important questions, for instance: *What are the semantics of overlapping operations in a concurrent sketch?*, *How can we prove error guarantees for such a sketch?*, and, in particular, *Can we reuse the myriad of clever analyses of existing sketches' error bounds in parallel settings without opening the black box?* In this paper we address these questions.

1.2 Our contributions

The most common correctness condition for concurrent objects is linearizability. Roughly speaking, it requires each parallel execution to have a *linearization*, which is a sequential execution of the object that "looks like" the parallel one. (See Section 2 for a formal definition.) But sometimes linearizability is too restrictive, leading to a high implementation cost.

In Section 3, we propose *Intermediate Value Linearizability (IVL)*, a new correctness criterion for quantitative objects. Intuitively, the return value of an operation of an IVL object is bounded between two legal values that can be returned in linearizations. The motivation for allowing this is that if the system designer is happy with either of the legal values, then the intermediate value should also be fine. For example, consider a system where processes count events, and a monitoring process detects when the number of events passes a threshold. The monitor constantly reads a shared counter, which other process increment in batches. If an operation increments the counter from 4 to 7 batching three events, IVL allows a concurrent read by the monitoring process to return 6, although there is no linearization in which the counter holds 6. We formally define IVL and prove that this property is *local*, meaning that a history composed of IVL objects is itself IVL. This allows reasoning about single objects rather than about the system as a whole. We formulate IVL first for sequential objects, and then extend it to capture randomized ones.

Next, we consider (ϵ, δ)-bounded algorithms like data sketches. Existing (sequential) algorithms have sequential error analyses which we wish to leverage for the concurrent case. In Section 4 we formally define (ϵ, δ)-bounded objects, including concurrent ones. We then prove

a key theorem about IVL, stating that an IVL implementation of a sequential (ϵ, δ)-bounded object is itself (ϵ, δ)-bounded. The importance of this theorem is that it provides a generic way to leverage the vast literature on sequential (ϵ, δ)-bounded sketches [27, 12, 5, 24, 7, 1] in concurrent implementations.

As an example, in Section 5, we present a concurrent CountMin sketch [7], which estimates the frequencies of items in a data stream. We prove that a straightforward parallelization of this sketch is IVL. By the aforementioned theorem, we deduce that the concurrent sketch adheres to the error guarantees of the original sequential one, without having to "open" the analysis. We note that this parallelization is *not* linearizable.

Finally, we show that IVL is sometimes inherently cheaper than linearizability. We illustrate this in Section 6 via the example of a *batched counter*. We present a wait-free IVL implementation of this object from single-writer-multi-reader (SWMR) registers with $O(1)$ step complexity for UPDATE operations. We then prove a lower bound of $\Omega(n)$ step complexity for the UPDATE operation of any wait-free linearizable implementation, using only SWMR registers. This exemplifies that there is an inherent and unavoidable cost when implementing linearizable algorithms, which can be circumvented by implementing IVL algorithms instead.

2 Preliminaries

Section 2.1 discusses deterministic shared memory objects and defines linearizability. In Section 2.2 we discuss randomized algorithms and their correctness criteria.

2.1 Deterministic objects

We consider a standard shared memory model [17], where a set of *processes* access atomic shared memory variables. Accessing these shared variables is instantaneous. Processes take *steps* according to an *algorithm*, which is a deterministic state machine, where a step can access a shared memory variable, do local computations, and possibly return some value. An *execution* of an algorithm is an alternating sequence of steps and states. We focus on algorithms that implement *objects*, which support *operations*, such as READ and WRITE. Operations begin with an *invocation* step and end with a *response* step. A *schedule*, denoted σ, is the order in which processes take steps, and the operations they invoke in invoke steps with their parameters. Because we consider deterministic algorithms, σ uniquely defines an execution of a given algorithm.

A *history* is the sequence of invoke and response steps in an execution. Given an algorithm A and a schedule σ, $H(A, \sigma)$ is the history of the execution of A with schedule σ. A *sequential* history is an alternating sequence of invocations and their responses, beginning with an invoke step. We denote the return value of operation op with parameter arg in history H by $\text{ret}(op, H)$. We refer to the invocation step of operation op with parameter arg by process p as $inv_p(op(arg))$ and to its response step by $rsp_p(op) \rightarrow ret$, where $ret = \text{ret}(op, H)$. A history defines a partial order on operations: Operation op_1 *precedes* op_2 in history H, denoted $op_1 \prec_H op_2$, if $rsp(op_1)$ precedes $inv(op_2(arg))$ in H. Two operations are *concurrent* if neither precedes the other.

A *well-formed* history is one that does not contain concurrent operations by the same process, and where every response event for operation op is preceded by an invocation of the same operation. A schedule is well-formed if it gives rise to a well-formed history, and an execution is well-formed if it is based on a well-formed schedule. We denote by $H|_x$ the sub-history of H consisting only of invocations and responses on object x. Operation op is *pending* in a history H if op is invoked in H but does not return.

Correctness of an object's implementation is defined with respect to a sequential specification $\mathcal{H}$, which is the object's set of allowed sequential histories. If the history spans multiple objects, $\mathcal{H}$ consists of sequential histories H such that for all objects x, $H|_x$ pertains to x's sequential specification (denoted $\mathcal{H}_x$). A *linearization* [17] of a concurrent history H is a sequential history H' such that (1) after removing some pending operations from H and completing others, it contains the same invocations and responses as H' with the same parameters and return values, and (2) H' preserves the partial order $\prec_H$. Note that our definition of linearization diverges from the one in [17] in that it is not associated with any sequential specification; instead we require that the linearization pertain to the sequential specification when defining linearizability as follows: Algorithm A is a *linearizable implementation* of a sequential specification $\mathcal{H}$ if every history of a well-formed execution of A has a linearization in $\mathcal{H}$.

2.2 Randomized algorithms

In randomized algorithms, processes have access to coin flips from some domain Ω. Every execution is associated with a coin flip vector $\vec{c} = (c_1, c_2, \dots)$, where $c_i \in \Omega$ is the ith coin flip in the execution. A *randomized algorithm* A is a probability distribution over deterministic algorithms $\{A(\vec{c})\}_{\vec{c}\in\Omega^\infty}$[1], arising when A is instantiated with different coin flip vectors. We denote by $H(A, \vec{c}, \sigma)$ the history of the execution of randomized algorithm A observing coin flip vector $\vec{c}$ in schedule σ.

Golab et al. show that randomized algorithms that use concurrent objects require a stronger correctness criterion than linearizability, and propose *strong linearizability* [15]. Roughly speaking, strong linearizability stipulates that the mapping of histories to linearizations must be prefix-preserving, so that future coin flips cannot impact the linearization order of earlier events. In contrast to us, they consider deterministic objects used by randomized algorithms. In this paper, we focus on randomized object implementations.

3 Intermediate value linearizability

Section 3.1 introduces definitions that we utilize to define IVL. Section 3.2 defines IVL for deterministic algorithms and proves that it is a local property. Section 3.3 extends IVL for randomized algorithms, and Section 3.4 compares IVL to other correctness criteria.

3.1 Definitions

Throughout this paper we consider the strongest progress guarantee, bounded wait-freedom. An operation *op* is *bounded wait-free* if whenever any process p invokes *op*, *op* returns a response in a bounded number of p's steps, regardless of steps taken by other processes. An operation's *step-complexity* is the maximum number of steps a process takes during a single execution of this operation. We can convert every bounded wait-free algorithm to a *uniform step complexity* one, in which each operation takes the exact same number of steps in every execution. This can be achieved by padding shorter execution paths with empty steps before returning. Note that in a randomized algorithm with uniform step complexity, coin flips have no impact on *op*'s execution times. For the remainder of this paper, we consider algorithms with uniform step complexity.

---

[1] We do not consider non-deterministic objects in this paper.

Our definitions use the notion of skeleton histories: A *skeleton history* is a sequence of invocation and response events, where the return values of the responses are undefined, denoted ?. For a history H, we define the operator $H^?$ as altering all response values in H to ?, resulting in a skeleton history.

In this paper we formulate correctness criteria for a class of objects we call *quantitative*. These are objects that support two operations: (1) UPDATE, which captures all mutating operations and does not return a value; and (2) QUERY, which returns a value from a totally ordered domain. In a *deterministic quantitative object* the return values of QUERY operations are uniquely defined. Namely, the object's sequential specification $\mathcal{H}$ contains exactly one history for every sequential history skeleton H; we denote this history by $\tau_{\mathcal{H}}(H)$. Thus, $\tau_{\mathcal{H}}(H^?) = H$ for every history $H \in \mathcal{H}$. Furthermore, for every sequential skeleton history H, by definition, $\tau_{\mathcal{H}}(H) \in \mathcal{H}$.

▶ **Example 1.** Consider an execution in which a batched counter (formally defined in Section 6.2) initialized to 0 is incremented by 3 by process p concurrently with a query by process q, which returns 0. Its history is:

$$H = inv_p(inc(3)), inv_q(query), rsp_p(inc), rsp_q(query \to 0).$$

The skeleton history $H^?$ is:

$$H^? = inv_p(inc(3)), inv_q(query), rsp_p(inc), rsp_q(query \to ?).$$

A possible linearization of $H^?$ is:

$$H' = inv_p(inc(3)), rsp_p(inc), inv_q(query), rsp_q(query \to ?).$$

Given the sequential specification $\mathcal{H}$ of a batched counter, we get:

$$\tau_{\mathcal{H}}(H') = inv_p(inc(3)), rsp_p(inc), inv_q(query), rsp_q(query \to 3).$$

In a different linearization, the query may return 0 instead.

3.2 Intermediate value linearizability

We now define intermediate value linearizability for quantitative objects.

▶ **Definition 2** (Intermediate value linearizability)**.** *A history H of an object is IVL with respect to sequential specification $\mathcal{H}$ if there exist two linearizations H_1, H_2 of $H^?$ such that for every* QUERY *Q that returns in H,*

$$ret(Q, \tau_{\mathcal{H}}(H_1)) \le ret(Q, H) \le ret(Q, \tau_{\mathcal{H}}(H_2)).$$

Algorithm A is an IVL implementation *of a sequential specification $\mathcal{H}$ if every history of a well-formed execution of A is IVL with respect to $\mathcal{H}$.*

Note that a linearizable object is trivially IVL, as the skeleton history of the linearization of H plays the roles of both H_1 and H_2. We emphasize that in a sequential execution, an IVL object is not relaxed in any way – it must follow the sequential specification. Similarly, in a well-formed history, operations of the same process never overlap, and so IVL executions satisfy program order. The following theorem, proven in the full paper [31], shows that this property is local (as defined in [17]):

▶ **Theorem 1.** *A history H of a well-formed execution of algorithm A over a set of objects $\mathcal{X}$ is IVL if and only if for each object $x \in \mathcal{X}$, $H|_x$ is IVL.*

Locality allows system designers to reason about their system in a modular fashion. Each object can be built separately, and the system as a whole still satisfies the property.

3.3 Extending IVL for randomized algorithms

In a randomized algorithm A with uniform step complexity, every invocation of a given operation returns after the same number of steps, regardless of the coin flip vector $\vec{c}$. This, in turn, implies that for a given σ, for any $\vec{c}, \vec{c}' \in \Omega^\infty$, the arising histories $H(A, \vec{c}, \sigma)$ and $H(A, \vec{c}', \sigma)$ differ only in the operations' return values but not in the order of invocations and responses, as the latter is determined by σ, so their skeletons are equal. For randomized algorithm A and schedule σ, we denote this arising skeleton history by $H^?(A, \sigma)$.

We are faced with a dilemma when defining the specification of a randomized algorithm A, as the algorithm itself is a distribution over a set of algorithms $\{A(\vec{c})\}_{\vec{c} \in \Omega^\infty}$. Without knowing the observed coin flip vector $\vec{c}$, the execution behaves unpredictably. We therefore define a deterministic sequential specification $\mathcal{H}(\vec{c})$ for each coin flip vector $\vec{c} \in \Omega^\infty$, so the sequential specification is a probability distribution on a set of sequential histories $\{\mathcal{H}(\vec{c})\}_{\vec{c} \in \Omega^\infty}$.

A correctness criterion for randomized objects needs to capture the property that the distribution of a randomized algorithm's outcomes matches the distribution of behaviors allowed by the specification. Consider, e.g., some sequential skeleton history H of an object defined by $\{\mathcal{H}(\vec{c})\}_{\vec{c} \in \Omega^\infty}$. Let Q be a query that returns in H, and assume that Q has some probability p to return a value v in $\tau_{\mathcal{H}(\vec{c})}(H)$ for a randomly sampled $\vec{c}$. Intuitively, we would expect that if a randomized algorithm A "implements" the specification $\{\mathcal{H}(\vec{c})\}_{\vec{c} \in \Omega^\infty}$, then Q has a similar probability to return v in sequential executions of A with the same history, and to some extent also in concurrent executions of A of which H is a linearization. In other words, we would like the distribution of outcomes of A to match the distribution of outcomes in $\{\mathcal{H}(\vec{c})\}_{\vec{c} \in \Omega^\infty}$.

We observe that in order to achieve this, it does not suffice to require that each history have an arbitrary linearization as we did for deterministic objects, because this might not preserve the desired distribution. Instead, for randomized objects we require a common linearization for each skeleton history that will hold true under all possible coin flip vectors. We therefore formally define IVL for randomized objects as follows:

▶ **Definition 3** (IVL for randomized algorithms). *Consider a skeleton history $H = H^?(A, \sigma)$ of some randomized algorithm A with schedule σ. H is* IVL *with respect to $\{\mathcal{H}(\vec{c})\}_{\vec{c} \in \Omega^\infty}$ if there exist linearizations H_1, H_2 of H such that for every coin flip vector $\vec{c}$ and query Q that returns in H,*

$$ret(Q, \tau_{\mathcal{H}(\vec{c})}(H_1)) \leq ret(Q, H(A, \vec{c}, \sigma)) \leq ret(Q, \tau_{\mathcal{H}(\vec{c})}(H_2)).$$

Algorithm A is an IVL implementation *of a sequential specification distribution $\{\mathcal{H}(\vec{c})\}_{\vec{c} \in \Omega^\infty}$ if every skeleton history of a well-formed execution of A is IVL with respect to $\{\mathcal{H}(\vec{c})\}_{\vec{c} \in \Omega^\infty}$.*

Note that the query Q in Definition 3 is not necessarily "correct"; rather, it is bounded between two (possibly erroneous) results returned in linearizations. Since we require a common linearization under all coin flip vectors, we do not need to strengthen IVL for randomized settings in the manner that strong linearizability strengthens linearizability. This is because the linearizations we consider are a fortiori independent of future coin flips.

3.4 Relationship to other relaxations

In spirit, IVL resembles the *regularity* correctness condition for single-writer registers [22], where a query must return either a value written by a concurrent write or the last value written by a write that completed before the query began. Stylianopoulos et al. [33] adopt a

similar condition for data sketches, which they informally describe as follows: "a query takes into account all completed insert operations and possibly a subset of the overlapping ones." If the object's estimated quantity (return value) is monotonically increasing throughout every execution, then IVL essentially formalizes this condition, while also allowing intermediate steps of a single update to be observed. But this is not the case in general. Consider, for example, an object supporting increment and decrement, and a query that occurs concurrently with an increment and an ensuing decrement. If the query takes only the decrement into account (and not the increment), it returns a value that is smaller than all legal return values that may be returned in linearizations, which violates IVL. Our interval-based formalization is instrumental to ensuring that a concurrent IVL implementation preserves the probabilistic error bounds of the respective sequential sketch, which we prove in the next section.

Another example of an object specified in the spirit of IVL is Lamport's monotonic clock [23], where a read is required to return a value bounded between the clock's values at the beginning and end of the read's interval.

Previous work on set-linearizability [28] and interval-linearizability [4] has also relaxed linearizability, allowing a larger set of return values in the presence of overlapping operations. The set of possible return values, however, must be specified in advance by a given state machine; operations' effects on one another must be predefined. In fact, interval-linearizability could be used to define IVL on a per-object basis, by defining a nondeterministic interval-sequential object in which a read operation can return any value in the interval defined by the update operations that are concurrent with it. In contrast, IVL is generic and does not require additional object-specific definitions; it provides an intuitive quantitative bound on possible return values of arbitrary quantitative objects.

Henzinger et al. [16] define the quantitative relaxation framework, which allows executions to differ from the sequential specification up to a bounded cost function. Alistarh et al. expand upon this and define *distributional linearizability* [2], which requires a distribution over the internal states of the object for its error analysis. Rinberg et al. consider strongly linearizable r-relaxed semantics for randomized objects [32]. We differ from these works in two points: First, a sequential history of an IVL object must adhere to the sequential specification, whereas in these relaxations even a sequential history may diverge from the specification. The second is that these relaxations are measured with respect to a single linearization. We, instead, bound the return value between two legal linearizations. The latter is the key to preserving the error bounds of sequential objects, as we next show.

4 (ϵ, δ)-bounded objects

In this section we show that for a large class of randomized objects, IVL concurrent implementations preserve the error bounds of the respective sequential ones. More specifically, we focus on randomized objects like data sketches, which estimate some quantity (or quantities) with probabilistic guarantees. Sketches generally support two operations: UPDATE(a), which processes element a, and QUERY(arg), which returns the quantity estimated by the sketch as a function of the previously processed elements. Sequential sketch algorithms typically have probabilistic error bounds. For example, the Quantiles sketch estimates the rank of a given element in a stream within $\pm\epsilon n$ of the true rank, with probability at least $1 - \delta$ [1].

We consider in this section a general class of (ϵ, δ)-*bounded objects* capturing PAC algorithms. A bounded object's behavior is defined relative to a deterministic sequential specification $\mathcal{I}$, which uniquely defines the *ideal* return value for every query in a sequential execution. In an (ϵ, δ)-bounded $\mathcal{I}$ object, each query returns the ideal return value within an

error of at most ϵ with probability at least $1-\delta$. More specifically, it over-estimates (and similarly under-estimates) the ideal quantity by at most ϵ with probability at least $1-\frac{\delta}{2}$. Formally:

▶ **Definition 4.** *A sequential randomized algorithm A implements an (ϵ,δ)-bounded $\mathcal{I}$ object if for every query Q returning in an execution of A with any schedule σ and a randomly sampled coin flip vector $\vec{c} \in \Omega^\infty$,*

$$ret(Q, H(A,\sigma,\vec{c})) \geq ret(Q, \tau_{\mathcal{I}}(H^?(A,\sigma)) - \epsilon \textit{ with probability at least } 1-\frac{\delta}{2},$$

and

$$ret(Q, H(A,\sigma,\vec{c})) \leq ret(Q, \tau_{\mathcal{I}}(H^?(A,\sigma)) + \epsilon \textit{ with probability at least } 1-\frac{\delta}{2}.$$

A induces a sequential specification $\{A(\vec{c})\}_{\vec{c}\in\Omega^\infty}$ of an (ϵ,δ)-bounded $\mathcal{I}$ object.

We next discuss parallel implementations of this specification.

To this end, we must specify a correctness criterion on the object's concurrent executions. As previously stated, the standard notion is (strong) linearizability, stipulating that we can "collapse" each operation in the concurrent schedule to a single point in time. Intuitively, this means that every query returns a value that could have been returned by the randomized algorithm at some point during its execution interval. So the query returns an (ϵ,δ) approximation of the ideal value at that particular point. But this point is arbitrarily chosen, meaning that the query may return an ϵ approximation of any value that the ideal object takes during the query's execution. We therefore look at the minimum and maximum values that the ideal object may take during a query's interval, and bound the error relative to these values.

We first define these minimum and maximum values as follows: For a history H, denote by $\mathcal{L}(H^?)$ the set of linearizations of $H^?$. For a query Q that returns in H and an ideal specification $\mathcal{I}$, we define:

$$v^{\mathcal{I}}_{min}(H,Q) \triangleq \min\{\text{ret}(Q, \tau_{\mathcal{I}}(L) \mid L \in \mathcal{L}(H^?)\});$$
$$v^{\mathcal{I}}_{max}(H,Q) \triangleq \max\{\text{ret}(Q, \tau_{\mathcal{I}}(L) \mid L \in \mathcal{L}(H^?))\}.$$

Note that even if H is infinite and has infinitely many linearizations, because Q returns in H, it appears in each linearization by the end of its execution interval, and therefore Q can return a finite number of different values in these linearizations, and so the minimum and maximum are well-defined. Correctness of concurrent (ϵ,δ)-bounded objects is then formally defined as follows:

▶ **Definition 5.** *A concurrent randomized algorithm A implements an (ϵ,δ)-bounded $\mathcal{I}$ object if for every query Q returning in an execution of A with any schedule σ and a randomly sampled coin flip vector $\vec{c} \in \Omega^\infty$,*

$$ret(Q, H(A,\sigma,\vec{c})) \geq v^{\mathcal{I}}_{min}(H(A,\sigma,\vec{c}),Q) - \epsilon \textit{ with probability at least } 1-\frac{\delta}{2},$$

and

$$ret(Q, H(A,\sigma,\vec{c})) \leq v^{\mathcal{I}}_{max}(H(A,\sigma,\vec{c}),Q) + \epsilon \textit{ with probability at least } 1-\frac{\delta}{2}.$$

In some algorithms, ϵ depends on the stream size, i.e., the number of updates preceding a query; to avoid cumbersome notations we use a single variable ϵ, which should be set to the maximum value that the sketch's ϵ bound takes during the query's execution interval. Since the query returns, its execution interval is necessarily finite, and so ϵ is bounded.

The following theorem shows that IVL implementations allow us to leverage the "legacy" analysis of a sequential object's error bounds.

▶ **Theorem 6.** *Consider a sequential specification $\{A(\vec{c})\}_{\vec{c}\in\Omega^\infty}$ of an (ϵ,δ)-bounded $\mathcal{I}$ object (Definition 4). Let A' be an IVL implementation of A (Definition 3). Then A' implements a concurrent (ϵ,δ)-bounded $\mathcal{I}$ object (Definition 5).*

Proof. Consider a skeleton history $H = H^?(A',\sigma)$ of A' with some schedule σ, and a query Q that returns in H. As A' is an IVL implementation of A, there exist linearizations H_1 and H_2 of H, such that for every $\vec{c}\in\Omega^\infty$, $\text{ret}(Q,\tau_{A(\vec{c})}(H_1)) \le \text{ret}(Q, H(A,\sigma,\vec{c})) \le \text{ret}(Q,\tau_{A(\vec{c})}(H_1))$. As $\{A(\vec{c})\}_{\vec{c}\in\Omega^\infty}$ captures a sequential (ϵ,δ)-bounded $\mathcal{I}$ object, $\text{ret}(Q,\tau_{A(\vec{c})}(H_i)$ is bounded as follows:

$$\text{ret}(Q,\tau_{A(\vec{c})}(H_1)) \ge \text{ret}(Q,\tau_{\mathcal{I}}(H_1)) - \epsilon \text{ with probability at least } 1-\frac{\delta}{2},$$

and

$$\text{ret}(Q,\tau_{A(\vec{c})}(H_2)) \le \text{ret}(Q,\tau_{\mathcal{I}}(H_2)) + \epsilon \text{ with probability at least } 1-\frac{\delta}{2}.$$

Furthermore, by definition of v_{min} and v_{max}:

$$\text{ret}(Q,\tau_{\mathcal{I}}(H_1)) \ge v^{\mathcal{I}}_{min}(H(A',\sigma,\vec{c}),Q); \qquad \text{ret}(Q,\tau_{A(\vec{c})}(H_2)) \le v^{\mathcal{I}}_{max}(H(A',\sigma,\vec{c}),Q).$$

Therefore, with probability at least $1-\frac{\delta}{2}$, $\text{ret}(Q,H(A',\sigma,\vec{c})) \ge v^{\mathcal{I}}_{min}(H(A',\sigma,\vec{c}),Q)-\epsilon$ and with probability at least $1-\frac{\delta}{2}$, $\text{ret}(Q,H(A',\sigma,\vec{c})) \ge v^{\mathcal{I}}_{max}(H(A',\sigma,\vec{c}),Q)+\epsilon$, as needed. ◀

While easy to prove, Theorem 6 shows that IVL is in some sense the "natural" correctness property for (ϵ,δ)-bounded objects; it shows that IVL is a sufficient criterion for keeping the error bounded. It is less restrictive – and as we show below, sometimes cheaper to implement – than linearizability, and yet strong enough to preserve the salient properties of sequential executions of (ϵ,δ)-bounded objects. As noted in Section 3.4, previously suggested relaxations do not inherently guarantee that error bounds are preserved. For example, regular-like semantics, where a query "sees" some subset of the concurrent updates [33], satisfy IVL (and hence bound the error) for monotonic objects albeit not for general ones. Indeed, if object values can both increase and decrease, the results returned under such regular-like semantics can arbitrarily diverge from possible sequential ones.

The importance of Theorem 6 is that it allows us to leverage the vast literature on sequential (ϵ,δ)-bounded objects [27, 12, 5, 24, 7, 1] in concurrent implementations. As an example, in the next section we give an example of an IVL parallelization of a popular data sketch. By Theorem 6, it preserves the original sketch's error bounds.

5 Concurrent CountMin sketch

Cormode et al. propose the *CountMin (CM)* sketch [7], which estimates the frequency of an item a, denoted f_a, in a data stream, where the data stream is over some alphabet Σ. The CM sketch supports two operations: UPDATE(a), which updates the object based on $a\in\Sigma$, and QUERY(a), which returns an estimate on the number of UPDATE(a) calls that preceded the query.

The sequential algorithm's underlying data structure is a matrix c of $w \times d$ counters, for some parameters w, d determined accordingly to the desired error and probability bounds. The sketch uses d hash functions $h_i : \Sigma \mapsto [1, w]$, for $1 \leq i \leq d$. The hash functions are generated using the random coin flip vector $\vec{c}$, and have certain mathematical properties whose details are not essential for understanding this paper. The algorithm's input (i.e., the schedule) is generated by a so-called *weak adversary*, namely, the input is independent of the randomly drawn hash functions.

The CountMin sketch, denoted $CM(\vec{c})$, is illustrated in Figure 1, and its pseudo-code is given in Algorithm 1. On UPDATE(a), the sketch increments counters $c[i][h_i(a)]$ for every $1 \leq i \leq d$. QUERY(a) returns $\hat{f}_a = \min_{1 \leq i \leq d}\{c[i][h_i(a)]\}$.

Algorithm 1 CountMin($\vec{c}$) sketch.

```
 1: array c[1...d][1...w]                              ▷ Initialized to 0
 2: hash functions h_1,...h_d         ▷ h_i : Σ ↦ [1,w], initialized using c⃗

 3: procedure UPDATE(a)
 4:     for i : 1 ≤ i ≤ d do
 5:         atomically increment c[i][h_i(a)]
 6: procedure QUERY(a)
 7:     min ← ∞
 8:     for i : 1 ≤ i ≤ d do
 9:         c ← c[i][h_i(a)]
10:         if min > c  then min ← c
11:     return min
```

Cormode et al. show that, for desired bounds δ and α, given appropriate values of w and d, with probability at least $1 - \delta$, the estimate of a query returning $\hat{f}_a$ is bounded by $f_a \leq \hat{f}_a \leq f_a + \alpha n$, where n is the the number of updates preceding the query and f_a is the ideal value. Thus, for $\epsilon = \alpha n$, CM is a sequential (ϵ, δ)-bounded object. Its sequential specification distribution is $\{CM(\vec{c})\}_{\vec{c} \in \Omega^\infty}$.

Proving an error bound for an efficient parallel implementation of the CM sketch for existing criteria is not trivial. Using the framework defined by Rinberg et al. [32] requires the query to take a strongly linearizable snapshot of the matrix [29]. Distributional linearizability [2] necessitates an analysis of the error bounds directly in the concurrent setting, without leveraging the sketch's existing analysis for the sequential setting.

Instead, we utilize IVL to leverage the sequential analysis for a parallelization that is not strongly linearizable (or indeed linearizable), without using a snapshot. Consider the straightforward parallelization of the CM sketch, whereby the operations of Algorithm 1 may be invoked concurrently and each counter is atomically incremented on line 5 and read on line 9. We call this parallelization $PCM(\vec{c})$. We next prove that i is IVL.

▶ **Lemma 7.** *PCM is an IVL implementation of CM.*

Proof. Let H be a history of an execution σ of PCM. Let H_1 be a linearization of $H^?$ such that every query is linearized prior to every concurrent update, and let H_2 be a linearization of $H^?$ such that every query is linearized after every concurrent update. Let σ_i for $i = 1, 2$ be a sequential execution of CM with history H_i. Consider some $Q =$QUERY(a) that returns in H, and let $U_1, \ldots, U_k$ be the concurrent updates to Q.

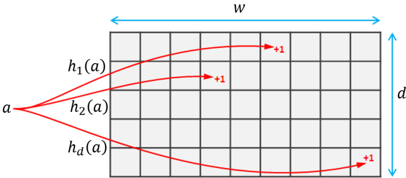


Figure 1 An example CountMin sketch, of size $w \times d$, where $h_1(a) = 6$, $h_2(a) = 4$ and $h_d(a) = w$.[2]

Denote by $c_\sigma(Q)[i]$ the value read by Q from $c[i][h_i(a)]$ in line 9 of Algorithm 1 in an execution σ. As processes only increment counters, for every $1 \le i \le d$, $c_\sigma(Q)[i]$ is at least $c_{\sigma_1}(Q)[i]$ (the value when the query starts) and at most $c_{\sigma_2}(Q)[i]$ (the value when all updates concurrent to the query complete). Therefore, $c_{\sigma_1}(Q)[i] \le c_\sigma(Q)[i] \le c_{\sigma_2}(Q)[i]$.

Consider a randomly sampled coin flip vector $\vec{c} \in \Omega^\infty$. Let j be the loop index the last time query Q alters the value of its local variable *min* (line 10), i.e., the index of the minimum read value. As a query in a history of $CM(\vec{c})$ returns the minimum value in the array, $\text{ret}(Q, \tau_{CM(\vec{c})}(H_1)) \le c_{\sigma_1}(Q)[j]$. Furthermore, $\text{ret}(Q, \tau_{CM(\vec{c})}(H_2))$ is at least $c_\sigma(Q)[j]$, otherwise Q would have read this value and returned it instead. Therefore:

$$\text{ret}(Q, \tau_{CM(\vec{c}))}(H_1)) \le \text{ret}(Q, H(PCM, \sigma, \vec{c})) \le \text{ret}(Q, \tau_{CM(\vec{c})}(H_2))$$

As needed. ◀

Combining Lemma 7 and Theorem 6, and by utilizing the sequential error analysis from [7], we have shown the following corollary:

▶ **Corollary 8.** *Let $\hat{f}_a$ be a return value from query Q. Let f_a^{start} be the ideal frequency of element a when the query starts, and let f_a^{end} be the ideal frequency of element a at its end, and let $\epsilon = \alpha n$ where n is the stream length at the end of the query. Then:*

$$f_a^{start} \le \hat{f}_a \le f_a^{end} + \epsilon \textit{ with probability at least } 1 - \delta.$$

The following example demonstrates that PCM is not a linearizable implementation of CM.

▶ **Example 9.** Consider the following execution σ of PDC: Assume that $\vec{c}$ is such that $h_1(a) = h_2(a) = 1$, $h_1(b) = 2$ and $h_2(b) = 1$. Assume that initially

$$c = \begin{array}{|c|c|}\hline 1 & 4 \\ \hline 2 & 3 \\ \hline \end{array}.$$

First, process p invokes U =UPDATE(a) which increments $c[1][1]$ to 2 and stalls. Then, process p invokes Q_1 =QUERY(a) which reads $c[1][1]$ and $c[2][1]$ and returns 2, followed by Q_2 =QUERY(b) which reads $c[1][2]$ and $c[2][1]$ and returns 2. Finally, process p increments $c[2][1]$ to be 3.

Assume by contradiction that H is a linearization if σ, and $H \in CM(\vec{c})$. The return values imply that $U \prec_H Q_1$ and $Q_2 \prec_H U$. As H is a linearization, it maintains the partial order of operations in σ, therefore $Q_1 \prec_H Q_2$. A contradiction.

[2] Source: `https://stackoverflow.com/questions/6811351/explaining-the-count-sketch-algorithm`, with alterations.

6 Shared batched counter

We now show an example where IVL is inherently less costly than linearizability. In Section 6.1 we present an IVL batched counter, and show that the UPDATE operation has step complexity $O(1)$. The algorithm uses single-writer-multi-reader(SWMR) registers. In Section 6.2 we prove that all linearizable implementations of a batched counter using SWMR registers have step complexity $\Omega(n)$ for the UPDATE operation. This is in contrast with standard (non-batched) counters, which can be implemented with a constant update time. Intuitively, the difference is that in a standard counter, all intermediate values "occur" in an execution (provided that return values are all integers and increments all add one), and so all values allowed by IVL are also allowed by linearizability.

6.1 IVL batched counter

We consider a *batched counter* object, which supports the operations UPDATE(v) where $v \geq 0$, and READ(). The sequential specification for this object is simple: a READ operation returns the sum of all values passed to UPDATE operations that precede it, and 0 if no UPDATE operations were invoked. The UPDATE operation returns nothing. When the object is shared, we denote an invocation of UPDATE by process i as UPDATE$_i$. We denote the sequential specification of the batched counter by $\mathcal{H}$.

Algorithm 2 Algorithm for process p_i, implementing an IVL batched counter.

```
1: shared array v[1...n]
2: procedure UPDATE_i(v)
3:     v[i] ← v[i] + v
4: procedure READ
5:     sum ← 0
6:     for i : 1 ≤ i ≤ n do
7:         sum ← sum + v[i]
8:     return sum
```

Algorithm 2 presents an IVL implementation for a batched counter with n processes using an array v of n SWMR registers. The implementation is a trivial parallelization: an UPDATE operation increments the process's local register while a READ scans all registers and returns their sum. This implementation is not linearizable because the reader may see a later UPDATE and miss an earlier one, as illustrated in Figure 2. We now prove the following lemma:

▶ **Lemma 10.** *Algorithm 2 is an IVL implementation of a batched counter.*

Proof. Let H be a well-formed history of an execution σ of Algorithm 2. We first complete H be adding appropriate responses to all UPDATE operations, and removing all pending READ operations, we denote this completed history as H'.

Let H_1 be a linearization of $H'^?$ given by ordering UPDATE operations by their return steps, and ordering READ operations after all preceding operations in $H'^?$, and before concurrent ones. Operations with the same order are ordered arbitrarily. Let H_2 be a linearization of $H'^?$ given by ordering UPDATE operations by their invocations, and ordering READ operations operations before all operations that precede them in $H'^?$, and after concurrent ones. Operations with the same order are ordered arbitrarily. Let σ_i for $i = 1, 2$ be a sequential execution of a batched counter with history $\tau_{\mathcal{H}}(H_i)$.

By construction, H_1 and H_2 are linearizations of $H'^?$. Let R be some READ operation that completes in H. Let $v[1 \dots n]$ be the array as read by R in σ, $v_1[1 \dots n]$ as read by R in σ_1 and $v_2[1 \dots n]$ as read by R in σ_2. To show that $\text{ret}(R, \tau_{\mathcal{H}}(H_1)) \leq \text{ret}(R, H) \leq \text{ret}(R, \tau_{\mathcal{H}}(H_2))$, we show that $v_1[j] \leq v[j] \leq v_2[j]$ for every index $1 \leq j \leq n$.

For some index j, only p_j can increment $v[j]$. By the construction of H_1, all UPDATE operations that precede R in H also precede it in H_1. Therefore $v_1[j] \leq v[j]$. Assume by contradiction that $v[j] > v_2[j]$. Consider all concurrent UPDATE operations to R. After all concurrent UPDATE operations end, the value of index j is $v' \geq v[j] > v_2[j]$. However, by construction, R is ordered after all concurrent UPDATE operations in H_2, therefore $v' \leq v_2[j]$. This is a contradiction, and therefore $v[j] \leq v_2[j]$.

As all entries in the array are non-negative, it follows that $\sum_{j=1}^{n} v_1[j] \leq \sum_{j=1}^{n} v[j] \leq \sum_{j=1}^{n} v_2[j]$, and therefore $\text{ret}(R, \tau_{\mathcal{H}}(H_1)) \leq \text{ret}(R, H) \leq \text{ret}(R, \tau_{\mathcal{H}}(H_2))$. ◀

Figure 2 shows a possible concurrent execution of Algorithm 2. This algorithm can efficiently implement a distributed or NUMA-friendly counter, as processes only access their local registers thereby lowering the cost of incrementing the counter. This is of great importance, as memory latencies are often the main bottleneck in shared object emulations [25]. As there are no waits in either UPDATE or READ, it follows that the algorithm is wait-free. Furthermore, the READ step complexity is $O(n)$, and the UPDATE step complexity is $O(1)$. Thus, we have shown the following theorem:

▶ **Theorem 11.** *There exists a bounded wait-free IVL implementation of a batched counter using only SWMR registers, such that the step complexity of* UPDATE *is $O(1)$ and the step complexity of* READ *is $O(n)$.*

6.2 Lower bound for linearizable batched counter object

The incentive for using an IVL batched counter instead of a linearizable one stems from a lower bound on the step-complexity of a wait-free linearizable batched counter implementation from SWMR registers. To show the lower bound we first define the binary snapshot object. A *snapshot object* has n components written by separate processes, and allows a reader to capture the shared variable states of all n processes instantaneously. We consider the *binary snapshot object*, in which each state component may be either 0 or 1 [20]. The object supports the $\text{UPDATE}_i(v)$ and SCAN operations, where the former sets the state of component i to value a $v \in \{0, 1\}$ and the latter returns all processes states instantaneously. It is trivial that the SCAN operation must read all states, therefore its lower bound step complexity is $\Omega(n)$. Israeli and Shriazi [21] show that the UPDATE step complexity of any implementation of a snapshot object from SWMR registers is also $\Omega(n)$. This lower bound was shown to hold also for multi writer registers [3]. While their proof was originally given for a multi value snapshot object, it holds in the binary case as well [20].

Figure 2 A possible concurrent history of the IVL batched counter: p_1 and p_2 update their local registers, while p_3 reads. p_3 returns an intermediate value between the counter's state when it starts, which is 0, and the counter's state when it completes, which is 10.

Algorithm 3 Algorithm for process p_i, solving binary snapshot with a batched counter object.

```
1: local variable v_i                                          ▷ Initialized to 0
2: shared batched counter object BC

3: procedure UPDATE_i(v)
4:     if v_i = v then return
5:     v_i ← v
6:     if v = 1 then BC.UPDATE_i(2^i)
7:     if v = 0 then BC.UPDATE_i(2^n − 2^i)
8: procedure SCAN
9:     sum ← BC.READ()
10:    v[0 . . . n − 1] ← [0 . . . 0]                   ▷ Initialize an array of 0's
11:    for i : 0 ≤ i ≤ n − 1 do
12:        if bit i is set in sum then v[i] ← 1
13:    return v[0 . . . n − 1]
```

To show a lower bound on the UPDATE operation of wait-free linearizable batched counters, we show a reduction from a binary snapshot to a batched counter in Algorithm 3. It uses a local variable v_i and a shared batched counter object. In a nutshell, the idea is to encode the value of the ith component of the binary snapshot using the ith least significant bit of the counter. When the component changes from 0 to 1, UPDATE_i adds 2^i, and when it changes from 1 to 0, UPDATE_i adds $2^n - 2^i$. We now prove the following invariant:

▶ **Invariant 1.** *At any point t in history H of a sequential execution of Algorithm 3, the sum held by the counter is $c \cdot 2^n + \sum_{i=0}^{n-1} v_i 2^i$, such that v_i is the parameter passed to the last invocation of UPDATE_i in H' before t if such invocation exists, and 0 otherwise, for some integer $c \in \mathbb{N}$.*

Proof. We prove the invariant by induction on the length of H, i.e., the number of invocations in H, denoted t. As H is a sequential history, each invocation is followed by a response.

Base: The base if for $t = 0$, i.e., H is the empty execution. In this case no updates have been invoked, therefore $v_i = 0$ for all $0 \leq i \leq n - 1$. The sum returned by the counter is 0. Choosing $c = 0$ satisfies the invariant.

Induction step: Our induction hypothesis is that the invariant holds for a history of length t. We prove that it holds for a history of length $t + 1$. The last invocation can be either a SCAN, or an UPDATE(v) by some process p_i. If it is a SCAN, then the counter value doesn't change and the invariant holds. Otherwise, it is an UPDATE(v). Here, we note two cases. Let v_i be p_i's value prior to the UPDATE(v) invocation. If $v = v_i$, then the UPDATE returns without altering the sum and the invariant holds. Otherwise, $v \neq v_i$. We analyze two cases, $v = 1$ and $v = 0$. If $v = 1$, then $v_i = 0$. The sum after the update is $c \cdot 2^n + \sum_{i=0}^{n-1} v_i 2^i + 2^i = c \cdot 2^n + \sum_{i=0}^{n-1} v_i' 2^i$, where $v_j' = v_j$ if $j \neq i$, and $v_i' = 1$, and the invariant holds. If $v = 0$, then $v_i = 1$. The sum after the update is $c \cdot 2^n + \sum_{i=0}^{n-1} v_i 2^i + 2^n - 2^i = (c + 1) \cdot 2^n + \sum_{i=0}^{n-1} v_i' 2^i$, where $v_j' = v_j$ if $j \neq i$, and $v_i' = 1$, and the invariant holds. ◀

Using the invariant, we prove the following lemma:

▶ **Lemma 12.** *For any sequential history H, if a SCAN returns v_i, and $\text{UPDATE}_i(v)$ is the last update invocation in H prior to the SCAN, then $v_i = v$. If no such update exists, then $v_i = 0$.*

Proof. Let S be a SCAN in H'. Consider the sum *sum* as read by scan S. From Invariant 1, the value held by the counter is $c \cdot 2^n + \sum_{i=0}^{n-1} v_i 2^i$. There are two cases, either there is an update invocation prior to S, or there isn't. If there isn't, then by Invariant 1 the corresponding $v_i = 0$. The process sees bit $i = 0$, and will return 0. Therefore, the lemma holds.

Otherwise, there is a an update prior to S in H. As the sum is equal to $c \cdot 2^n + \sum_{i=0}^{n-1} v_i 2^i$, by Invariant 1, bit i is equal to 1 iff the parameter passed to the last invocation of update was 1. Therefore, the scan returns the parameter of the last update and the lemma holds. ◀

▶ **Lemma 13.** *Algorithm 3 implements a linearizable binary snapshot using a linearizable batched counter.*

Proof. Let H be a history of Algorithm 3, and let H' be H where each operation is linearized at its access to the linearizable batched counter, or its response if $v_i = v$ on line 4. Applying Lemma 12 to H', we get $H' \in \mathcal{H}$ and therefore H is linearizable. ◀

It follows from the algorithm that if the counter object is bounded wait-free then the SCAN and UPDATE operations are bounded wait-free. Therefore, the lower bound proved by Israeli and Shriazi [21] holds, and the UPDATE must take $\Omega(n)$ steps. Other than the access to the counter in the UPDATE operation, it takes $O(1)$ steps. Therefore, the access to the counter object must take $\Omega(n)$ steps. We have proven the following theorem.

▶ **Theorem 14.** *For any linearizable wait-free implementation of a batched counter object with n processes from SWMR registers, the step-complexity of the* UPDATE *operation is $\Omega(n)$.*

7 Conclusion

We have presented IVL, a new correctness criterion that provides flexibility in the return values of quantitative objects while bounding the error that this may introduce. IVL has a number of desirable properties: First, like linearizability, it is a local property, allowing designers to reason about each part of the system separately. Second, also like linearizability but unlike other relaxations of it, IVL preserves the error bounds of PAC objects. Third, IVL is generically defined for all quantitative objects, and does not necessitate object-specific definitions. Finally, IVL is inherently amenable to cheaper implementations than linearizability in some cases.

Via the example of a CountMin sketch, we have illustrated that IVL provides a generic way to efficiently parallelize data sketches while leveraging their sequential error analysis to bound the error in the concurrent implementation.

The notion of IVL raises a number of questions for future research. First, in this work we have shown that IVL is a sufficient condition for parallel (ϵ, δ)-bounded objects, in that it preserves their sequential error. It would be interesting to investigate whether IVL is also necessary, or whether some weaker condition is sufficient. Second, we would like to extend IVL to objects like priority queues, which are in a sense "semi quantitative", namely, their return values are associated with a quantity (the priority) but also include a non-quantitative element (the queued item). Additionally, it could be useful to apply IVL to additional sketches, and to study whether it can meaningfully apply to non-monotonic quantities.

References

1. Pankaj K Agarwal, Graham Cormode, Zengfeng Huang, Jeff M Phillips, Zhewei Wei, and Ke Yi. Mergeable summaries. *ACM Transactions on Database Systems (TODS)*, 38(4):1–28, 2013.
2. Dan Alistarh, Trevor Brown, Justin Kopinsky, Jerry Z Li, and Giorgi Nadiradze. Distributionally linearizable data structures. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, pages 133–142. ACM, 2018.
3. Hagit Attiya, Faith Ellen, and Panagiota Fatourou. The complexity of updating multi-writer snapshot objects. In *International Conference on Distributed Computing and Networking*, pages 319–330. Springer, 2006.
4. Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. Unifying concurrent objects and distributed tasks: Interval-linearizability. *Journal of the ACM (JACM)*, 65(6):1–42, 2018.
5. Jacek Cichon and Wojciech Macyna. Approximate counters for flash memory. In *2011 IEEE 17th International Conference on Embedded and Real-Time Computing Systems and Applications*, volume 1, pages 185–189. IEEE, 2011.
6. Graham Cormode, Minos Garofalakis, Peter J Haas, and Chris Jermaine. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends in Databases*, 4(1–3):1–294, 2012.
7. Graham Cormode and Shan Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.
8. Graham Cormode, Shanmugavelayutham Muthukrishnan, and Ke Yi. Algorithms for distributed functional monitoring. *ACM Transactions on Algorithms (TALG)*, 7(2):1–20, 2011.
9. Mayur Datar and Piotr Indyk. Comparing data streams using hamming norms. In *Proceedings 2002 VLDB Conference: 28th International Conference on Very Large Databases (VLDB)*, page 335. Elsevier, 2002.
10. Druid. Apache DataSketches (Incubating). `https://incubator.apache.org/clutch/datasketches.html`, Accessed May 14, 2020.
11. Druid. Druid. `https://druid.apache.org/blog/2014/02/18/hyperloglog-optimizations-for-real-world-systems.html`, Accessed May 14, 2020.
12. Philippe Flajolet. Approximate counting: a detailed analysis. *BIT Numerical Mathematics*, 25(1):113–134, 1985.
13. Philippe Flajolet and G Nigel Martin. Probabilistic counting. In *24th Annual Symposium on Foundations of Computer Science (sfcs 1983)*, pages 76–82. IEEE, 1983.
14. Phillip B Gibbons and Srikanta Tirthapura. Estimating simple functions on the union of data streams. In *Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures*, pages 281–291, 2001.
15. Wojciech Golab, Lisa Higham, and Philipp Woelfel. Linearizable implementations do not suffice for randomized distributed computation. In *Proceedings of the forty-third annual ACM symposium on Theory of computing*, pages 373–382, 2011.
16. Thomas A Henzinger, Christoph M Kirsch, Hannes Payer, Ali Sezgin, and Ana Sokolova. Quantitative relaxation of concurrent data structures. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 317–328, 2013.
17. Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.
18. Stefan Heule, Marc Nunkesser, and Alexander Hall. Hyperloglog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm. In *Proceedings of the 16th International Conference on Extending Database Technology*, pages 683–692, 2013.
19. Hillview. Hillview: A Big Data Spreadsheet. `https://research.vmware.com/projects/hillview`, Accessed May 14, 2020.
20. Jaap-Henk Hoepman and John Tromp. Binary snapshots. In *International Workshop on Distributed Algorithms*, pages 18–25. Springer, 1993.

21 Amos Israeli and Asaf Shirazi. The time complexity of updating snapshot memories. *Information Processing Letters*, 65(1):33–40, 1998.

22 Leslie Lamport. On interprocess communication. *Distributed computing*, 1(2):86–101, 1986.

23 Leslie Lamport. Concurrent reading and writing of clocks. *ACM Transactions on Computer Systems (TOCS)*, 8(4):305–310, 1990.

24 Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 101–114, 2016.

25 Nihar R Mahapatra and Balakrishna Venkatrao. The processor-memory bottleneck: problems and solutions. *XRDS: Crossroads, The ACM Magazine for Students*, 5(3es):2, 1999.

26 Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *International Conference on Database Theory*, pages 398–412. Springer, 2005.

27 Robert Morris. Counting large numbers of events in small registers. *Communications of the ACM*, 21(10):840–842, 1978.

28 Gil Neiger. Set-linearizability. In *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*, page 396, 1994.

29 Sean Ovens and Philipp Woelfel. Strongly linearizable implementations of snapshots and other types. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 197–206, 2019.

30 Presto. HyperLogLog in Presto: A significantly faster way to handle cardinality estimation. `https://engineering.fb.com/data-infrastructure/hyperloglog/`, Accessed May 14, 2020.

31 Arik Rinberg and Idit Keidar. Intermediate value linearizability: A quantitative correctness criterion. *arXiv preprint arXiv:2006.12889*, 2020.

32 Arik Rinberg, Alexander Spiegelman, Edward Bortnikov, Eshcar Hillel, Idit Keidar, Lee Rhodes, and Hadar Serviansky. Fast concurrent data sketches. In *Proceedings of the 2020 ACM Symposium on Principles and Practice of Parallel Programming*. ACM, 2020.

33 Charalampos Stylianopoulos, Ivan Walulya, Magnus Almgren, Olaf Landsiedel, and Marina Papatriantafilou. Delegation sketch: a parallel design with support for fast and accurate concurrent operations. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

The Splay-List: A Distribution-Adaptive Concurrent Skip-List

Vitaly Aksenov
ITMO, St. Petersburg, Russia
aksenov.vitaly@gmail.com

Dan Alistarh
IST Austria, Klosterneuburg, Austria
dan.alistarh@ist.ac.at

Alexandra Drozdova
ITMO University, St. Petersburg, Russia
drsanusha1@gmail.com

Amirkeivan Mohtashami
Sharif University of Technology, Tehran, Iran
akmohtashami97@gmail.com

Abstract

The design and implementation of efficient concurrent data structures has seen significant attention. However, most of this work has focused on concurrent data structures providing good *worst-case* guarantees. In real workloads, objects are often accessed at different rates, since access distributions may be non-uniform. Efficient distribution-adaptive data structures are known in the sequential case, e.g. the splay-trees; however, they often are hard to translate efficiently in the concurrent case.

In this paper, we investigate distribution-adaptive concurrent data structures, and propose a new design called the splay-list. At a high level, the splay-list is similar to a standard skip-list, with the key distinction that the height of each element adapts dynamically to its access rate: popular elements "move up," whereas rarely-accessed elements decrease in height. We show that the splay-list provides order-optimal amortized complexity bounds for a subset of operations, while being amenable to efficient concurrent implementation. Experimental results show that the splay-list can leverage distribution-adaptivity to improve on the performance of classic concurrent designs, and can outperform the only previously-known distribution-adaptive design in certain settings.

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

Keywords and phrases Data structures, self-adjusting, concurrency

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.3

Related Version A full version of the paper is available at [2], `https://arxiv.org/abs/2008.01009`.

Supplementary Material The code is available at `https://github.com/demon1999/splaylist`.

Funding *Vitaly Aksenov*: Government of Russian Federation (Grant 08-08).
Dan Alistarh: ERC Starting Grant 805223 ScaleML.

1 Introduction

The past decades have seen significant effort on designing efficient concurrent data structures, leading to fast variants being known for many classic data structures, such as hash tables, e.g. [19, 14], skip lists, e.g. [11, 13, 17], or search trees, e.g. [10, 20]. Most of this work has focused on efficient concurrent variants of data structures with optimal *worst-case* guarantees. However, in many real workloads, the access rates for individual objects are not uniform. This fact is well-known, and is modelled in several industrial benchmarks, such as YCSB [8],

© Vitaly Aksenov, Dan Alistarh, Alexandra Drozdova, and Amirkeivan Mohtashami;
licensed under Creative Commons License CC-BY

or TPC-C [21], where the generated access distributions are heavy-tailed, e.g., following a Zipf distribution [8]. While in the sequential case the question of designing data structures which adapt to the access distribution is well-studied, see e.g. [16] and references therein, the concurrent case is significantly less explored. The intuitive reason for this difficulty is that self-adjusting data structures require non-trivial and frequent pointer manipulations, such as node rotations in a balanced search tree, which can be complex to implement concurrently.

To date, the CBTree [1] is the only concurrent data structure which leverages the skew in the access distribution for faster access. At a high level, the CBTree is a concurrent search tree maintaining internal balance with respect to the access statistics per node. Its sequential variant is static optimal, i.e., the complexity of performing requests σ coincides with the best complexity of performing σ on the static data structure built given requests in advance, and empirical results show that it provides significant performance benefits over a classic non-adaptive concurrent design for skewed workloads. At the same time, the CBTree may be seen as fairly complex, due to the difficulty of re-balancing in a concurrent setting, and the paper's experimental validation suggests that maintaining exact access statistics and balance in a concurrent setting come at some performance cost – thus, the authors propose a limited-concurrency variant, where rebalancing is delegated to a single thread.

In this paper, we revisit the topic of distribution-adaptive concurrent data structures, and propose a design called the *splay-list*. At a very high level, the splay-list is very similar to a classic skip-list [22]: it consists of a sequence of sorted lists, ordered by containment, where the bottom-most list contains all the elements present, and each higher list contains a sub-sample of the elements from the previous list. The crucial distinction is that, in contrast to the original skip-list, where the height of each element is chosen randomly, in the splay-list, the height of each element *adapts* to its access rate: elements that are accessed more often move "up," and will be faster to access, whereas elements which are accessed less often are demoted towards the bottom-most list. Intuitively, this property ensures that popular elements are closer to the "top" of the list, and are thus accessed faster.

This intuition can be made precise: we provide a rebalancing algorithm which ensures that, after m operations, the amortized search and delete time for an item x in a sequential splay-list is $\mathcal{O}\left(\log \frac{m}{f(x)}\right)$ where $f(x)$ is the number of previous searches for x, whereas insertion takes amortized $\mathcal{O}(\log m)$ time. This asymptotically matches the guarantees of the CBTree [1], and implies static optimality. Since maintaining exact access statistics for each object can hurt performance – as every search has to update the counters – we introduce and present guarantees for variants of the data structure which only maintains *approximate* access counts. If rebalancing is only performed with probability $1/c$ – meaning that only this fraction of readers will have to write – then we show that the expected amortized cost of a contains operation becomes $\mathcal{O}\left(c \log \frac{m}{f(x)}\right)$. Since c is a constant, this trade-off can be beneficial.

From the perspective of concurrent access, an advantage of the splay-list is that it can be easily implemented on top of existing skip-list designs [14]: the pointer changes for promotion and demotion of nodes are operationally a subset of skip-list insertion and deletion operations [12]. At the same time, our design does come with some limitations: (1) since it is based on a skip-list backbone, the splay-list may have higher memory footprint and path length relative to a tree; (2) as discussed above, approximate access counts are necessary for good performance, but come at an increase in amortized expected cost, which we believe to be inherent; (3) for simplicity, our update operations are lock-based (although this limitation could be removed).

We implement the splay-list in C++ and compare it with the CBTree and a regular skip-list on uniform and skewed workloads, and for different update rates. Experiments show that the splay-list can indeed leverage workload skew for higher performance, and that it can scale well when access counts are approximate.

Overall, the results suggest a trade-off between the performance of the two data structures and the workload characteristics, both in terms of access distribution and access types. The fact that the splay-list can outperform the CBTree in some practical scenarios may appear surprising, given that the splay-list leads to longer access paths on average due to its skip-list backbone. However, our design benefits from allowing additional concurrency, and the caching mechanism serves to hide some of the additional access costs.

Related Work. The literature on *sequential* self-adjusting data structures is well-established, and extremely vast. We therefore do not attempt to cover it in detail, and instead point the reader to classic texts, e.g. [16, 23], for details. Focusing on self-adjusting skip-lists, we note that statically-optimal *deterministic* skip-list-like data structures can be derived from the k-forest structure of Martel [18], or from the working set structure of Iacono [15]. Ciriani et al. [7] provide a similar randomized approach for constructing a self-adjusting skip-list for string dictionary operations in the external memory model. Bagchi et al. [4] introduced a general *biased skip-list* data structure, which maintains balance w.r.t. node height when nodes can have arbitrary weight, while Bose et al. [5] built on biased skip-lists to obtain a *dynamically-optimal* skip-list data structure.

Relative to our work, we note that, naturally, the above theoretical references provide stronger guarantees relative to the splay-list in the sequential setting. At the same time, they are quite complex, and would not extend efficiently to a concurrent setting. Two practical additions that our design brings relative to this prior work is that we are the first to provide bounds even when the access count values are *approximate* (Section 4), and that our concurrent design allows the splay-list adjustment to occur in a single pass (Section 5). CBTree paper [1] posed the existence of an efficient self-balancing skip-list variant as an open question – we answer this question here, in the affirmative.

The splay-list ensures similar complexity guarantees as the CBTree [1], although its structure is different. Both references provide complexity guarantees under *sequential* access. In addition, we provide complexity guarantees in the case where the access counts are maintained via *approximate* counters, in which case the CBTree is not known to provide guarantees. One obvious difference relative to our work is that we are investigating a skip-list-based design. This allows for more concurrency: the proposed practical implementation of CBTree [1] assumes that adjustments are performed only by a dedicated thread, whereas splay-list updates can be performed by any thread. At the same time, our design shares some of the limitations of skip-list-based data structures, as discussed above.

There has been a significant amount of work on efficient concurrent ordered maps, see e.g. [6, 3] for an overview of recent work. However, to our knowledge, the CBTree remained the only non-trivial self-adjusting concurrent data structure.

2 The Sequential Splay-List

The splay-list design builds on the classic skip-list by Pugh [22]. In the following, we will only briefly overview the skip-list structure, and focus on the main technical differences. We refer the reader to "The Art of Multiprocessor Programming" book [14] for a more in-depth treatment of concurrent skip-lists.

Preliminaries. Similar to skip-lists, the splay-list maintains a set of sorted lists, starting from the bottom list, which contains all the objects present in the data structure at a given point in time. We assume that each object consists of a key-value pair. Thus, we use the terms *object* and *key* interchangeably. It is useful to view these lists as stacked on top of each other; a list's index (starting from the bottom one, indexed at 0) is also called its *height*. The lists are also ordered by containment, as a higher-index list contains a subset of the objects present in a lower-index list. The higher-index lists are also called *sub-lists*. Unlike skip-lists, where the choice of which objects should be present in each sub-list is random, a splay-list's structure is adjusted according to the access distribution across keys/objects.

The following definitions make it easier to understand how the operations are handled in splay-lists. The *height of the splay-list* is the number of its sub-lists. The *height of an object* is the height of the highest sub-list containing it. Typically, we do not distinguish between the object and its key. The height of a key u is the height of a corresponding object h_u. Key u is the *parent of key* v *at height* h if u is the largest key whose value is smaller than or equal to v, and whose height is at least h. That is, u is the last key at height h in the traversal path to reach v. Critically, note that, v is its own parent at heights less than or equal to h_v; otherwise, its parent is some node $v \neq u$. In addition, we call the set of objects for which u is the parent at height h, its h*-children* or the *subtree of* u *at height* h*, denoted by* C_u^h.

Our data structure supports three standard methods: `contains`, `insert` and `delete`. We say that a `contains` operation is *successful* (returns *true*) if the requested key is found in the data structure and was not marked as deleted; otherwise, the operation is *unsuccessful*. An `Insert` operation is *successful* (returns *true*) if the requested key was not present upon insertion; otherwise, it is *unsuccessful*. A `Delete` operation is *successful* (returns *true*) if the requested key is found and was not marked as deleted, otherwise, the operation is *unsuccessful*. As suggested, in our implementation a `delete` operation does not always unlink the object from the lists–instead, it may just mark it as deleted.

For every key u, we maintain a counter $hits_u$, which counts the total number of operations which *visit the object*. In particular, *successful* `contains(`u`)`, `insert(`u`)`, and `delete(`u`)` operations increment $hits_u$. Moreover, unsuccessful operations can also increment $hits_u$ if the element is physically present in the data structure, even though logically deleted, upon the operation. In this case, the marked element is still visited by the corresponding operation. (We will re-discuss this notion in the later sections, but the simple intuition here is that we cannot store access counts for elements which are not physically present in the data structure, and therefore ignore their access counts.) We will refer to operations that visits an object with the corresponding key simply as *hit-operations*.

For any set of keys S, we define a function $hits(S)$ to be the total number of hits-operations to the keys in S. As usual, sentinel *head* and *tail* nodes are added to all sub-lists. The height of a sentinel node height is equal to the height of the splay-list itself, and exceeds the height of all other nodes by at least 1. By convention, $hits_{head} = hits_{tail} = 1$.

2.1 The `contains` Operation

Overview. The contains operation consists of two phases: the search phase and the balancing phase. The search phase is exactly as in skip-list: starting from the head of the top-most list, we traverse the current list until we find the last object with key lower than or equal to the search key. If this object's key is not equal to the search key, the search continues from the same object in the lower list. Otherwise, the search operation completes. The process is repeated until either the key is found or the algorithm attempts to descend from the bottom list, in which case the key is not present. If the operation finds its target object, its *hits*

counter is incremented and the balancing phase starts: its goal is to update the splay-list's structure to better fit the access distribution, by traversing the search path backwards and checking two conditions, which we call the *ascent* and *descent* conditions.

We now overview these conditions. For the descent condition, consider two neighbouring nodes at height h, corresponding to two keys $v < u$. Assume that both v and u are on level h, and consider their respective subtrees C_v^h and C_u^h. Assume further that the number of hits to objects in their subtrees ($hits(C_v^h \cup C_u^h)$) became smaller than a given threshold, which we deem appropriate for the nodes to be at height h. (This threshold is updated as more and more operations are performed.) To fix this imbalance, we can "merge" these two subtrees, by descending the right neighbour, u, below v, thus creating a new subtree of higher overall hit count. Similarly, for the ascent condition, we check whether an object's subtree has *higher* hit count than a threshold, in which case we increase its height by one.

Now, we describe the conditions more formally. Assume that the total number of hit-operations to all objects, including those marked for deletion, appearing in splay-list is m, and that the current height of the splay-list is equal to $k+1$. Thus, there are k sub-lists, and the sentinel sub-list containing exclusively *head* and *tail*. Excluding the head, for each object u on a backward path, the following conditions are checked in order.

The Descent Condition. Since u is not the head, there must exist an object v which precedes it in the forward traversal order, such that v has height bigger than or equal to h_u. If

$$hits(C_u^{h_u}) + hits(C_v^{h_u}) = hits(C_u^{h_u} \cup C_v^{h_u}) \leq \frac{m}{2^{k-h_u}},$$

then the object u is demoted from height h_u, by simply being removed from the sub-list at height h_u. The object stays a member of the sub-list at height $h_u - 1$ and h_u is decremented. The backward traversal is then continued at v.

The Ascent Condition. Let w be the first successor of u in the list at height h_u, such that w has height *strictly greater than* h_u. Denote the set of objects with keys in the interval $[u, w)$ with height equal to h_u by S_u. If the following inequality holds:

$$\sum_{x \in S_u} hits(C_x^{h_u}) = hits\left(\bigcup_{x \in S_u} C_x^{h_u}\right) > \frac{m}{2^{k-h_u-1}},$$

then u is promoted and inserted into the sub-list at height $h_u + 1$. The backward traversal is then continued from u, which is now in the higher-index sub-list. The rest of the path at height h_u is skipped. Note that the object u is again checked against the ascent condition at height $h_u + 1$, so it may be promoted again. Also note that the calculated sum is just an interval sum, which can be maintained efficiently, as we show later.

Splay-List Initialization and Expansion. Initially, the splay-list is empty and has only one level with two nodes, head and tail. Suppose that the total number of hits to objects in splay-list is m. The lowest level on which the object can be depends on how low the element can be demoted. Suppose that the current height of the list is $k+1$. Consider any object at the lowest level 0: in the descent condition we compare $hits(C_u^0) + hits(C_v^0)$ against $\frac{m}{2^k}$. While m is less than 2^{k+1}, the object cannot satisfy this condition since $C_v^{h_u} \geq hits_v \geq 1$, but when m becomes larger than this threshold, it could. Thus, we have to increase the height of splay-list and add a new list to allow such an object to be demoted. By that, the height

of the splay-list is always $\log m$. This process is referred to as *splay-list expansion*. Notice that this procedure could eventually lead to a skip-list of unbounded height. However, this height does not exceed 64, since this would mean that we performed at least 2^{64} successful operations which is unrealistic. We discuss ways to make this procedure more practical, i.e., lazily increase the height of an object only on its traversal, in Section 5.

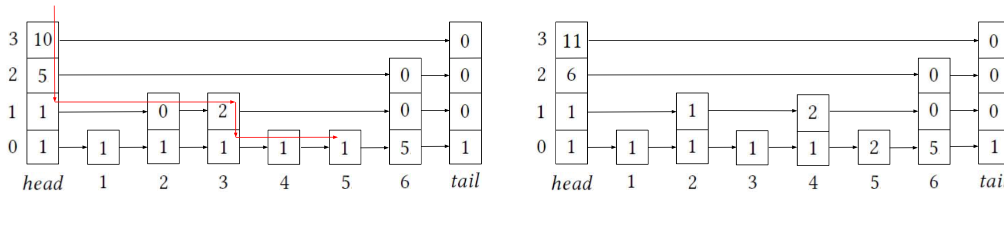
The Backward Pass. Now, we return to the description of the `contains` function. The first phase is the forward pass, which is simply the standard search algorithm which stores the traversal path. If the key is not found, then we stop. Otherwise, suppose that we found an object t. We have to restructure the splay-list by applying ascent and descent conditions. Note, that the only objects that are affected and can change their height lie on the stored path. For that, in each object u we store the total hits to the object itself, $hits_u$ or sh_u, as well as the total number of hits into the "subtree" of each height excluding u, i.e., for all h we maintain $hits_u^h = hits(C_u^h \setminus \{u\})$. Thus, when traversing the path backwards we check the following:

1. If the object $u \neq t$ is a parent of t on some level $h \leq h_u$, we increase its $hits_u^h$ counter.
2. Check the descent condition for v and u as $sh_v + hits_v^{h_u} + sh_u + hits_u^{h_u} \leq \frac{m}{2^{k-h_u}}$. If this is satisfied, demote u and increment $hits_v^{h_u}$ by $sh_u + hits_u^{h_u}$. Continue on the path.
3. Check the ascent condition for u by comparing $\sum_{w \in S_u} sh_w + hits_w^{h_u}$ with $\frac{m}{2^{k-h_u-1}}$. If this is satisfied, add u to the sub-list $h_u + 1$, set $hits_u^{h_u+1}$ to the calculated sum minus sh_u and decrease $hits_v^{h_u+1}$ by the calculated sum, where v is a parent of u at height $h_u + 1$. We then continue with the sub-list on level $h_u + 1$. Below, we describe how to maintain this sum in constant time.

The partial sums trick. Suppose that $p(u)$ is the parent of u on level $h_u + 1$. During the forward pass, we compute the sum of $hits(C_x^{h_u}) = sh_x + hits_x^{h_u}$ over all objects x which lie on the traversal path between $p(u)$ (including it) and u (not including it). Denote this sum by P_u. Thus, to check the ascent condition on the backward pass, we simply have to compare $\sum_{x \in S_u} sh_x + hits_x^{h_u} = sh_{p(u)} + hits_{p(u)}^{h_u+1} - P_u$ against $\frac{m}{2^{k-h_u-1}}$. Observe that the partial sums $hits(\bigcup_{x \in S_u} C_x^{h_u})$ can be increased only by one after each operation which is the *ascent potential*. And since ascent potential for objects in S_u decreases from left to right, the only object on level h that can be promoted is the leftmost object on this level. For the first object u, $hits(\bigcup_{x \in S_u} C_x^{h_u})$ can be calculated as $hits_{p(u)}^{h_u+1} - hits_{p(u)}^{h_u}$. In addition, after the promotion of u, only u and $p(u)$ have their $hits^{h_u+1}$ counters changed. Moreover, there is no need to skip the objects to the left of the promoted object, as suggested by the ascent condition, since there cannot be any such objects.

Example. To illustrate, consider the splay-list provided on Figure 1a. It contains keys $1, \ldots, 6$ with values $m = 10$ and $k = \lfloor \log m \rfloor = 3$. We can instantiate the sets described above as follows: $C_3^1 = \{3, 4, 5\}$, $C_2^1 = \{2\}$, $C_{head}^1 = \{head, 1\}$ and $C_{head}^2 = \{head, 1, 2, \ldots, 5\}$. At the same time, $S_4 = \{4, 5\}$, $S_3 = \{3\}$ and $S_2 = \{2, 3\}$. At the figure, the cell of u at height $h > 0$ contains $hits_u^h$, while the cell at height 0 contains sh_u. For example, $sh_3 = 1$ and $hits_3^1 = sh_4 + sh_5 = 2$, $sh_2 = 1$ and $hits_2^1 = 0$, $sh_1 = 1$ and $hits_{head}^2 = 5$.

Assume we execute `contains(5)`. On the forward path, we find 5 and the path to it is $head \to 2 \to 3 \to 4 \to 5$. We increment m, sh_5, $hits_3^1$ and $hits_{head}^2$ by one. Now, we have to adjust our splay-list on the backward path. We start with 5: we check the descent condition by comparing $hits(C_4^0) + hits(C_5^0) = 3$ with $\frac{m}{2^{k-0}} = \frac{11}{8}$ and the ascent condition by comparing $hits(\bigcup_{x \in S_5} C_x^0) = 2$ with $\frac{m}{2^{k-0-1}} = \frac{11}{4}$. Obviously, neither condition is satisfied. We continue



(a) Before `contains(5)`.

(b) After `contains(5)`.

Figure 1 Example of splay-list.

with 4: the descent condition by comparing $hits(C_3^0) + hits(C_4^0) = 2$ with $\frac{11}{8}$ and the ascent condition by comparing $hits(\bigcup_{x \in S_4} C_x^0) = 3$ with $\frac{11}{4}$ – the ascent condition is satisfied and we promote object 4 to height 1, change the counter $hits_3^1$ to 0 and update S_2 and S_3 to have 4. For 3, we compared $hits(C_2^1) + hits(C_3^1) = 2$ with $\frac{11}{4}$ and $hits(\bigcup_{x \in S_3} C_x^1) = 4$ with $\frac{11}{2}$ – the descent condition is satisfied, we demote object 3 to height 0, change the counter $hits_2^1$ to 1 and remove 3 from S_2. Finally, for 2 we compared $hits(C_{head}^1) + hits(C_2^1) = 4$ with $\frac{11}{4}$ and $hits(\bigcup_{x \in S_2} C_x^1) = 5$ with $\frac{11}{2}$ – none of the conditions are satisfied. As a result we get the splay-list shown on Figure 1b.

2.2 `Insert` and `Delete` operations

Insertion. Inserting a key u is done by first finding the object with the largest key lower than or equal to u. In case an object with the key is found, but is marked as logically deleted, the insertion unmarks the object, increases its hits counter and completes successfully. Otherwise, u is inserted on the lowest level after the found object. This item has hits count set to 1. In both cases, the structure has to be re-balanced on the backward pass as in `contains` operation. Unlike the skip-list, splay-lists always physically inserts into the lowest-level list.

Deletion. This operation needs additional care. The operation first searches for an object with the specified key. If the object is found, then the operation logically deletes it by marking it as `deleted`, increases the hits counter and performs the backward pass. Otherwise, the operation completes.

Notice that we maintain the total number of hits on currently logically deleted objects. When it becomes at least half of m, the total number of hits to all objects, we initialize a new structure, and move all non-deleted objects with corresponding hits to it.

Efficient Rebuild. The only question left is how to build a new structure efficiently enough to amortize the performed delete operations. Suppose that we are given a sorted list of n keys $k_1, \ldots, k_n$ with the number of hit-operations on them $h_1, \ldots, h_n$, where their sum is equal to M. We propose an algorithm that builds a splay-list such that no node satisfies the ascent and descent conditions, using $O(M)$ time and $O(n \log M)$ memory.

The idea behind the algorithm is the following. We provide a recursive procedure that takes the contiguous segment of keys $k_l, \ldots, k_r$ with the total number of accesses $H = h_l + \ldots + h_r$. The procedure finds p such that $2^{p-1} \leq H < 2^p$. Then, it finds a key k_s such that $h_l + \ldots + h_{s-1} \leq \frac{H}{2}$ and $h_{s+1} + \ldots + h_r \leq \frac{H}{2}$. We create a node for the key k_s with the height p, and recursively call the procedure on segments $k_l, \ldots, k_{s-1}$ and $k_{s+1}, \ldots, k_r$. There exists a straightforward implementation which finds the split point s in $O(r - l)$, i.e., linear time. The resulting algorithm works in $O(n \log M)$ time and takes $O(n \log M)$ memory: the depth of the recursion is $\log M$ and on each level we spend $O(n)$ steps.

However, the described algorithm is not efficient if M is less than $n \log M$. To achieve $O(M)$ complexity, we would like to answer the query to find the split point s in $O(1)$ time. For that, we prepare a special array T which contains in sorted order h_1 times key k_1, h_2 times key k_2, ..., h_n times key k_n. To get the required s, at first, we take a subarray of T that corresponds to the segment $[l, r]$ under the process, i.e., h_l times key k_l, ..., h_r times key k_r. Then, we take the key k_i that is located in the middle cell $\lceil \frac{h_l + \ldots + h_r}{2} \rceil$ of the chosen subarray. This i is our required s. Let us calculate the total time spent: the depth of the recursion is $\log M$; there is one element on the topmost level which we insert in $\log M$ lists, there are at most two elements on the next to topmost level which we insert in $\log M - 1$ lists, and etc., there are at most 2^i elements on the i-th level from the top which we insert in $\log M - i$ lists. The total sum is clearly $O(M)$.

Thus, the final algorithm is: if M is larger than $n \log M$, then we execute the first algorithm, otherwise, we execute the second algorithm. The overall construction works in $O(M)$ time and uses $O(n \log M)$ memory.

3 Sequential Splay-List Analysis

Properties. We begin by stating some invariants and general properties of the splay-list.

▶ **Lemma 1.** *After each operation, no object can satisfy the ascent condition.*

Proof. Note that we only consider the hit-operations, i.e., the operations that change *hits* counters, because other operations do not affect any conditions. We will proceed by induction on the total number m of hit-operations on the objects of splay-list.

For the base case $m = 2$, the splay-list contains only head and tail and the hypothesis trivially holds. For the induction step, we assume that the hypothesis holds before the start of the m-th operation, and we verify that it holds after the operation completes.

First, recall that, for a fixed object u, the set S_u is defined to include all objects of the same height between u and the successor of u with height *greater* than h_u. Specifically, we name the sum $\sum_{x \in S_u} hits(C_x^h)$ in the ascent condition as the object u's **ascent potential**. Note that after the forward pass and the increment of sh_u and $hits_v^h$ counters where v is a parent of u on height h, only the objects on the path have their ascent potential increased by one and, thus, only they can satisfy the ascent condition.

Now, consider the restructuring done on the backward pass. If the object u satisfies the descent condition, i.e., v precedes u and $T = hits(C_v^{h_u}) + hits(C_u^{h_u}) \leq \frac{m}{2^{k-h}}$, we have to demote it. After the descent, the ascent potential of the objects between v and u on the lower level $h_u - 1$ have changed. However, the ascent potentials of lower level objects cannot exceed T, meaning that these objects cannot satisfy the ascent condition.

Consider the backward pass, and focus on the set of objects at height h. We claim that only the leftmost object at that height can be promoted, i.e., its preceding object has a height greater than h. This statement is proven by induction on the backward path. Suppose that we have ℓ objects with height h on the path, which we denote by $u_1, u_2, \ldots, u_\ell$. By induction, we know that none of the objects on the path with lower height can ascend higher than h: these objects appear to the right of u_1. We know that each object was accessed at least once, $sh_{u_i} \geq 1$, and, thus, we can guarantee that $hits(\bigcup_{x \in S_{u_1}} C_x^h) > hits(\bigcup_{x \in S_{u_2}} C_x^h) > \ldots > hits(\bigcup_{x \in S_{u_\ell}} C_x^h)$. Since the ascent potentials $hits(\bigcup_{x \in S_{u_i}} C_x^h)$ are increased only by one per operation, the first and the only object that can satisfy the ascent condition is u_1, i.e., the leftmost object with the height h. If it satisfies the condition, we promote it. Consider the predecessor of u_1 on the forward path: the object

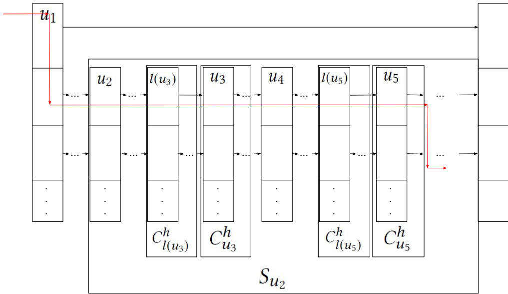


Figure 2 Depiction of the proof of Lemma 3.

v with height $h_v > h$. Object u_1 can be promoted to height h_v, but not higher, since the ascent potential of the objects on the path with height h_v does not change after the promotion of u, and only the leftmost object on that level can ascend. However, note that $hits_v^{h_v}$ can decrease and, thus, it can satisfy the descent condition, while u_1 cannot since $hits_{u_1}^h$ was equal to $hits(\bigcup_{x \in S_{u_1}} C_x^h)$ before the promotion and it satisfied the ascent condition.

Because the only objects that can satisfy the ascent condition lie on the path, and we promoted necessary objects during the backward pass, no object may satisfy the ascent condition at the end of the traversal. That is exactly what we set out to prove. ◀

▶ **Lemma 2.** *Given a hit-operation with argument u, the number of sub-lists visited during the forward pass is at most $3 + \log \frac{m}{sh_u}$.*

Proof. During the forward pass the number of hits does not change; thus, according to Lemma 1, the ascent condition does not hold for u. Hence $sh_u \le \frac{m}{2^{k-h_u-1}}$. We get that $k - h_u - 1 \le \log \frac{m}{sh_u}$. Since during the forward pass $(k+1) - h_u + 1$ sub-lists are visited (notice the sentinel sub-list), the claim follows. ◀

▶ **Lemma 3.** *In each sub-list, the forward pass visits at most four objects that do not satisfy the descent condition.*

Proof. Suppose the contrary and that the algorithm visits at least five objects $u_1, u_2, \ldots, u_5$ in order from left to right, that do not satisfy the descent condition in sub-list h. The height of the objects $u_2, \ldots, u_5$ is h, while the height of u_1 might be higher. See Figure 2.

Note that if the descent condition does not hold for an object u, the demotion of another object of the same height cannot make the descent condition for u satisfiable. Therefore, since the condition is not met for u_3 and u_5, the sum $hits(\cup_{x \in S_{u_2}} C_x^h) \ge (hits(C^h_{l(u_3)}) + hits(C^h_{u_3})) + (hits(C^h_{l(u_5)}) + hits(C^h_{u_5})) > \frac{m}{2^{k-h}} + \frac{m}{2^{k-h}} = \frac{m}{2^{k-h-1}}$, where $l(u_3)$ and $l(u_5)$ are the predecessors of u_3 and u_5 on height h. Note that it is possible that $l(u_3)$ and $l(u_5)$ would be the same as u_2 and u_4 respectively. This means that u_2 satisfies the ascent condition, which contradicts Lemma 1.

Note that we considered four objects since u_1 is an object of height greater than h. ◀

Since only the leftmost object can be promoted, the backward path coincides with the forward path. Thus, the following lemma trivially holds.

▶ **Lemma 4.** *During the backward pass, in each sub-list h, at most four objects are visited that do not satisfy the descent condition.*

▶ **Theorem 5.** *If d descents occur when accessing object u, the sum of the lengths of the forward and backward paths is at most $2d + 8y$, where $y = 3 + \log \frac{m}{sh_u}$.*

Proof. Each object satisfying the descent condition is passed over twice, once in the forward and again in the backward pass. According to Lemma 2, there are at most y sub-lists that are visited during either passes. Excluding the descended objects, the total length of the forward path, according to Lemma 3 is $4y$. Lemma 4 gives the same result for the backward path. Hence, the total length is $2d + 8y$ which is the desired result. ◀

Asymptotic analysis. We can now finally state our main analytic result.

▶ **Theorem 6.** *The hit-operations with argument u take amortized $O\left(\log \frac{M}{sh_u}\right)$ time, where M is the total number of hits to non-marked objects of the splay-list. At the same time, all other operations take amortized $O(\log M)$ time.*

Proof. We will prove the same bounds but with m instead of M. Please note that since we rebuild the splay-list is triggered when M becomes less than $\frac{m}{2}$, we can always assume that $M \geq \frac{m}{2}$ and, thus, the bounds with m and M differ only by a constant.

First, we deal with the splay-list expansion procedure: it adds only $O(1)$ amortized time to an operation. The expansion happens when m is equal to the power of two and costs $O(m)$. Since, from the last expansion we performed at least $\frac{m}{2}$ hits operations we can amortize the cost $O(m)$ against them. Note that each operation will be amortized against only once, thus the amortization increases the complexity of an operation only by $O(1)$.

Since the primitive operations such as following the list pointer, a promotion with the ascent check and a demotion with the descent check are all $O(1)$, the cost of an operation is in the order of the length of the traversed path. According to Theorem 5, the total length of the traversed path during an operation is $2 \cdot d + 8 \cdot y$ where d is the number of vertices to demote and y is the number of traversed layers: if the object u was found y is equal to $O\left(\log \frac{m}{sh_u}\right)$, otherwise, it is equal to $\log m$, the height of the splay-list.

Note that the number of promotions per operation cannot exceed the number of passed levels y, since only one object can satisfy the ascent condition per level. At the same time, the total number of demotions across all operations, i.e., the sum of all d terms, cannot exceed the total number of promotions since we insert the objects on the lowest level. Thus, the amortized time of the operation can be bounded by $O(\text{number of levels passed})$ which is equal to what we required.

The amortized bound for `delete` operation needs some additional care. The operation can be split into two parts: 1) find the object in the splay-list, mark it as deleted and adjust the path; 2) the reconstruction part when the object is physically deleted. The first part is performed in $O(\log \frac{m}{sh_u})$ as shown above. For the second part, we perform the reconstruction only when the number of hits on objects marked for deletion $m - M$ exceeds the number of hits on all objects m, and, thus, $M \leq \frac{m}{2}$. The reconstruction is performed in $O(M) = O(m)$ time as explained in *Efficient Rebuild* part. Thus we can amortize this $O(m)$ to hits operations performed on logically deleted items. Since there were $O(m - M) = O(m)$ such operations, the amortization "increases" their complexities only on some constant and only once, since after the reconstruction the corresponding objects are going to be deleted physically. ◀

▶ Remark 7. For example, if all our operations were successful `contains`, then the asymptotics for `contains(`u`)` will be $O(\log \frac{m}{sh_u})$ where m is the total number of operations performed.

Furthermore, under the same load we can prove the static optimality property [16]. Let $m_i \leq m$ be the total number of operations when we executed i-th operation on u, then the

total time spent on operations with argument u is $O\left(\sum_{i=1}^{sh_u} \log \frac{m_i}{i}\right) = O\left(\sum_{i=1}^{sh_u} \log \frac{m}{i}\right)$ which by Lemma 3 from [1] is equal to $O(sh_i + sh_i \cdot \log \frac{m}{sh_i})$. This is exactly the static optimality property.

4 Relaxed Rebalancing

If we build the straightforward concurrent implementation on top of the sequential implementation described in the previous section, it will obviously suffer in terms of performance since each operation (either `contains`, `insert` or `delete`) must take locks on the whole path to update hits counters. This is not a reasonable approach, especially in the case of the frequent `contains` operation. Luckily for us, `contains` can be split into two phases: the *search* phase, which traverses the splay-list and is lock-free, and the *balancing* phase, which updates the counters and maintains ascent and descent conditions.

A straightforward heuristic is to perform rebalancing infrequently – for example, only once in c operations. For this, we propose that the operation perform the update of the global operation counter m and per-object hits counter sh_u only with a fixed probability $1/c$. Conveniently, if the operation does not perform the global operation counter update and the balancing, the counters will not change and, so, all the conditions will still be satisfied. The only remaining question is how much this relaxation will affect the data structure's guarantees. The next result characterizes the effects of this relaxation.

▶ **Theorem 8.** *Fix a parameter $c \geq 1$. In the relaxed sequential algorithm where operation updates hits counters and performs balancing with probability $\frac{1}{c}$, the hit-operation takes $O\left(c \cdot \log \frac{m}{sh_u}\right)$ expected amortized time, where m is the total number of hit-operations performed on all objects in splay-list up to the current point in the execution.*

Proof. The theoretical analysis above (Theorems 5 and 6) is based on the assumption that the algorithm maintains exact values of the counters m and sh_u – the total number of hit-operations performed to the existing objects and the current number of hit-operations to u. However, given the relaxation, the algorithm can no longer rely on m and sh_u since they are now updated only with probability c. We denote by m' and sh'_u the relaxed versions of the real counters m and sh_u.

The proof consists of two parts. First, we show that the amortized complexity of hits operation to u is equal to $O\left(c \cdot \log \frac{m'}{sh'_u}\right)$ in expectation. Secondly, we show that the approximate counters behave well, i.e., $\mathbb{E}\left[\log \frac{m'}{sh'_u}\right] = O\left(\log \frac{m}{sh_u}\right)$. Bringing these two together yields that the amortized complexity of hits operations is $O\left(c \cdot \log \frac{m}{sh_u}\right)$ in expectation.

The first part is proven similarly to Theorem 6. We start with the statement that follows from Theorem 5: the complexity of any contains operation is equal to $2d + 8y$ where d is the number of objects satisfying the descent condition and $y = 3 + \log \frac{m'}{sh'_u}$. Obviously, we cannot use the same argument as in Theorem 6 since now d is not equal to the number of descents: the objects which satisfy the descent condition are descended only with probability $\frac{1}{c}$. Thus, we have to bound the sum of d by the total number of descents.

Consider some object x that satisfies the descent condition, i.e. it is counted in d term of the complexity. Then x will either be descended, or will not satisfy the descent condition after c operations passing through it in expectation. Mathematically, the event that x is descended follows an exponential distribution with success (demotion) probability $\frac{1}{c}$. Hence, the expected number of operations before x descends is c.

This means that the object x will be counted in terms of type d no more than c times in expectation. By that, the total complexity of all operations is equal to the sum of $8y$ terms plus $2c$ times the number of descents. Since the number of descents cannot exceed the number of ascents, which in turn cannot exceed the sum of the y terms, the total complexity does not exceed the sum of $10 \cdot c \cdot y$ terms. Finally, this means that the amortized complexity complexity of hits operation is $O(c \cdot y) = O\left(c \cdot \log \frac{m'}{sh'_u}\right)$ in expectation.

Next, we prove the second main claim, i.e., that

$$\mathbb{E}\left(\log \frac{m'}{sh'_u}\right) = O\left(\log \frac{m}{sh_u}\right).$$

Note that the relaxed counters m' and sh'_u are Binomial random variables with probability parameter $p = \frac{1}{c}$, and number of trials m and sh_u, respectively.

To avoid issues with taking the logarithm of zero, let us bound $\mathbb{E}\left(\log \frac{m'+1}{sh'_u+1}\right)$, which induces only a constant offset. We have:

$$\begin{aligned}\mathbb{E}\left[\log \frac{m'+1}{sh'_u+1}\right] &= \mathbb{E}\left[\log(m'+1)\right] - \mathbb{E}\left[\log(sh'_u+1)\right] \\ &\underset{\text{Jensen}}{\leq} \log(\mathbb{E}m'+1) - \mathbb{E}\log(sh'_u+1) = \log(mp+1) - \mathbb{E}\log(sh'_u+1).\end{aligned}$$

The next step in our argument will be to lower bound $\mathbb{E}\log(sh'_u+1)$. For this, we can use the observation that $sh'_u \sim Bin_{sh_u,p}$, the Chernoff bound, and a careful derivation to obtain the following result, whose proof is left to the full version [2].

▷ Claim 9. If $X \sim Bin_{n,p}$ and $np \geq 3n^{2/3}$ then $\mathbb{E}\left[\log(X+1)\right] \geq \log np - 4$.

Based on this, we obtain $\log(mp+1) - \mathbb{E}[\log(sh'_u+1)] \leq \log(mp+1) - \log(sh_u \cdot p) + 4 \leq \log \frac{m}{sh_u} + 5$.

However, this bound works only for the case when $sh_u \cdot p \geq 3 \cdot (sh_u)^{2/3}$. Consider the opposite: $sh_u \leq \frac{27}{p^3}$. Then, $\mathbb{E}[\log(sh'_u+1)] \geq 0 \geq \log sh_u - \log \frac{27}{p^3}$. Note that the last term is constant, so we can conclude that $\mathbb{E}[\log \frac{m'+1}{sh'_u+1}] \leq \log \frac{m}{sh_u} + C$. This matches our initial claim that $\mathbb{E}[\log \frac{m'+1}{sh'_u+1}] = O(\log \frac{m}{sh_u})$. ◀

5 The Concurrent Splay-List

Overview. In this section we describe on how to implement scalable lock-based implementation of the splay-list described in the previous section. The first idea that comes to the mind is to implement the operations as in Lazy Skip-list [14]: we traverse the data structure in a lock-free manner in the search of x and fill the array of predecessors of x on each level; if x is not found then the operation stops; otherwise, we try to lock all the stored predecessors; if some of them are no longer the predecessors of x we find the real ones or, if not possible, we restart the operation; when all the predecessors are locked we can traverse and modify the backwards path using the presented sequential algorithm without being interleaved. When the total number of operations m becomes a power of two, we have to increase the height of the splay-list by one: in a straightforward manner, we have to take the lock on the whole data structure and then rebuild it.

There are several major issues with the straightforward implementation described above. At first, the *balancing* part of the operation is too coarse-grained – there are a lot of locks to be taken and, for example, the lock on the topmost level forces the operations to serialize. The second is that the list expansion by freezing the data structure and the following rebuild when m exceeds some power of two is very costly.

Relaxed and Forward Rebalancing. The first problem can be fixed in two steps. The most important one is to relax guarantees and perform *rebalancing* only periodically, for example, with probability $\frac{1}{c}$ for each operation. Of course, this relaxation will affect the bounds – please see Section 4 for the proofs. However, this relaxation is not sufficient, since we cannot relax the balancing phase of `insert`(u) which physically links an object. All these `insert` functions are going to be serialized due to the lock on the topmost level. Note that without further improvements we cannot avoid taking locks on each predecessor of x, since we have to update their counters. We would like to have more fine-grained implementation. However, our current sequential algorithm does not allow this, since it updates the path only backwards and, thus, needs the whole path to be locked. To address this issue, we introduce a different variant of our algorithm, which does rebalancing *on the forward traversal*.

We briefly describe how this *forward-pass algorithm* works. We maintain the basic structure of the algorithm. At first, we make a lock-free traversal to find x. Only after this we perform forward traversal with rebalancing if necessary. We traverse the splay-list in the search of x, and suppose that we are now at the last node v on the level h which precedes x. The only node on level $h-1$ which can be ascended is v's successor on that level, node u: we check the ascent condition on u or, in other words, compare $\sum_{w \in S_u} hits(C_w^{h-1}) = hits_v^h - hits_v^{h-1}$ with $\frac{m}{2^{k-h}}$, and promote u, if necessary. Then, we iterate through all the nodes on the level $h-1$ while the keys are less than x: if the node satisfies the descent condition, we demote it. Note that the complexity bounds for that algorithm are the same as for the previous one and can be proven exactly the same way (see Theorem 6).

The main improvement brought by this forward-pass algorithm is that now the locks can be taken in a hand-over-hand manner: take a lock on the highest level h and update everything on level $h-1$; take a lock on level $h-1$, release the lock on level h and update everything on level $h-2$; take a lock on level $h-2$, release the lock on level $h-1$ and update everything on level $h-3$; and so on. By this locking pattern, the balancing part of different operations is performed in a sequential manner: an operation cannot overtake the previous one and, thus, the *hits* counters cannot be updated asynchronously. However, at the same time we reduce contention: locks are not taken for the whole duration of the operation.

Lazy Expansion. The expansion issue is resolved in a lazy manner. The splay-list maintains the counter *zeroLevel* which represents the current lowest level. When m reaches the next power of two, *zeroLevel* is decremented, i.e., we need one more level. (To be more precise, we decrement *zeroLevel* also lazily: we do this only when some node is going to be demoted from the current lowest level.) Each node is allocated with an array of *next* pointers with length 64 (as discussed, the height 64 allows us to perform 2^{64} operations which is more than enough) and maintains the lowest level to which the node belonged during the last traverse. When we traverse a node and it appears to have the lowest level higher than *zeroLevel*, we update its lowest level and fill the necessary cells of *next* pointers. By doing that we make a lazy expansion of splay-list and we do not have to freeze whole data structure to rebuild. For the pseudo-code of the splay-list, we refer to the full version of the paper [2].

Table 1 Operations per second and average length of a path on $10^5-90-10$ workload.

$10^5-90-10$	Skip-list	SL $p=1$	SL $p=\frac{1}{2}$	SL $p=\frac{1}{5}$	SL $p=\frac{1}{10}$	SL $p=\frac{1}{100}$	SL $p=\frac{1}{1000}$
ops/sec	2874600.0	0.60x	0.78x	1.00x	1.10x	1.12x	1.02x
length	30.81	23.06	23.07	23.08	23.13	23.75	25.06
		CBTree $p=1$	CBTree $p=\frac{1}{2}$	CBTree $p=\frac{1}{5}$	CBTree $p=\frac{1}{10}$	CBTree $p=\frac{1}{100}$	CBTree $p=\frac{1}{1000}$
ops/secs		1.15x	1.36x	1.59x	1.71x	1.71x	1.52x
length		9.13	9.14	9.15	9.17	9.37	9.81

Table 2 Operations per second and average length of a path on 10^5-95-5 workload.

10^5-95-5	Skip-list	SL $p=1$	SL $p=\frac{1}{2}$	SL $p=\frac{1}{5}$	SL $p=\frac{1}{10}$	SL $p=\frac{1}{100}$	SL $p=\frac{1}{1000}$
ops/sec	2844520.0	0.69x	0.93x	1.21x	1.34x	1.39x	1.17x
length	30.84	21.62	21.63	21.65	21.70	22.33	24.46
		CBTree $p=1$	CBTree $p=\frac{1}{2}$	CBTree $p=\frac{1}{5}$	CBTree $p=\frac{1}{10}$	CBTree $p=\frac{1}{100}$	CBTree $p=\frac{1}{1000}$
ops/secs		1.33x	1.61x	1.90x	2.04x	2.09x	1.79x
length		8.61	8.61	8.62	8.65	8.90	9.58

Table 3 Operations per second and average length of a path on 10^5-99-1 workload.

10^5-99-1	Skip-list	SL $p=1$	SL $p=\frac{1}{2}$	SL $p=\frac{1}{5}$	SL $p=\frac{1}{10}$	SL $p=\frac{1}{100}$	SL $p=\frac{1}{1000}$
ops/sec	3559320.0	0.85x	1.19x	1.65x	1.89x	2.01x	1.64x
length	31.00	17.13	17.16	17.23	17.30	18.59	21.00
		CBTree $p=1$	CBTree $p=\frac{1}{2}$	CBTree $p=\frac{1}{5}$	CBTree $p=\frac{1}{10}$	CBTree $p=\frac{1}{100}$	CBTree $p=\frac{1}{1000}$
ops/secs		1.37x	1.72x	2.06x	2.25x	2.36x	2.04x
length		7.25	7.23	7.26	7.28	7.52	8.53

The following Theorem trivially holds due to the specificity of skip-list: if an operation reaches a sub-list of lower height than its target element it will still find it, if it is present.

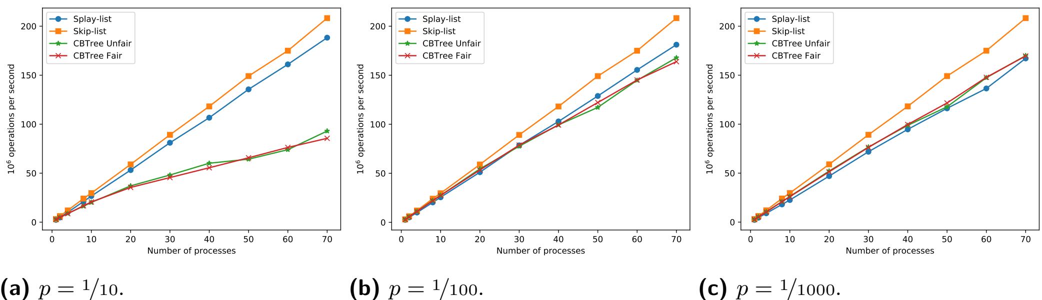
▶ **Theorem 10.** *The presented concurrent splay-list algorithm is linearizable.*

6 Experimental Evaluation

Environment and Methodology. We evaluate algorithms on a 4-socket Intel Xeon Gold 6150 2.7 GHz server with 18 threads per socket. The code is written in C++ and was compiled by MinGW GCC 6.3.0 compiler with `-O2` optimizations. Each experiment was performed 10 times and all the values presented are averages. The code is available at `https://github.com/demon1999/splaylist`.

Workloads and Parameters. Due to space constraints, our experiments in this section consider read-only workloads with unbalanced access distribution, which are the focus of our paper. We also execute uniform and read-write workloads, whose results we present in the full version [2]. In our experiments, we describe a family of workloads by $n-x-y$, which should be read as: given n keys, x% of the `contains` are performed on y% of the keys. More precisely, we first populate the splay-list with n keys and randomly choose a set of "popular" keys S of size $y \cdot n$. We then start T threads, each of which iteratively picks an element and performs the `contains` operation, for 10 seconds. With probability x we choose a random element from S, otherwise, we choose an element outside of S uniformly at random.

For our experiments, we choose the following workloads: $10^5-90-10$, 10^5-95-5 and 10^5-99-1. That is, 90%, 95%, and 99% of the operations go into 10%, 5%, and 1% of the keys, respectively. Further, we vary the *balancing rate/probability*, which we denote by p: this is the probability that a given operation will update hit counters and perform rebalancing. In full version [2], we also examine uniform and Zipf distributions.



(a) $p = 1/10$. **(b)** $p = 1/100$. **(c)** $p = 1/1000$.

Figure 3 Concurrent throughput for $10^5 - 90 - 10$ workload.

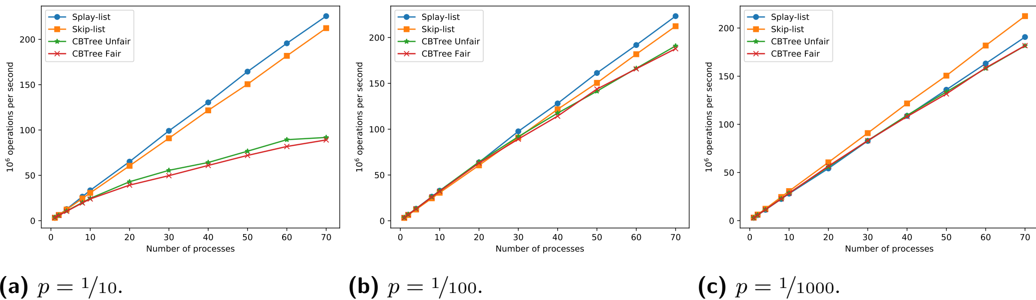
Goals and Baselines. We aim to determine whether 1) the splay-list can improve over the throughput of the baseline skip-list by successfully leveraging the skewed access distribution; 2) whether it scales, and what is the impact of update rates and number of threads; and, finally, 3) whether it can be competitive with the CBTree data structure in sequential and concurrent scenarios.

Sequential evaluation. In the first round of experiments, we compare how the single-threaded splay-list performs under the chosen workloads. We execute it with different settings of p, the probability of adjustment, taking values 1, $\frac{1}{2}$, $\frac{1}{5}$, $\frac{1}{10}$, $\frac{1}{100}$ and $\frac{1}{1000}$. We compare against the sequential skip-list and CB-Tree. We measure two values: the number of operations per second and the average length of the path traversed. The results are presented in Tables 1 – 3 (Splay-List is abbreviated SL). For readability, throughput results are presented relative to the skip-list baseline.

Relative to the skip-list, the first observation is that, for high update rates (1 through 1/5), the splay-list predictably only matches or even loses performance. However, this trend improves as we reduce the update rate, and, more significantly, as we increase the access rate imbalance: for $99 - 1$, the sequential splay-list obtains a throughput improvement of $2\times$. This improvement directly correlates with the length of the access path (see third row). At the same time, notice the negative impact of very low update rates (last column), as the average path length increases, which leads to higher average latency and decreased throughput. We empirically found the best update rate to be around 1/100, trading off latency with per-operation cost.

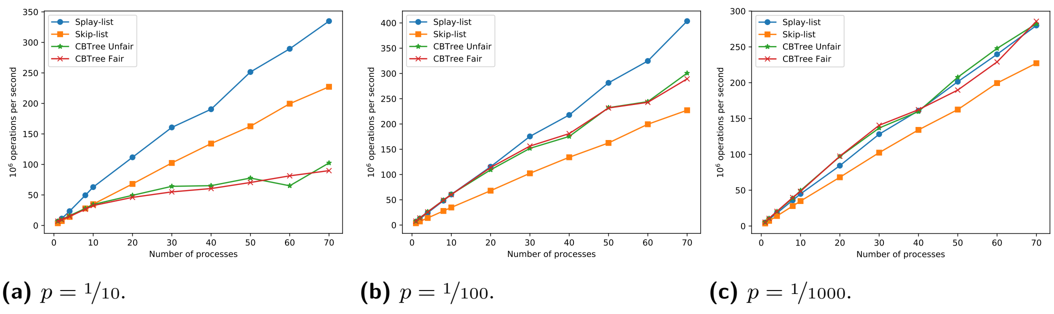
Relative to the sequential CBTree, we notice that the splay-list generally yields lower throughput. This is due to two factors: 1) the CBTree is able to yield shorter access paths, due to its structure and constants; 2) the tree tends to have better cache behavior relative to the skip-list backbone, i.e., more nodes can be stored in cache. Given the large difference in terms of average path length, it may seem surprising that the splay-list is able to provide close performance. This is because of the caching mechanism: as long as the path length for popular elements is short enough so that all the topmost nodes are mostly in cache, the average path length is not critical. We will revisit this observation in the concurrent case.

Concurrent evaluation. Next, we analyze concurrent performance. Unfortunately, the original implementation of the CBTree is not available, and we therefore re-implemented it in our framework. Here, we make an important distinction relative to usage: the authors of the CBTree paper propose to use a single thread to perform all the rebalancing. However, this approach is not standard, as in practice, updates could come at different threads.



(a) $p = 1/10$. **(b)** $p = 1/100$. **(c)** $p = 1/1000$.

Figure 4 Concurrent throughput for $10^5 - 95 - 5$ workload.



(a) $p = 1/10$. **(b)** $p = 1/100$. **(c)** $p = 1/1000$.

Figure 5 Concurrent throughput for $10^5 - 99 - 1$ workload.

Therefore, we implement two versions of the CBTree, one in which updates are performed by a single thread (CBTree-Unfair), and one in which updates can be performed by every thread with some probability (CBTree-Fair). In both cases, synchronization between readers and writers is performed via an efficient readers-writers lock [9], which prevents concurrent updates to the tree. We note that in theory we could further optimize the CBTree to allow fully-concurrent updates via fine-grained synchronization. However, 1) this would require a significant re-working of their algorithm; 2) as we will see below, this would not change results significantly.

Our experiments, presented in Figures 3, 4, and 5, analyze the performance of the splay-list relative to standard skip-list and the CBTree across different workloads (one per figure), different update rates (one per panel), and thread counts (X axis).

Examining the figures, first notice the relatively good scalability of the splay-list under all chosen update rates and workloads. By contrast, the CBTree scales well for moderately skewed workloads and low update rates, but performance decays for skewed workloads and high update rates (see for instance Figure 5(a)). We note that, in the former case the CBTree matches the performance of the splay-list in the low-update case (see Figure 3(c)), but its performance can decrease significantly if the update rates are reasonably high ($p = 1/100$). We further note the limited impact of whether we consider the fair or unfair variant of the CBTree (although the Unfair variant usually performs better).

These results may appear surprising given that the splay-list generally has longer access paths. However, it benefits significantly from the fact that it allows additional concurrency, and that the caching mechanism serves to hide some of its additional access cost. Our intuition here is that one critical measure is which fraction of the "popular" part of the data structure fits into the cache. This suggests that the splay-list can be practically competitive relative to the CBTree on a subset of workloads.

Additional Experiments. Further experiments in full version [2] examine: 1) the overheads in the uniform access case, 2) performance for a Zipf access distribution; 3) performance under moderate insert/delete rates. We also examine performance over longer runs, as well as the correlation between element height in the list and its "popularity."

7 Discussion

We revisited the question of efficient self-adjusting concurrent data structures, and presented the first instance of a self-adjusting concurrent skip-list, addressing an open problem posed in CBTree paper [1]. Our design ensures static optimality, and has an arguably simple structure and implementation, which allows for additional concurrency and good performance under skewed access. In addition, it is the first design to provide guarantees under approximate access counts, required for good practical behavior. In future work, we plan to expand the experimental evaluation to include a range of real-world workloads, and to prove the guarantees under concurrent access.

References

1. Yehuda Afek, Haim Kaplan, Boris Korenfeld, Adam Morrison, and Robert E. Tarjan. Cbtree: A practical concurrent self-adjusting search tree. In *Proceedings of the 26th International Conference on Distributed Computing*, DISC'12, pages 1–15, Berlin, Heidelberg, 2012. Springer-Verlag. doi:10.1007/978-3-642-33651-5_1.
2. Vitaly Aksenov, Dan Alistarh, Alexandra Drozdova, and Amirkeivan Mohtashami. The splay-list: A distribution-adaptive concurrent skip-list. *arXiv preprint arXiv:2008.01009*, 2020.
3. Maya Arbel-Raviv, Trevor Brown, and Adam Morrison. Getting to the root of concurrent binary search tree performance. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 295–306, Boston, MA, July 2018. USENIX Association. URL: https://www.usenix.org/conference/atc18/presentation/arbel-raviv.
4. Amitabha Bagchi, Adam L Buchsbaum, and Michael T Goodrich. Biased skip lists. *Algorithmica*, 42(1):31–48, 2005.
5. Prosenjit Bose, Karim Douïeb, and Stefan Langerman. Dynamic optimality for skip lists and b-trees. In *Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1106–1114, 2008.
6. Trevor Brown. *Techniques for Constructing Efficient Data Structures*. PhD thesis, PhD thesis, University of Toronto, 2017.
7. Valentina Ciriani, Paolo Ferragina, Fabrizio Luccio, and Shanmugavelayutham Muthukrishnan. Static optimality theorem for external memory string access. In *The 43rd Annual IEEE Symposium on Foundations of Computer Science, 2002. Proceedings.*, pages 219–227. IEEE, 2002.
8. Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.
9. Andreia Correia and Pedro Ramalhete. Scalable reader-writer lock in c++1x. http://concurrencyfreaks.blogspot.com/2015/01/scalable-reader-writer-lock-in-c1x.html, 2015.
10. Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '10, pages 131–140, New York, NY, USA, 2010. ACM. doi:10.1145/1835698.1835736.
11. Keir Fraser. Practical lock-freedom. Technical Report UCAM-CL-TR-579, University of Cambridge, Computer Laboratory, February 2004.

12 Keir Fraser. *Practical lock-freedom.* PhD thesis, PhD thesis, Cambridge University Computer Laboratory, 2003. Also available as Technical Report UCAM-CL-TR-579, 2004.

13 Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. A simple optimistic skiplist algorithm. In *Proceedings of the 14th international conference on Structural information and communication complexity*, SIROCCO'07, pages 124–138, Berlin, Heidelberg, 2007. Springer-Verlag.

14 Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

15 John Iacono. Alternatives to splay trees with o (log n) worst-case access times. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 516–522. Society for Industrial and Applied Mathematics, 2001.

16 Donald Ervin Knuth. *The art of computer programming*, volume 3. Pearson Education, 1997.

17 Doug Lea, 2007. URL: `http://java.sun.com/javase/6/docs/api/java/util/concurrent/ConcurrentSkipListMap.html`.

18 Charles Martel. Self-adjusting multi-way search trees. *Information Processing Letters*, 38(3):135–141, 1991.

19 Maged M Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 73–82. ACM, 2002.

20 Aravind Natarajan and Neeraj Mittal. Fast concurrent lock-free binary search trees. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 317–328, New York, NY, USA, 2014. ACM. `doi:10.1145/2555243.2555256`.

21 Meikel Poess and Chris Floyd. New tpc benchmarks for decision support and web commerce. *ACM Sigmod Record*, 29(4):64–71, 2000.

22 William Pugh. Concurrent maintenance of skip lists, 1998.

23 Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM (JACM)*, 32(3):652–686, 1985.

Efficient Multi-Word Compare and Swap

Rachid Guerraoui
EPFL, Lausanne, Switzerland
rachid.guerraoui@epfl.ch

Alex Kogan
Oracle Labs, Burlington, MA, USA
alex.kogan@oracle.com

Virendra J. Marathe
Oracle Labs, Burlinton, MA, USA
virendra.marathe@oracle.com

Igor Zablotchi[1]
EPFL, Lausanne, Switzerland
igor.zablotchi@epfl.ch

Abstract

Atomic lock-free multi-word compare-and-swap (MCAS) is a powerful tool for designing concurrent algorithms. Yet, its widespread usage has been limited because lock-free implementations of MCAS make heavy use of expensive compare-and-swap (CAS) instructions. Existing MCAS implementations indeed use at least $2k + 1$ CASes per k-CAS. This leads to the natural desire to minimize the number of CASes required to implement MCAS.

We first prove in this paper that it is impossible to "pack" the information required to perform a k-word CAS (k-CAS) in less than k locations to be CASed. Then we present the first algorithm that requires $k + 1$ CASes per call to k-CAS in the common uncontended case. We implement our algorithm and show that it outperforms a state-of-the-art baseline in a variety of benchmarks in most considered workloads. We also present a durably linearizable (persistent memory friendly) version of our MCAS algorithm using only 2 persistence fences per call, while still only requiring $k + 1$ CASes per k-CAS.

2012 ACM Subject Classification Theory of computation → Concurrent algorithms

Keywords and phrases lock-free, multi-word compare-and-swap, persistent memory

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.4

Related Version `https://arxiv.org/abs/2008.02527`

Funding This work has been supported in part by the European Research Council (ERC) Grant 339539 (AOC).

1 Introduction

Compare-and-swap (CAS) is a foundational primitive used pervasively in concurrent algorithms on shared memory systems. In particular, it is used extensively in *lock-free* algorithms, which avoid the pitfalls of blocking synchronization (e.g., that employs locks) and typically deliver more scalable performance on multicore systems. CAS conditionally updates a memory word such that a new value is written if and only if the old value in that word matches some expected value. CAS has been shown to be universal, and thus can implement any shared object in a non-blocking manner [32]. This primitive (or the similar load-linked/store-conditional (LL/SC)) is nowadays provided by nearly every modern architecture.

[1] This work was done when the author was an intern at Oracle Labs.

© Rachid Guerraoui, Alex Kogan, Virendra J. Marathe, and Igor Zablotchi;
licensed under Creative Commons License CC-BY
34th International Symposium on Distributed Computing (DISC 2020).
Editor: Hagit Attiya; Article No. 4; pp. 4:1–4:19
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

CAS does have an inherent limitation: it operates on a single word. However, many concurrent algorithms require atomic modification of multiple words, thus introducing significant complexity (and overheads) to get around the 1-word restriction of CAS [9, 17, 23, 24, 39, 44]. As a way to address the 1-word limitation, the research community suggested a natural extension of CAS to multiple words – an atomic multi-word compare-and-swap (MCAS). MCAS has been extensively investigated over the last two decades [4, 5, 17, 23, 24, 31, 32, 43, 51]. Arguably, this work partly led to the advent of the enormous wave of Transactional Memory (TM) research [29, 30, 34]. In fact, MCAS can be considered a special case of TM. While MCAS is not a silver bullet for concurrent programming [19, 33], the extensive body of literature demonstrates that the task of designing concurrent algorithms becomes much easier with MCAS. Not surprisingly, there has been a resurgence of interest in MCAS in the context of persistent memory, where the persistent variant of MCAS (PMCAS) serves as a building block for highly concurrent data structures, such as skip lists and B+-trees [6, 53], managed in persistent memory.

Existing lock-free MCAS constructions typically make heavy use of CAS instructions [4, 31, 43], requiring between 2 and 4 CASes per word modified by MCAS. That resulting cost is high: CASes may cost up to 3.2× times more cycles than load or store instructions [16]. Naturally, algorithm designers aim to minimize the number of CASes in their MCAS implementations.

Toward this goal, it may be tempting to try to "pack" the information needed to perform the MCAS in fewer than k memory words and perform CAS only on those words. We show in this paper that this is impossible. While this result might not be surprising, the proof is not trivial, and is done in two steps. First, we show through a bivalency argument that lock-free MCAS calls with non-disjoint sets of arguments must perform CAS on non-disjoint sets of memory locations, or violate linearizability. Building on this first result, we then show that any lock-free, disjoint-access-parallel k-word MCAS implementation admits an execution in which some call to MCAS must perform CAS on at least k different locations. (Our impossibility result focuses on *disjoint-access-parallel* (DAP) algorithms, in which MCAS operations on disjoint sets of words do no interfere with each other. DAP is a desirable property of scalable concurrent algorithms [37].)

We also show, however, in the paper that MCAS can be "efficient". We present the first MCAS algorithm that requires $k+1$ CAS instructions per call to k-CAS (in the common uncontended case). Furthermore, our construction has the desirable property that reads do not perform any writes to shared memory (unless they encounter an ongoing MCAS operation). This is to be contrasted with existing MCAS constructions (in which read operations do not write) that use at least $3k+1$ CASes per k-CAS. Furthermore, we extend our MCAS construction to work with persistent memory (PM). The extension does not change the number of CASes and requires only 2 persistence fences per call (in the common uncontended case), comparing favorably to the prior work that employs $5k+1$ CASes and $2k+1$ fences [53].

Most previous MCAS constructions follow a multi-phase approach to perform a k-CAS operation *op*. In the first (*locking*) phase, *op* "locks" its designated memory locations one by one by replacing the current value in those locations with a pointer to a *descriptor* object. This descriptor contains all the information necessary to complete *op* by the invoking thread or (potentially) by a helper thread. In the second (*status-change*) phase, *op* changes a status flag in the descriptor to indicate successful (or unsuccessful) completion. In the third (*unlocking*) phase, *op* "unlocks" those designated memory locations, replacing pointers to its descriptor with new or old values, depending on whether *op* has succeeded or failed.

In order to obtain lower complexity, our algorithm makes two crucial observations concerning this unlocking phase. First, this phase can be deferred off the critical path with no impact on correctness. In our algorithm, once an MCAS operation completes, its descriptor is left in place until a later time. The unlocking is performed later, either by another MCAS operation locking the same memory location (and thus effectively eliminating the cost of unlocking for *op*) or during the memory reclamation of operation descriptors. (We describe a delayed memory reclamation scheme that employs epochs and amortizes the cost of reclamation across multiple operations.)

Our second, and perhaps more surprising, observation is that deferring the unlocking phase allows the *locking* phase to be implemented more efficiently. In order to avoid the ABA problem, many existing algorithms require extra complexity in the locking phase. For instance, the well-known Harris et al. [31] algorithm uses the atomic *restricted double-compare single-swap* (*RDCSS*) primitive (that requires at least 2 CASes per call) to conditionally lock a word, provided that the current operation was not completed by a helping thread. Naively performing the locking phase using CAS instead of RDCSS would make the Harris et al. algorithm prone to the ABA problem (we provide an example in the full version of our paper [25]). However, in our algorithm, we get ABA prevention "for free" by using a memory reclamation mechanism to perform the unlocking phase, because such mechanisms already need to protect against ABA in order to reclaim memory safely.

Deferring the unlocking phase allows us to come up with an elegant and, arguably, simple MCAS construction. Prior work shows, however, that the correctness of MCAS constructions should not be taken for granted: for instance, Feldman et al. [20] and Cepeda et al. [12] describe correctness pitfalls in MCAS implementations. Thus, we carefully prove the correctness of our construction. We also evaluate our construction empirically by comparing to a state-of-the-art MCAS implementation and showing superior performance in a variety of benchmarks (including a production quality B+-Tree [6]) in most considered scenarios.

We note that the delayed unlocking/cleanup introduces a trade-off between higher MCAS performance (due to fewer CASes per MCAS, which also leads to less slow-down due to less helping) and lower read performance (because of the extra level of indirection reads have to traverse when encountering a descriptor left in place after a completed MCAS). One may argue that it also increases the amount of memory consumed by the MCAS algorithm. Regarding the former, our evaluation shows that the benefits of the lower complexity overcome the drawbacks of indirection in all workloads that experience MCAS contention. Furthermore, we propose a simple optimization to mitigate the impact of indirection in reads. As for the latter, we note that much like any lock-free algorithm, the memory consumption of our construction can be tuned by performing memory reclamation more (or less) often.

The rest of the paper is organized as follows. In Section 2 we describe our model. In Section 3 we present our impossibility result. Sections 4 and 5 detail our MCAS algorithms for volatile and persistent memory. Section 6 elaborates our lazy memory reclamation scheme. Section 7 presents the results of our experimental evaluation. We review related work in Section 8 and conclude in Section 9. Due to space limitations, some content (proofs, additional performance results etc.) has been omitted and appears in the full version of this paper [25].

2 System Model

2.1 Volatile Memory

We assume a standard model of asynchronous shared memory [35], with basic atomic *read*, *write* and *compare-and-swap* (CAS) operations. The latter receives three arguments – an address, an expected value and a new value; it reads the value stored in the given address and if it is equal to the expected value, atomically stores the new value in the given address, returning the indication of success or failure.

Using those atomic operations, we implement an atomic MCAS operation with the following semantics. The MCAS operation receives an array of tuples, where each tuple contains an address, an expected value and a new value. For ease of presentation, we assume the size of the array is a known constant N. (In practice, the size of the array can be dynamic, and different for every MCAS operation.) The MCAS operation reads values stored in the given addresses, and if they all are equal to respective expected values, atomically writes new values to the corresponding address and returns an indication of success. Otherwise, if at least one read value is different from an expected one, the MCAS operation returns an indication of failure. We also provide a custom implementation of a read operation from a memory location that can be a target of an MCAS operation (which, in the most general case, can be any shared memory location).

Our MCAS implementation is *linearizable* [35]. This means, informally, that each (read or MCAS) operation appears to take effect instantaneously at some point in time in the interval during which the operation executes. In terms of progress, our MCAS implementation is *non-blocking*. That is, a lack of progress of any thread (e.g., due to the suspension or failure of that thread) does not prevent other threads from applying their operations. Furthermore, the MCAS implementation guarantees *lock-freedom*. That is, given a set of threads applying operations, it guarantees that, eventually, at least one of those threads will complete its operation.

Similar to many non-blocking algorithms, our design makes use of operation descriptors, which store information on existing MCAS operations, including the status of the operation and the array of tuples with addresses and values. We assume each word in the shared memory can contain either a regular value or a pointer to such a descriptor. A similar assumption has been made in prior work on MCAS [20, 31, 52, 53]. In practice, a single (e.g., least significant) bit can be used to distinguish between the two.

Initialization of the descriptor is done before invocation of the MCAS operation. We assume that all the addresses in the descriptor are sorted in a monotonic total order. This assumption is crucial for the liveness property of our algorithm, but can be easily lifted by explicitly sorting the array of tuples by corresponding addresses before an MCAS operation is executed.

2.2 Persistent Memory

We extend the model in Section 2.1 with standard assumptions about PM [13, 15, 22, 38]. We assume the system is equipped with persistent shared memory that can be accessed through the same set of atomic primitives (read, write and CAS). The system may also be equipped with DRAM to be used as transient storage. As in previous work [38], we assume that the overall system can crash at any time and possibly recover later. On such a full-system crash, we assume that the contents of persistent memory – but not those of processor caches, registers or volatile memory – are preserved. Moreover, threads that are

active at the time of the crash are assumed to be lost forever and replaced by new threads in case of recovery. After a full-system crash but before the system recovers and resumes normal execution, we assume a *recovery* routine may be executed, in order to bring persistent memory-resident objects to a consistent state. The recovery routine can be executed in a single thread, and thus it does not have to be thread-safe. Another full-system crash, however, may occur during the recovery routine.

As is standard practice [13, 15, 53], we assume that a-priori there is no guarantee on when and in what order cache lines are written back to persistent memory. We assume the existence of two primitives to enforce such write backs. The first primitive is `PERSISTENT_FLUSH(addr)`, which takes as argument a memory location and asynchronously writes the contents of that location to persistent memory. Multiple invocations of this primitive are not ordered with respect to each other and thus several flushes can proceed in parallel. Concrete examples of this primitive are `clflushopt` and `clwb` [36]. The second primitive is `PERSISTENT_FENCE()`, which stalls the CPU until any pending flushes are committed to persistent memory. A concrete example of this primitive is `sfence` [36]. LOCK-prefixed instructions such as CAS also act as persistent fences [36]. Since persistent flushes do not stall the CPU, whereas persistent fences do, the cost of writing to persistent memory is dominated by the latter instructions and we consider the cost of the former to be negligible.

Regarding initialization, we assume descriptor contents are made persistent before invocation of MCAS.

The safety criterion we use when working with persistent memory is durable linearizability [38]. Informally, an implementation of an object is durably linearizable if it is linearizable and has the following additional properties in case of a full-system crash and recovery: (1) all operations that completed before the crash are reflected in the post-recovery state and (2) if some operation *op* that was ongoing at the time of the crash is reflected in the post-recovery state, then so are all the operations on which *op* depends (i.e., operations whose effects *op* observed and thus need to be linearized before *op*).

3 Impossibility

In this section we show that any lock-free disjoint-access-parallel (DAP) implementation of MCAS requires at least one CAS per modified word. Consider a call to k-CAS($addr_1, \dots, addr_k$, [old and new values]). We call $addr_1, \dots, addr_k$ the *set of targets* of the call. We also define the *range* of the call in an execution E to be the set of locations on which CAS (single-word CAS) is performed, successfully or not, during the call in E. Intuitively, we say that an MCAS implementation is *DAP* if non-conflicting calls to k-CAS do not access the same memory locations; for the formal definition, see [37].

▶ **Definition 1** (Star Configuration). *We say that a set $\{c_0, \dots, c_\ell\}$ of calls to k-CAS are in a* star configuration *if (1) the sets of targets of c_0 and c_i are non-disjoint for all $i \in \{1, \dots, \ell\}$, and (2) the sets of targets of c_i and c_j are disjoint for all $i \neq j \in \{1, \dots, \ell\}$.*

An example of a star configuration for $\ell = k$ is the following set of calls $\mathcal{C} = \{c_0, \dots, c_k\}$, where we omit old and new values for ease of notation and we assume that addresses $a_i^{(j)}$ are all distinct:

- c_0: k-CAS$(a_1^{(0)}, \dots, a_k^{(0)})$
- c_1: k-CAS$(a_1^{(0)}, a_2^{(1)}, \dots, a_k^{(1)})$. Call c_1's set of targets intersects that of c_0 in $a_1^{(0)}$.
- c_i, $1 \leq i \leq k$: k-CAS$(a_1^{(i)}, \dots, a_i^{(0)}, \dots, a_k^{(i)})$. Call c_i's set of targets intersects that of c_0 in $a_i^{(0)}$ and is disjoint from the set of targets of c_j for all $j \neq i, j \neq 0$.

In this section, we assume without loss of generality that all calls in $\mathcal{C}$ have the correct old values for their target addresses and that each new value is distinct from its respective old value. Under these assumptions, in every execution it must be that either c_0 succeeds and all $c_1, \ldots, c_k$ fail, or that c_0 fails and all $c_1, \ldots, c_k$ succeed.

We say that a state S of an implementation $\mathcal{A}$ is c_0-valent with respect to (*wrt*) some subset $C \subseteq \mathcal{C}$ if, for any call $c_i \in C$, in any execution starting from S in which only c_0 and c_i take steps, c_0 succeeds. Similarly, we say that a state S is C-valent *wrt* c_0 if, for any call $c_i \in C$, in any execution starting from S in which only c_0 and c_i take steps, c_0 fails. We say that a state is univalent *wrt* c_0 and C if it is c_0-valent or C-valent; otherwise it is bivalent *wrt* c_0 and C. A state is critical *wrt* c_0 and C when (1) it is bivalent *wrt* c_0 and C and (2) if any process in $\{c_0\} \cup C$ takes a step, the state becomes univalent *wrt* c_0 and C.

Note that the initial state of $\mathcal{A}$ must be bivalent *wrt* c_0 and any non-empty subset of $\mathcal{S}$.

▶ **Lemma 2.** *Consider a lock-free implementation $\mathcal{A}$ of k-CAS and let $\mathcal{C} = \{c_0, \ldots, c_\ell\}$ be a star configuration of calls to k-CAS. Then there exists an execution E of $\mathcal{A}$ such that, for all $i \geq 1$, the ranges of c_0 and c_i in E are non-disjoint.*

Proof. We follow a bivalency proof structure. We construct an execution in which process p_i performs call c_i, $i \geq 0$. For ease of notation, we say that "call c_i takes a step" to mean "process p_i takes a step in its execution of c_i".

The execution proceeds in stages. In the first stage, as long as some call in $\mathcal{C}$ can take a step without making the state univalent *wrt* c_0 and any non-empty subset of $\mathcal{C}$, let that call take a step. If the execution runs forever, the implementation is not lock-free. Otherwise, the execution enters a state S where no such step is possible, which must be a critical state *wrt* c_0 and some subset $C_1 \subseteq \mathcal{C} \setminus \{c_0\}$. We choose C_1 to be maximal, i.e., state S is not critical *wrt* c_0 and any subset of $\mathcal{C} \setminus C_1$ (otherwise, add that subset to C_1).

We prove in Lemma 3 below that c_0 and all calls in C_1 are about to perform CAS on some common location l_1. We let c_0 perform that CAS step, bringing the protocol to state S'. By our choice of C_1 as maximal, S' must be bivalent *wrt* c_0 and any subset of $\mathcal{C} \setminus C_1$. The execution now enters the second stage, in which we let calls in $\mathcal{C} \setminus C_1$ take steps until they reach a critical state *wrt* c_0 and some subset $C_2 \subseteq \mathcal{C} \setminus C_1$. By induction, we can show that eventually c_0 will have reached critical points *wrt* all calls in $\mathcal{C}$. At the end of the execution, we resume each process in $\mathcal{C} \setminus c_0$ for one step; they were each about to perform a CAS step on some location on which c_0 has already performed a CAS step. Thus, in this execution, all calls in $\mathcal{C} \setminus c_0$ have performed a CAS on a common location with c_0. ◀

▶ **Lemma 3.** *Consider a lock-free implementation $\mathcal{A}$ of k-CAS and let $\mathcal{C} = \{c_0, \ldots, c_k\}$ be a star configuration of calls to k-CAS. If S is a critical state of $\mathcal{A}$ wrt c_0 and some subset $C \subseteq \mathcal{C}$, then in S, c_0 and all calls in C are about to perform a CAS step on a common location l.*

Proof. From S, we consider the next steps of c_0 and any $c_i \in C$:

Case 1 One of the calls is about to read; assume *wlog* it is c_0. Consider two possible scenarios. First scenario: c_i moves first and runs solo until it returns (c_i must succeed because c_i took the first step). Second scenario: c_0 moves first and reads, then c_i runs solo until it returns (c_i must fail because c_0 took the first step). But the two scenarios are indistinguishable to c_i, thus c_i must either succeed in both or fail in both, a contradiction.

Case 2 Both calls are about to write. In this case, they must be about to write to the same register r, otherwise their writes commute. First scenario: c_0 writes r, then c_i writes r, then c_i runs solo until it returns (c_i must fail since c_0 took the first step). Second

scenario: c_i writes r and then runs solo until it returns (c_i must succeed since c_i took the first step). But the two scenarios are indistinguishable to c_i, since its write to r obliterated any potential write by c_0 to r, so c_i must either succeed in both scenarios or fail in both; a contradiction.

Case 3 c_0 is about to CAS and c_i is about to write (or vice-versa). In this case, their operations must be to the same memory location r (otherwise they commute). First scenario: c_0 CASes r, then c_i writes to r and then runs solo until c_i returns (c_i must fail since c_0 took the first step). Second scenario: c_i writes to r and then runs solo until it returns (c_i must succeed since c_i took the first step). But the two scenarios are indistinguishable to c_i, since its write to r obliterated any preceding CAS by c_0 to r; thus c_i must either succeed in both scenarios or fail in both; a contradiction.

Case 4 Both calls are about to CAS. In this case, they must be about to CAS the same location, otherwise their CASes commute. ◀

▶ **Theorem 4.** *Consider a lock-free disjoint-access-parallel implementation $\mathcal{A}$ of k-CAS in a system with $n > k$ processes. Then there exists some execution E of $\mathcal{A}$ such that in E some call to k-CAS performs CAS on at least k locations.*

Proof. We prove the theorem by contradiction. We first assume that calls to k-CAS perform CAS on *exactly* $k-1$ locations and derive a contradiction; we later show how assuming that k-CAS performs CAS on *at most* $k-1$ locations also leads to a contradiction.

We construct an execution E in which two concurrent but non-contending k-CAS calls (i.e., two k-CAS calls with disjoint sets of targets) perform CAS on the same location, thus contradicting the disjoint-access-parallelism (DAP) property and proving the theorem.

Let $c_0, \ldots, c_k$ be $k+1$ calls to k-CAS in a star configuration. By Lemma 2, there exists an execution E of $\mathcal{A}$ such that, for all $i \geq 1$, the ranges of c_0 and c_i in E are non-disjoint.

Let $l_1, \ldots, l_{k-1}$ be the range of c_0. By Lemma 2, in E the range of c_1 must intersect that of c_0 in at least one location; assume *wlog* it is l_1. Furthermore, the range of c_2 must also intersect that of c_0 in at least one location; moreover, due to the DAP property, the intersection must contain some location other than l_1, since c_1 and c_2 have disjoint sets of targets. By induction, we can show that the range of each call $c_i, i \in \{1, 2, \ldots, k-1\}$ intersects the range of c_0 in l_i. However, the range of c_k must also intersect the range of c_0 in some location other than $l_1, \ldots, l_{k-1}$, due to the DAP property. We have reached a contradiction.

If we now assume that calls to k-CAS perform CAS on $k-1$ *or fewer* locations, then we also reach a similar contradiction as above. In fact, if some call c_i performs CAS on strictly fewer than $k-1$ locations, this may cause the contradiction to occur before call c_k, as c_i now has fewer locations to choose from in order intersect with the range of c_0 in some location that is not in the ranges of $c_1, \ldots, c_{i-1}$. ◀

4 Volatile MCAS with $k+1$ CAS

In this section we describe our MCAS construction for volatile memory. Our algorithm uses $k+1$ CAS operations in the common uncontended case, and does not involve cleaning up after completed MCAS operations. In Section 6 we describe a memory management scheme that can be used to clean up after completed MCAS operations as well as for reclaiming or reusing operation descriptors employed by the algorithm.

Listing 1 Data structures used by our algorithm

```
struct WordDescriptor {
    void* address;
    uintptr_t old;
    uintptr_t new;
    MCASDescriptor* parent; };

enum StatusType { ACTIVE, SUCCESSFUL, FAILED };

struct MCASDescriptor {
    StatusType status;
    size_t N;
    WordDescriptor words[N]; };
```

4.1 High-level Description

As is standard practice [28, 31, 52], our MCAS construction supports two operations: MCAS and `read`. Similarly to most MCAS algorithms [28, 31, 52], the MCAS operation uses operation descriptors that contain a set of addresses (the *target* addresses or words), and *old* and *new* values for each target address. In addition, each operation descriptor contains a *status* word indicating the status of the corresponding MCAS operation.

The MCAS operation proceeds in two stages. In the first stage, we attempt to install a pointer to the operator descriptor in each memory word targeted by the MCAS operation. If we succeed to install the pointer, we say that the target address is *owned* (or *locked*) by the descriptor. The first stage ends when all target addresses are owned by the descriptor, or if we find a target address with a value different from the expected one. In the second stage, we *finalize* the MCAS operation by atomically changing its status to indicate its success or failure, depending on whether the first stage was successful (i.e., all target addresses have been locked). The `read` operation returns the current value at an address, either by reading it directly from the target address or by reading the appropriate value from a descriptor of a completed MCAS operation installed in that address. If either MCAS or `read` encounter another MCAS in progress (e.g., when they attempt to read the current value in the target address), they first help that MCAS operation to complete.

4.2 Technical Details

Structures and Terminology. We describe the structures used by our algorithm and explain the terminology. Pseudocode for the structures is shown in Listing 1. An `MCASDescriptor` describes an MCAS operation. It contains a status field, which can be `ACTIVE`, `SUCCESSFUL` or `FAILED`, the number `N` of words targeted by the MCAS and an array of `WordDescriptors` for those words. These `WordDescriptors` are the *children* of the `MCASDescriptor`, who is their *parent*. We say that an `MCASDescriptor` (and the MCAS it describes) is *active* if its status is `ACTIVE` and *finalized* otherwise.

The `WordDescriptor` contains information related to a given word as target of an MCAS operation: the word's address in memory, its expected value and the new intended value. The `WordDescriptor` also contains a pointer to the descriptor of its parent MCAS operation. As described later, the pointer is used as an optimization for fast lookup of the status field in the `MCASDescriptor`, and can be eliminated.

Algorithm. Both MCAS and `read` operations rely on the auxiliary `readInternal` function shown in Listing 2. The `readInternal` function takes an address `addr` and an `MCASDescriptor self` (called the *current descriptor*) and returns a tuple. The tuple contains

Listing 2 The `readInternal` auxiliary function, used by our algorithm.

```
1  readInternal(void* addr, MCASDescriptor *self) {
2  retry_read:
3    val = *addr;
4    if (!isDescriptor(val)) then return <val,val>;
5    else { // found a descriptor
6      MCASDescriptor* parent = val->parent;
7      if (parent != self && parent->status == ACTIVE) {
8        MCAS(parent);
9        goto retry_read;
10     } else {
11       return parent->status == SUCCESSFUL ?
12           <val,val->new> : <val,val->old>; }  }  }
```

Listing 3 Our main algorithm. Commands in *italic* are related to memory reclamation (discussed in a later section).

```
13 read(void* address) {
14   epochStart();
15   <content, value> = readInternal(address, NULL);
16   epochEnd();
17   return value; }
18
19 MCAS(MCASDescriptor* desc) {
20   epochStart();
21   success = true;
22   for wordDesc in desc->words {
23 retry_word:
24     <content, value> = readInternal(wordDesc.address, desc);
25     // if this word already points to the right place, move on
26     if (content == &wordDesc) continue;
27     // if the expected value is different, the MCAS fails
28     if (value != wordDesc.old) { success = false; break;  }
29     if (desc->status != ACTIVE) break;
30     // try to install the pointer to my descriptor; if failed,  retry
31     if (!CAS(wordDesc.address, content, &wordDesc)) goto retry_word; }
32   if (CAS(&desc.status, ACTIVE, success ? SUCCESSFUL : FAILED)){
33     // if I finalized this descriptor, mark it for reclamation
34     retireForCleanup(desc); }
35   returnValue = (desc.status == SUCCESSFUL);
36   epochEnd();
37   return returnValue; }
```

two values (which might be identical), and, intuitively, represent the contents in the given (target) address and the actual value the former represents. More specifically, `readInternal` reads the content of the given `addr` (Line 3). If `addr` does not point to a descriptor (this is determined by the `isDescriptor` function; see below), the returned tuple contains two copies of the contents of `addr` (Line 4). If `addr` points to an active `WordDescriptor` whose parent is not the same as `self`, then `readInternal` helps the other (MCAS) operation to complete (Line 8) and then restarts (Line 9). Therefore, the role of the `self` pointer is to avoid an (MCAS) operation to help itself recursively. If `addr` points to a finalized descriptor, the tuple returned by `readInternal` contains the pointer to the descriptor and the final value, corresponding to the status of the descriptor (Line 12). Finally, if `addr` points to a descriptor whose parent is equal to `self`, then `readInternal` returns the pointer to that descriptor (Line 12; a value is also returned in the tuple in this case, but is disregarded; see below).

Listing 3 provides the pseudo-code for the `read` and MCAS operations. The pseudo-code includes extensions relevant to memory management (in *italics*), whose discussion is deferred to Section 6.

The `read` operation is simply a call to `readInternal` with a `self` equal to `null` as the current operation descriptor (Line 15).

The MCAS operation takes as argument an `MCASDescriptor` and returns a boolean indicating success or failure. As mentioned above, the operation proceeds in two stages. In the first stage, MCAS attempts to take ownership of (or *acquire*) each target word (Lines 22–31). To this end, for each `WordDescriptor` w in its `words` array, we start by calling `readInternal` on w's target address `addr` (Line 24; as described above, this handles any helping required in case another active operation owns `addr`). If `addr` is already owned by the current MCAS, we move on to the next word (Line 26). Otherwise, if the current value at `addr` does not match the expected value of w, the MCAS cannot succeed and thus we can skip the next `WordDescriptor`s and go to the second stage (Line 28). If the values do match, we re-check if the operation is still active (line 29); otherwise we go to the second stage – this prevents a memory location from being re-acquired by the current operation *op* in case *op* was already finalized by a helping thread. Finally, we attempt to take ownership of `addr` through a CAS (Line 31). Note that the failure of this CAS might mean that another thread has concurrently helped this MCAS to lock the target word. Therefore, we simply retry taking ownership on this target word, rather than failing the MCAS operation (Line 31).

In the second stage (Lines 32–34), MCAS finalizes the descriptor by atomically changing its status from `ACTIVE` to `SUCCESSFUL` (if all word acquisitions were successful in stage one) or to `FAILED` (otherwise).

Our pseudocode assumes the existence of the `isDescriptor` function, which takes a value and returns `true` if and only if the value is a pointer to a `WordDescriptor`. This function can be implemented, for instance, by designating a low-order *mark bit* in a word to indicate whether it contains a pointer to a descriptor or not [31, 53].

We give a proof of correctness for our algorithm in the full version of our paper [25].

5 Persistent MCAS with $k+1$ CAS and 2 Persistent Fences

We discuss the modifications required to make our volatile MCAS algorithm work with persistent memory. In the full version of our paper [25], we give the complete pseudocode for these modifications.

In the `MCAS` function, after all target locations have been successfully acquired, we add one persistent flush per target word and one persistent fence overall. The persistent fence ensures that all target locations persistently point to their respective `WordDescriptor`s before attempting to modify the status.

When finalizing the status, we mark the status with a special `DirtyFlag`. This flag indicates that the status is not yet persistent. We then perform a persistent flush and fence after the status has been finalized. This ensures that the finalized status of the descriptor is persistent before returning from the MCAS. Finally, we unset the `DirtyFlag` with a simple store; this store cannot create a race with the CAS that finalizes the status because that CAS must fail (the status must be already finalized if some thread is already attempting to unset the dirty flag).

We also modify the `readInternal` function such that, when an operation *op* encounters another operation op' whose status is finalized but still has the `DirtyFlag` set, *op* helps op' persist its status and unsets the `DirtyFlag` on op' status.

Our modifications enforce the following invariants. First, at the time when a descriptor becomes finalized, its acquisitions of target locations are persistent. Second, at the time when an MCAS operation returns, its finalized status is persistent. Third, when a read

or MCAS operation *op* returns, all operations on which *op* depends are finalized and their statuses are persistent. With these invariants, we can argue that our persistent MCAS is correct. By correctness we refer to lock-freedom (liveness) and durable linearizability (safety). Lock-freedom is clearly preserved by our additions, thus we focus on durable linearizability. We examine the point in time when a full-system crash may occur during the execution of an MCAS operation *op*. There are two possibilities to consider:

1. If the crash occurs before *op*'s status was finalized and made persistent, then we know that no operation op' which observed the effects of *op* could have returned before the crash; otherwise, op' would have helped *op* and persisted its status. In this case, neither *op* nor any such op' will be linearized before the crash; during recovery, their effects will be rolled back by reverting any acquired locations to their old values.
2. If the crash occurs after *op*'s status was finalized and made persistent, then *op* is linearized before the crash. During recovery, any locations still acquired by *op* will be detached and given either their new or old values (depending on *op*'s success or failure status), as specified in *op*'s descriptor.

In sum, the recovery procedure of our algorithm is as follows. The recovery goes through each operation descriptor D. If D's status is not finalized, then we roll D back by going through each target location ℓ of D; if ℓ is acquired by D (i.e., points to D), then we write into ℓ its old value, as specified in D. If D's status is finalized, then we detach D and install final values; we go through each target location ℓ of D; if ℓ is acquired by D and D was successful (resp. failed), then we write into ℓ the new (resp. old) value as specified in D.

6 Memory Management

The MCAS algorithm has been presented so far under the assumption that no memory is ever reclaimed. For practical considerations, however, one should to be able to reclaim and/or reuse MCAS operation descriptors. While efficient memory management of concurrent data structures remains an active area of research (see, e.g., [3, 10, 18, 50, 54]), here we describe one possible mechanism suitable for an MCAS implementation. Due to space limitations, we briefly outline the mechanism here and defer its full description, as well as optimizations for persistent memory and efficient reads, to the extended version of our paper [25].

We note that the life cycle of an operation descriptor comprises several phases. Once its status is no longer `ACTIVE`, the (finalized) descriptor cannot be recycled just yet as certain memory locations can point to it. Therefore, we need first to *detach* such a descriptor by replacing the pointers to the descriptor (using CAS) with actual values (respective to whether the corresponding MCAS has succeeded or failed) in affected memory locations. Only after that, a detached descriptor can be recycled, provided no concurrently running thread holds a reference to it. Note that CASes in the detachment phase are necessary only for those affected memory locations that still point to the to-be-detached descriptor, which, as our evaluation shows, is rare in practice.

Our scheme keeps track of two categories of descriptors: (1) those that have been finalized but not yet detached and (2) those that have been detached but to which other threads might still hold references. Similar to RCU approaches [41, 42], we use thread-local epoch counters to track threads' progress and infer when a descriptor can be moved from category (1) to category (2), and when a descriptor from category (2) can be reclaimed.

7 Evaluation

7.1 Experimental Setup

We evaluate our algorithm on a 2-socket Intel Xeon machine with two E5-2630 v4 processors operating at 3.1 GHz. Each processor has 10 cores, each core has 2 hardware threads (40 hardware threads total). Each experimental run lasts 5 seconds; shown values are the average of 5 runs. We base our evaluation on the framework available from the authors of PMwCAS [49, 53].

The baseline of our evaluation is the volatile version of PMwCAS [49, 53], a state-of-the-art implementation of the Harris et al. [31] algorithm. Like the Harris et al. algorithm, volatile PMwCAS requires $3k+1$ CASes per k-CAS. We use PMwCAS as our baseline since (1) it has recent, openly available and well-maintained code and (2) it is to our knowledge the only other MCAS algorithm in which readers do not write to shared memory in the common uncontended case.

PMwCAS implements an optimization of the Harris et al. algorithm: it marks pointers with a special *RDCSS* flag instead of allocating a distinct RDCSS descriptor. However, we found that this optimization made the PMwCAS algorithm incorrect, due to an ABA vulnerability. In our evaluation, we fixed the PMwCAS implementation to allocate and manually manage RDCSS descriptors.

Our evaluation uses three benchmarks: an *array benchmark* in which threads perform MCAS-based read-modify-write operations at random locations in an array, a *doubly-linked list benchmark*, in which threads perform MCAS-based operations on a list implementing an ordered set, and a *B+tree benchmark* in which threads perform MCAS-based operations on a B+-tree. The first two benchmarks are based on the implementation available in [49], and the third is based on PiBench [48] and BzTree [6, 11]. We note, however, that we modified the benchmark in [49] so all threads operate on the same key range (rather than having each thread using a unique set of keys), so we could induce contention by controlling the size of the key range.

In each experiment, we vary the number of threads from 1 to 39 (we reserve one hardware thread for the main thread). Threads are assigned according to the default settings in the evaluation frameworks used [11, 48, 49]. In the array and list benchmarks, threads are assigned in the following way: we first populate the first hardware thread of each core on the first socket, then on the second socket, then we populate the second hardware thread on each core on the first socket, and finally the second hardware thread on each core on the second socket. The B+-tree benchmark uses OpenMP [45], which dictates thread assignment; it also employs a scalable memory allocator [1].

7.2 Array Benchmark

The benchmark consists of each thread performing the following in a tight loop: reading k locations at random from the array ($k = 4$ in our experiments), computing a new value for each location, and attempting to install the new values using an MCAS.

In this benchmark we measure two quantities. The first is throughput: the number of read-modify-write operations completed successfully per time unit. The second metric is the *helping ratio*. We measure the helping ratio by dividing the number of *ongoing* MCAS operations encountered (and helped) during read or MCAS operations by the total number of MCAS operations. A higher helping ratio thus means more operations are slowed down due to the need to help other, incomplete MCAS operations.

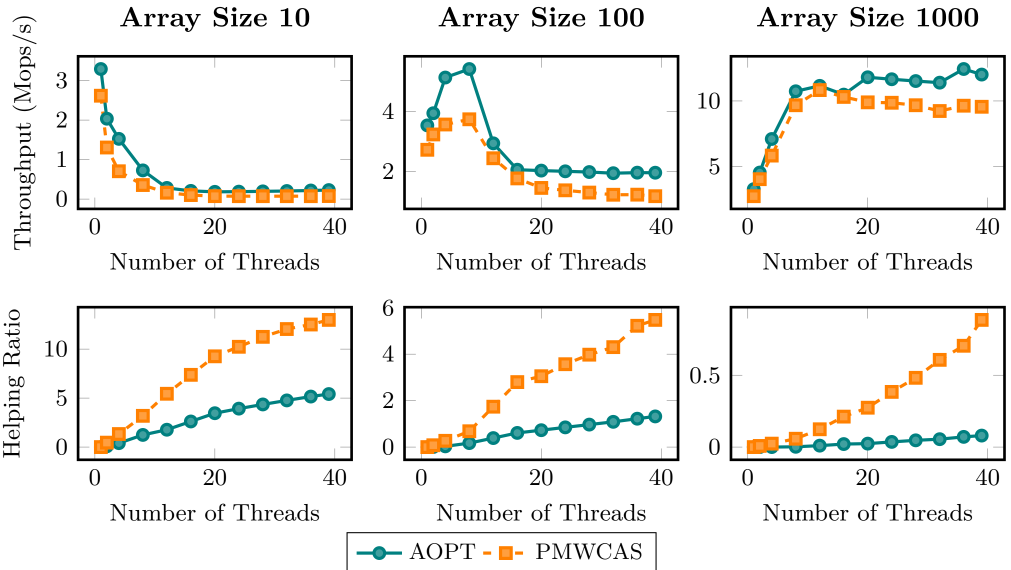


Figure 1 Array benchmark. Top row shows throughput (higher is better), bottom row shows helping ratio (lower is better). Each column corresponds to a different array size (10, 100 and 1000, respectively).

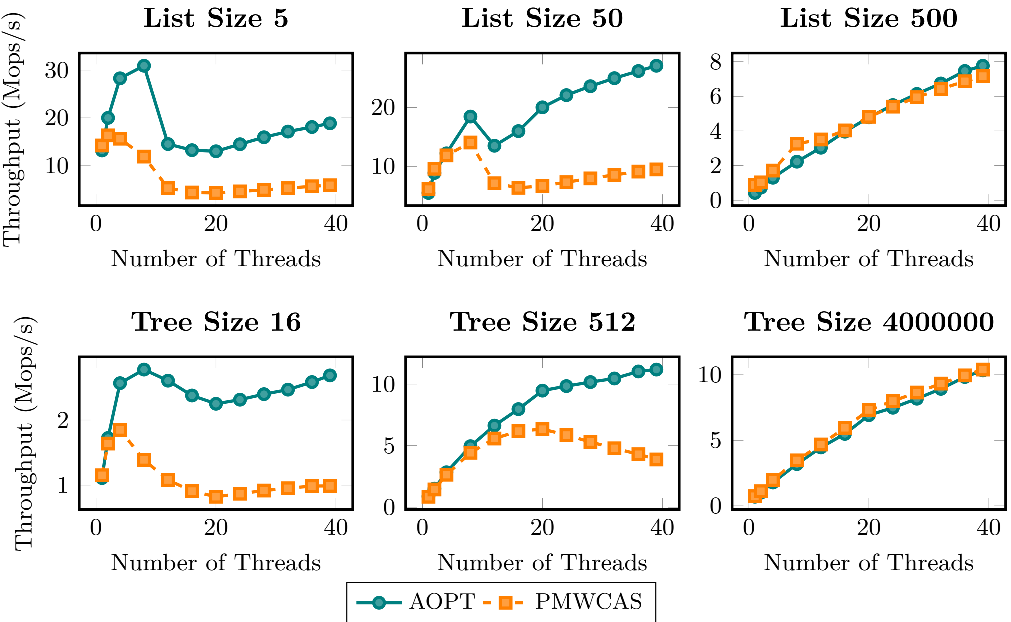


Figure 2 Top row: Doubly-linked list benchmark (80% reads) with different initial list sizes (5, 50 and 500 elements). Bottom row: B+-tree benchmark (80% reads) with different initial tree sizes (16, 512 and 4000000 elements).

We run the benchmark with three array sizes (10, 100, and 1000) in order to capture different contention levels. The results of this benchmark are shown in Figure 1 (our algorithm is denoted $AOPT$ in all figures in this section).

The top row of Figure 1 shows that our algorithm outperforms PMwCAS at every contention level and at every thread count, including in single-threaded mode. This can be explained by two related factors. First, our algorithm has a lower CAS complexity ($k+1$ CASes per k-CAS for our algorithm compared to $3k+1$ for PMwCAS). Second, as a consequence of its lower complexity, in our algorithm there is a shorter "window" for each MCAS operation to interfere with other operations by forcing them to help.

To illustrate the second factor above, we examine the helping ratios of the two algorithms (bottom row of Figure 1). We observe that the helping ratio of our algorithm is considerably lower than that of PMwCAS. This means that, on average, each operation helps (and is slowed down by) fewer MCAS operations in our algorithm than in PMwCAS.

In order to quantify the impact of descriptor cleanup on performance in our algorithm, we also measure the *detaching ratio*: the number of CASes performed in order to detach (in the sense of Section 6) finalized MCAS descriptors, divided by the total number of completed MCAS operations. We find the detaching ratio to be less than 0.001 for every thread count and array size. This is because finalized MCAS descriptors are constantly being replaced by ongoing MCAS operations, and thus recycling these detached descriptors requires no CASes. We conclude that the vast majority of our MCAS operations do not incur any cleanup CASes.

7.3 Doubly-linked List Benchmark

In this benchmark we operate on a shared ordered set object implemented from a doubly-linked list. The list supports search and update (insert and delete) operations. Insertions are done using 2-CAS and deletions are done using 3-CAS. We initialize the list by inserting a predefined (configurable) number of nodes. During the benchmark, each thread selects an operation type (search, insert or delete) at random, according to a configurable distribution; the thread also selects a value at random; it then performs the selected operation with the selected value.

We perform this benchmark with three initial list sizes (5, 50 and 500 elements). The operation distribution is: 80% reads, 20% updates (in all our experiments, updates are evenly distributed among insertions and deletions). As is standard practice, the initial size of the list is half of the key range. Results are shown in the top row of Figure 2. We also ran experiments with 50%, 98%, and 100% reads; performance graphs for these less representative cases are available in the full version of our paper [25].

Our algorithm outperforms PMwCAS for list sizes 5 and 50 by 2.6× and 2.2× on average, respectively. This shows that under high and moderate contention, our algorithm's faster MCAS operations (due to the double effect of lower complexity and lower helping ratio) compensate for its slower read operations (due to the extra level of indirection). In the low contention case (list size 500), PMwCAS outperforms our algorithm at low thread counts and is outperformed at high thread counts. On average, PMwCAS outperforms our algorithm by 1.2×. Under low contention, operations have a low probability to conflict on the same element and thus the lower read complexity of PMwCAS has a stronger impact on performance than the lower MCAS complexity of our algorithm.

7.4 B+-tree Benchmark

In this benchmark we operate on a B+-tree which supports search and update (insert and delete) operations. Insertions and deletions use k-CAS, where k may vary, e.g., depending on whether the operation led to nodes being split or merged.

Similar to the previous benchmark, we initialize the B+-tree with a configurable number of entries; threads then select operations and values at random. We perform the benchmark with 80% reads and three initial tree sizes (16, 512, and 4000000). As for the previous benchmark, performance graphs for the 50%, 98% and 100% reads cases are shown in the full version of our paper [25]. As before, the initial size of the tree is half of the key range. Results are shown in the bottom row of Figure 2.

We observe a similar behavior to the previous benchmark. Our algorithm outperforms PMwCAS under high and medium contention (because it performs fewer CASes and triggers less helping) and is slightly outperformed under low contention (where helping no longer plays a major role).

8 Related Work

Lock- and wait-free implementations of MCAS. Our algorithm shares similarities with previous work [31, 53]: as has become standard practice, it uses operation descriptors and a three-phase design (locking, status-change and unlocking). However, our algorithm introduces key differences with respect to previous work: it defers the unlocking phase and combines it with the reclamation of descriptors, without compromising correctness. This deferment has a triple beneficial effect on complexity: (1) it removes k CASes from the critical path, (2) it allows these CASes to be amortized across several operations, and (3) it removes the onus of ABA-prevention from the locking phase, thus shaving off k further CASes from the latter.

Table 1 summarizes the differences between our algorithm and existing non-blocking MCAS implementations, while the detailed treatment of each of the numerous prior efforts is deferred to the full version of our paper [25]. The results in Table 1 reflect the number of CASes per MCAS operation required for correctness by each algorithm in the common uncontended case. We note that previous MCAS implementations perform descriptor cleanup immediately after applying MCAS, and it is not clear how to separate cleanup from these algorithms while preserving correctness. If we take the cleanup cost into consideration for our algorithm as well, its theoretical (worst-case) complexity becomes $2k + 1$, the same as some of the previous work. As our experiments in Section 7 demonstrate, however, the number of CASes in the cleanup phase is negligible in practice. Furthermore, we highlight the fact that unlike most previous work, including the one that employs $2k+1$ CASes, readers in our case do not write into the shared memory in the common case, even when cleanup is considered.

General techniques. Transactional memory (TM) [34, 51] can be seen as the most general approach to providing atomic access to multiple objects. It allows a block of code to be designated as a transaction and thus executed atomically, with respect to other transactions. Thus, TM is strictly more general than MCAS. This generality comes at a cost: software implementations of transactional memory (STM) have prohibitive performance overheads, whereas hardware support (HTM) is subject to spurious aborts and thus only provides "best-effort" guarantees. Prior work on nonblocking STMs [21, 40] share goals similar to our work; namely reduction of overheads in the critical path. However, these works (i) either employ k extra cleanup CASes [21] on the critical path, incurring precisely the overheads we avoid in our work, or (ii) employ a vastly more complex "stealing" framework to avoid overheads from the critical path [40].

Table 1 Comparison of non-blocking MCAS implementations in terms of the number of CAS instructions required, whether readers perform writes to shared memory or expensive atomic instructions, and the number of persistent fences (all per k-word MCAS, in the uncontended case).

	CASes	Readers write	P. fences
Israeli and Rappoport [37]	$3k + 2$	Yes	N/A
Anderson and Moir [4]	$3k + 2$	Yes	N/A
Moir [43]	$3k + 4$	Yes	N/A
Harris et al. [31]	$3k + 1$	No	N/A
Ha and Tsigas [27, 28]	$2k + 2$	Yes	N/A
Attiya and Hillel [7]	$6k + 2$	N/A	N/A
Sundell [52]	$2k + 1$	Yes	N/A
Feldman et al. [20]	$3k - 1$	Yes	N/A
Wang et al. [53] (volatile)	$3k + 1$	No	N/A
Wang et al. [53] (persistent)	$5k + 1$	No	$2k + 1$
Our algorithm	$k + 1$	No	2

Prior Work on Persistent MCAS. Pavlovic et al. [46] provide an implementation of MCAS for persistent memory which differs from ours in the progress guarantee (theirs is blocking) and hardware assumpions (theirs uses HTM).

Wang et al. [6, 53] introduce the first lock-free persistent implementation, based on the algorithm of Harris et al. [31]. The main differences with respect to our algorithm are outlined in Table 1. This algorithm uses a per-word dirty flag to indicate that the word is not yet guaranteed to be written to persistent memory. Operations encountering a set dirty flag will persist the associated word and then unset the flag. This technique avoids unnecessary persistent flushes, but uses 2 extra CAS instructions per target location in order to manipulate the dirty flag.

In our work we use the recent durable linearizability correctness condition [38], which assumes a full-system crash-recovery model, but other models of persistent memory can be explored in this context [2, 8, 14, 26, 47].

9 Conclusion

Atomic multi-word primitives significantly simplify concurrent algorithm design, but existing implementations have high overhead. In this paper, we propose a simple and efficient lock-free algorithm for multi-word compare-and-swap, designed for both volatile and persistent memory. The complementary lower bound shows that the complexity of our algorithm, as measured in the number of CASes in the uncontended case, is nearly optimal.

References

1 Yehuda Afek, Dave Dice, and Adam Morrison. Cache index-aware memory allocation. In *Proceedings of the International Symposium on Memory Management (ISMM)*, page 55–64. Association for Computing Machinery, 2011.

2 Marcos K Aguilera and Svend Frølund. Strict linearizability and the power of aborting. Technical Report HPL-2003-241, HP Labs, 2003.

3. Dan Alistarh, William M. Leiserson, Alexander Matveev, and Nir Shavit. Forkscan: Conservative Memory Reclamation for Modern Operating Systems. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017*, pages 483–498, 2017.
4. James H. Anderson and Mark Moir. Universal Constructions for Multi-object Operations. In *14th Annual ACM Symposium on Principles of Distributed Computing*, pages 184–193, 1995.
5. James H. Anderson, Srikanth Ramamurthy, and Rohit Jain. Implementing Wait-free Objects on Priority-based Systems. In *16th Annual ACM Symposium on Principles of Distributed Computing*, pages 229–238, 1997.
6. Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. BzTree: A High-Performance Latch-free Range Index for Non-Volatile Memory. In *44th International Conference on Very Large Data Bases*, 2018.
7. Hagit Attiya and Eshcar Hillel. Highly concurrent multi-word synchronization. *Theor. Comput. Sci.*, 412(12-14):1243–1262, 2011.
8. Ryan Berryhill, Wojciech M. Golab, and Mahesh Tripunitara. Robust shared objects for non-volatile main memory. In *19th International Conference on Principles of Distributed Systems, OPODIS 2015*, pages 20:1–20:17, 2015.
9. Anastasia Braginsky and Erez Petrank. A Lock-free B+Tree. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 58–67, 2012.
10. Trevor Alexander Brown. Reclaiming Memory for Lock-Free Data Structures: There Has to Be a Better Way. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, pages 261–270, 2015.
11. Bztree: a high-performance latch-free range index for non-volatile memory. `https://github.com/wangtzh/bztree`, 2019.
12. Diego Cepeda, Sakib Chowdhury, Nan Li, Raphael Lopez, Xinzhe Wang, and Wojciech Golab. Toward linearizability testing for multi-word persistent synchronization primitives. In *23rd International Conference on Principles of Distributed Systems, OPODIS 2019*, volume 153, pages 19:1–19:17, 2019.
13. Nachshon Cohen, Rachid Guerraoui, and Igor Zablotchi. The Inherent Cost of Remembering Consistently. In *Proceedings of the 30th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2018*, 2018.
14. 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 *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009*, pages 133–146, 2009.
15. Tudor David, Aleksandar Dragojevic, Rachid Guerraoui, and Igor Zablotchi. Log-Free Concurrent Data Structures. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018*, 2018.
16. Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 33–48, 2013.
17. David Detlefs, Christine H. Flood, Alex Garthwaite, Paul Martin, Nir Shavit, and Guy L. Steele, Jr. Even Better DCAS-Based Concurrent Deques. In *14th International Conference on Distributed Computing*, pages 59–73, 2000.
18. Dave Dice, Maurice Herlihy, and Alex Kogan. Fast non-intrusive memory reclamation for highly-concurrent data structures. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*, pages 36–45, 2016.
19. Simon Doherty, David L. Detlefs, Lindsay Groves, Christine H. Flood, Victor Luchangco, Paul A. Martin, Mark Moir, Nir Shavit, and Guy L. Steele, Jr. DCAS is Not a Silver Bullet for Nonblocking Algorithm Design. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 216–224, 2004.

20 Steven D. Feldman, Pierre LaBorde, and Damian Dechev. A Wait-Free Multi-Word Compare-and-Swap Operation. *International Journal of Parallel Programming*, 43(4):572–596, 2015.

21 Keir Fraser. *Practical lock-freedom.* PhD thesis, University of Cambridge, UK, 2004.

22 Michal Friedman, Maurice Herlihy, Virendra J. Marathe, and Erez Petrank. A persistent lock-free queue for non-volatile memory. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2018*, pages 28–40, 2018.

23 Michael Greenwald. Non-blocking synchronization and system design. *ph.d. thesis, stanford university*, 1999.

24 Michael Greenwald. Two-handed emulation: how to build non-blocking implementation of complex data-structures using DCAS. In *21st Annual ACM Symposium on Principles of Distributed Computing*, pages 260–269, 2002.

25 Rachid Guerraoui, Alex Kogan, Virendra J. Marathe, and Igor Zablotchi. Efficient multi-word compare and swap. *ArXiv preprint arXiv:2008.02527*, 2020. URL: `https://arxiv.org/abs/2008.02527`.

26 Rachid Guerraoui and Ron R. Levy. Robust Emulations of Shared Memory in a Crash-Recovery Model. In *24th International Conference on Distributed Computing Systems (ICDCS 2004)*, pages 400–407, 2004.

27 Phuong Hoai Ha and Philippas Tsigas. Reactive Multi-Word Synchronization for Multiprocessors. In *12th International Conference on Parallel Architectures and Compilation Techniques (PACT 2003)*, pages 184–193, 2003.

28 Phuong Hoai Ha and Philippas Tsigas. Reactive Multi-word Synchronization for Multiprocessors. *J. Instruction-Level Parallelism*, 6, 2004.

29 Tim Harris, James R. Larus, and Ravi Rajwar. *Transactional Memory: 2nd Edition.* Morgan & Claypool, 2010.

30 Timothy L. Harris and Keir Fraser. Language support for lightweight transactions. In *Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 388–402, 2003.

31 Timothy L. Harris, Keir Fraser, and Ian A. Pratt. A Practical Multi-word Compare-and-Swap Operation. In *16th International Conference on Distributed Computing*, pages 265–279, 2002.

32 Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, November 1993.

33 Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software Transactional Memory for Dynamic-sized Data Structures. In *Proceedings of the Twenty-second Annual Symposium on Principles of Distributed Computing*, pages 92–101, 2003.

34 Maurice Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-free Data Structures. In *20th Annual International Symposium on Computer Architecture*, pages 289–300, 1993.

35 Maurice Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

36 Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual Combined. `https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf`, 2018.

37 Amos Israeli and Lihu Rappoport. Disjoint-Access-Parallel Implementations of Strong Shared Memory Primitives. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 151–160, 1994.

38 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 2016*, pages 313–327, 2016.

39 Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. The Bw-Tree: A B-tree for New Hardware Platforms. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, pages 302–313, 2013.

40 Virendra J. Marathe and Mark Moir. Toward high performance nonblocking software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2008, Salt Lake City, UT, USA, February 20-23, 2008*, pages 227–236. ACM, 2008.

41 Paul E. McKenney. Is parallel programming hard, and, if so, what can you do about it?, 2017.

42 Paul E. McKenney and John D. Slingwine. Read-Copy Update: Using Execution History to Solve Concurrency Problems. In *Parallel and Distributed Computing and Systems*, 1998.

43 Mark Moir. Transparent Support for Wait-Free Transactions. In *11th International Workshop on Distributed Algorithms*, pages 305–319, 1997.

44 Aravind Natarajan and Neeraj Mittal. Fast Concurrent Lock-free Binary Search Trees. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 317–328, 2014.

45 The OpenMP API specification for parallel programming. `https://www.openmp.org/`, 2019.

46 Matej Pavlovic, Alex Kogan, Virendra J. Marathe, and Tim Harris. Persistent Multi-Word Compare-and-Swap. In *ACM Synposium on Principles of Distributed Computing*, 2018.

47 Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory persistency: Semantics for byte-addressable nonvolatile memory technologies. *IEEE Micro*, 35(3):125–131, 2015.

48 Benchmarking framework for index structures on persistent memory. `https://github.com/wangtzh/pibench`, 2019.

49 Persistent multi-word compare-and-swap (PMwCAS) for NVRAM. `https://github.com/microsoft/pmwcas`, 2019.

50 Manuel Pöter and Jesper Larsson Träff. *Stamp-it*, amortized constant-time memory reclamation in comparison to five other schemes. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2018*, pages 413–414, 2018.

51 Nir Shavit and Dan Touitou. Software Transactional Memory. In *14th Annual ACM Symposium on Principles of Distributed Computing*, pages 204–213, 1995.

52 Håkan Sundell. Wait-Free Multi-Word Compare-and-Swap Using Greedy Helping and Grabbing. *International Journal of Parallel Programming*, 39(6):694–716, 2011.

53 Tianzheng Wang, Justin Levandoski, and Per-Ake Larson. Easy Lock-Free Indexing in Non-Volatile Memory. In *34th IEEE International Conference on Data Engineering*, 2018.

54 Haosen Wen, Joseph Izraelevitz, Wentao Cai, H. Alan Beadle, and Michael L. Scott. Interval-based memory reclamation. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–13, 2018.

LL/SC and Atomic Copy: Constant Time, Space Efficient Implementations Using Only Pointer-Width CAS

Guy E. Blelloch
Carnegie Mellon University, Pittsburgh, PA, USA
guyb@cs.cmu.edu

Yuanhao Wei
Carnegie Mellon University, Pittsburgh, PA, USA
yuanhao1@cs.cmu.edu

Abstract

When designing concurrent algorithms, Load-Link/Store-Conditional (LL/SC) is a very useful primitive since it avoids ABA problems. The full semantics of LL/SC are not supported in hardware by any modern architecture, so there has been a significant amount of work on simulations of LL/SC using CAS. However, all previous algorithms that are constant time either use unbounded sequence numbers (and thus base objects of unbounded size), or require $\Omega(MP)$ space to implement M LL/SC objects for P processes.

We present the first constant time implementation of LL/SC from bounded-sized CAS objects using only constant space overhead per LL/SC variable. In particular, our implementation uses $\Theta(M+kP^2)$ space, where k is the number of outstanding LL operations per process, and only requires pointer-width CAS operations. In most algorithms that use LL/SC, k is a small constant which reduces our additive space overhead to $\Theta(P^2)$. Our algorithm can also be extended to implement L word LL/SC objects in $\Theta(L)$ time for LL and SC, $O(1)$ time for VL, and $\Theta((M+kP^2)L)$ space.

To achieve these bounds, our main technical contribution is implementing a new primitive called Single-Writer Copy which takes a pointer to a word sized memory location and atomically copies its contents into another object. The restriction is that only one process is allowed to write/copy into the destination object at a time. The ability to read from one memory location and write to another atomically, and in constant-time, is very powerful and we believe this primitive will be useful in designing other algorithms.

2012 ACM Subject Classification Computing methodologies → Concurrent algorithms

Keywords and phrases LL/SC, Atomic Copy, CAS, Constant Time

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.5

Related Version A full version of the paper is available at `https://arxiv.org/abs/1911.09671`.

Funding This work was supported in part by NSF grants CCF-1901381, CCF-1910030, and CCF-1919223. Yuanhao Wei was also supported by a NSERC PGS-D Scholarship.

Acknowledgements We would like to thank our anonymous reviewers for their helpful comments.

1 Introduction

In lock-free, shared memory programming, it is well known that the choice of atomic primitives makes a big difference in terms of ease of programmability, efficiency, and even computability. Most processors today support a set of basic synchronization primitives such as Compare-and-Swap, Fetch-and-Add, Fetch-and-Store, etc. However, many useful primitives are not supported, which motivates the need for efficient software implementations of these primitives. In this work, we present constant time, space-efficient implementations of a widely used primitive called Load-Link/Store-Conditional (LL/SC) as well as a new primitive we call

© Guy E. Blelloch and Yuanhao Wei;
licensed under Creative Commons License CC-BY
34th International Symposium on Distributed Computing (DISC 2020).
Editor: Hagit Attiya; Article No. 5; pp. 5:1–5:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Single-Writer Copy (**swcopy**). All our implementations use only pointer-width read, write, and CAS. In particular, restricting ourselves to pointer-width operations means that we do not use unbounded sequence numbers, which are often used in other LL/SC from CAS implementations [27, 26, 22]. Many other algorithms based on CAS also use unbounded sequence numbers (often alongside double-wide CAS) to get around the ABA problem and this is sometimes called the *IBM tag methodology* [24, 18]. Our LL/SC implementation can be used in these algorithms to avoid unbounded sequence numbers and double-wide CAS.

The Single-Writer Atomic Copy (**swcopy**) primitive allows processes to atomically read from one memory location and write the result to another. The memory location being read can be arbitrary, but the location being written to has to be a special **Destination** object. A **Destination** object supports three operations, **read**, **write**, and **swcopy** and it allows any process to **read** from it, but only a single process to **write** or **swcopy** into it. This primitive is very useful in announcement array based algorithms because it removes any delay between reading a value and announcing that value. A recent paper uses this primitive to solve various problems related to resource management, such as concurrent reference counting, in constant (expected) time [12].

In this work, we focus on bounded wait-free solutions. Roughly speaking, *bounded wait-freedom* ensures that *each* process makes progress within a bounded number of its own steps regardless of how it is scheduled. In particular, algorithms satisfying this property do not suffer from problems such as deadlock, livelock, and priority inversion. All algorithms presented in this paper take either $O(1)$ or $O(L)$ time (where L is the number of words spanned by the implemented object), which makes them also bounded wait-free. The correctness condition we consider is *linearizability*, which intuitively means that all operations appear to take effect at a single point.

In our results below, the time complexity of an operation is the number of instructions that it executes (both local and shared) in a worst-case execution and space complexity of an object is the number of words that it uses (both local and shared). Counting local objects/instructions is consistent with previous papers on the topic [5, 26, 22]. There has been a significant amount of prior work on implementing LL/SC from CAS [6, 27, 20, 22, 26] and we discuss them in more detail in Section 2.

Result 1 (Load-Link/Store-Conditional): *A collection of M LL/SC objects operating on L-word values shared by P processes, each performing at most k outstanding LL operations, can be implemented with:*

1. $\Theta(L)$ *time for LL and SC,* $O(1)$ *time for VL and CL,*
2. $\Theta((M + kP^2)L)$ *space,*
3. *single word (at least pointer-width) read, write, CAS.*

Our algorithm requires knowing k, the maximum number of outstanding LL/SC operations, and P in advance. In theory, k could be as large as M, but for most data structures implemented from LL/SC, such as Fetch-And-Increment [14] and various Universal Construction [16, 10, 1], k is at most 2. Assuming w-bits is enough to store a pointer, given a data structure implemented from w-bit LL/SC objects, Result 1 implies that it can be implemented from w-bit CAS objects while maintaining the same time complexities and using only $\Theta(kP^2)$ additional space across all instances of the data structure. In contrast, using previous approaches [5, 6, 20, 27] to implement the w-bit LL/SC objects from w-bit CAS objects would require $\Omega(P)$ space *per* LL/SC object.

Our main technical contribution is in implementing a **Destination** object supporting **read**, **write** and **swcopy** with the following bounds.

Result 2 (Single-Writer Copy): *A collection of M* **Destination** *objects, each storing single word values, shared by P processes can be implemented with:*

1. *$O(1)$ worst-case time for read, write, and swcopy,*
2. *$\Theta(M + P^2)$ space,*
3. *single word (at least pointer-width) read, write, CAS.*

To help implement the **Destination** objects, we implement a weaker version of LL/SC which allows the LL operation to fail if it is concurrent with a successful SC. Our version of weak LL/SC is a little different from what was previously studied [5, 20, 27], and we compare the two in more detail in Section 2. Our algorithm for weak LL/SC uses several known techniques which we also cover in Section 2.

Result 3 (Weak Load-Link/Store Conditional): *A collection of M weak LL/SC objects operating on L-word values shared by P processes, each performing at most one outstanding wLL, can be implemented with:*

1. *$\Theta(L)$ time for wLL and SC, $O(1)$ time for VL and CL,*
2. *$\Theta((M + P^2)L)$ space,*
3. *single word (at least pointer-width) read, write, CAS.*

Our implementations of **swcopy** and LL/SC are closely connected. We begin in Section 4 by implementing a weaker version of LL/SC (Result 3). Then, in Section 5, we use this weaker LL/SC to implement **swcopy** (Result 2), and finally, in Section 6, we use **swcopy** to implement the full semantics of LL/SC (Result 1). As we shall see, once we have **swcopy**, our algorithm for regular LL/SC becomes almost identical to our algorithm for weak LL/SC.

2 Related Work

LL/SC Implementations. We consider three types of implementations, single-word LL/SC from single-word CAS, multi-word LL/SC from single-word LL/SC, and multi-word LL/SC from single-word CAS. Existing implementations of each type are summarized in Table 1. All the algorithm shown in the table are wait-free and have optimal time bounds. Aghazadeh and Woelfel [4] do not directly implement multi-word LL/SC from single-word LL/SC, but their single CAS universal construction (Figure 1 of [4]) can be used to achieve the bounds shown in Table 1.

All previous implementations of single or multi-word LL/SC from CAS have one of three drawbacks. They either (1) are not constant time [13, 19], (2) use unbounded sequence numbers [27, 26, 22, 23], or (3) require $\Omega(MP)$ space even in the common case where k is constant [6, 20, 27].

The simplest way to implement an LL/SC object is to tag a CAS object with an unbounded sequence number [27]. To avoid using unbounded sequence numbers, various algorithms were proposed that recycle these tags [27, 6, 20]. However, all these algorithms are only able to implement single-word LL/SC, and in some of these algorithms (i.e. [6] and Figures 4 and 7 of [27]), the size of the simulated LL/SC object is smaller than the size of the CAS objects that they use. Furthermore, these algorithms [27, 6, 20] are unable to efficiently implement a large number of LL/SC objects as they require at least $\Omega(PM)$ space.

To efficiently implement a large number of large LL/SC objects, a common technique is to use a level of indirection combined with some form of memory management. This means that the simulated LL/SC object stores a pointer to a buffer which contains the actual value of the LL/SC object. To perform an **SC**, the process first allocates a new buffer, initializes it with the desired value, and then tries to change the LL/SC object to point to

Table 1 Cost of implementing M LL/SC objects from either LL/SC or CAS, where k represents the maximum number of outstanding LL/SC operations per process, and L is the number of words in each simulated LL/SC object.

Single-word LL/SC from CAS	**Uses Unbounded Sequence Numbers**	**Time**	**Space**
Anderson and Moir [6], Figure 1	No	$O(1)$	$O(P^2M)$
Moir [27], Figure 4	Yes	$O(1)$	$O(P+M)$
Moir [27], Figure 7	No	$O(1)$	$O(P^2+PM)$
Jayanti and Petrovic [20]	No	$O(1)$	$O(PM)$
Jayanti and Petrovic [23]	Yes	$O(1)$	$O(P^2+PM)$
Multi-word LL/SC from LL/SC	**Unbounded Seq. #**	**Time**	**Space**
Aghazadeh and Woelfel [4]	No	$O(L)$	$O(MP^5L)$
Anderson and Moir [5], Figure 2	No	$O(L)$	$O(P^2ML)$
Jayanti and Petrovic [21]	No	$O(L)$	$O(PML)$
Multi-word LL/SC from CAS	**Unbounded Seq. #**	**Time**	**Space**
Michael [26]	Yes	$O(L)$[1]	$O((P^2k+M)L)$
Jayanti and Petrovic [22]	Yes	$O(L)$	$O((P^2+M)L+Pk)$
This Paper	No	$O(L)$	$O((P^2k+M)L)$

the new buffer with a CAS. This technique is used in Michael's algorithm [26], Jayanti and Petrovic's algorithm [22], as well as our algorithm. The general idea has also been used to implement various other objects such as descriptors [7], writable objects [2], and resetable TAS objects [3]. The main difference between these algorithms is in how they recycle buffers to ensure bounded space usage.

In Jayanti and Petrovic's algorithm, they allow **LL** operations to read from a buffer that has already been recycled and they provide a mechanism for the **LL** to detect if this is the case. Whenever an **LL** operation reads from a buffer that has already been recycled, their helping mechanism ensure that the **LL** operation is sent a consistent value that is safe to return. A consequence of reusing buffers prematurely is that the ABA problem could occur when a process tries to update the buffer pointer with a CAS. To prevent this, they tag each pointer with an unbounded sequence number.

Another approach is to have each **LL** operation acquire a hazard pointer [25] to the buffer before accessing it. The buffer will not be recycled as long as there is a hazard pointer to it. This approach is used by our algorithm, Michael's algorithm [26], and others [2, 3]. The main challenge is in acquiring a hazard pointer in constant time, which requires some form of helping. Michael's helping technique [26] makes use of unbounded sequence numbers which seem difficult to recycle because their relative ordering matters. Unlike other algorithms which require storing both a pointer and an unbounded sequence number in the same word, Michael's algorithm stores the sequence number separately. This reduces the likelihood of wrap around in practice. Aghazadeh, Golab and Woelfel present a clever helping technique that avoids unbounded sequence numbers using only registers [2]. However, using their technique to implement M LL/SC objects would require at least MP space. The helping technique we use is encapsulated by our **swcopy** algorithm.

[1] Amortized expected time

There has also been some work on algorithms that do not require knowing the number of processes in advance, for example Jayanti and Petrovic [23] and Doherty et al. [13]. Michael mentions that his algorithm from [26] can be extended to not require knowing k and P in advance by using ideas from Section 3.2 of the hazard pointers paper [25]. Our algorithm can be extended in the same way, but this extension would not maintain our time bounds as allocating a new hazard pointer takes $\Theta(P)$ time in the worst case.

Weak LL/SC Implementations. A variant of Weak LL/SC was introduced by Anderson and Moir [5] and also studied elsewhere [20, 27]. The version we consider is less restrictive than theirs because they require a failed **wLL** operation to return the process id of the **SC** operation that caused it to fail whereas we do not require failed **wLL** operations to return anything. While prior work is able to implement the stronger version of **wLL**, they either employ stronger primitives like LL/SC [5], use unbounded sequence numbers [27], require $O(MP)$ space for M Weak LL/SC objects [5, 20]. To match the bounds stated in Result 3, we implement our own version of weak LL/SC that is sufficient for our **swcopy** algorithm. Conveniently, the majority of our weak LL/SC algorithm from Section 4 can be reused when implementing full LL/SC in Section 6.

Atomic Copy. A similar primitive called **memory-to-memory move** was studied in Herlihy's classic wait-free hierarchy paper [15]. The primitive allows atomic reads and writes to any memory location and supports a **move** instruction which atomically copies the value at one memory location into another. Herlihy showed that this primitive has consensus number infinity. Our **swcopy** is a little different because it allows arbitrary atomic operations (e.g. Fetch-and-Add, Compare-and-Swap, Write, etc) on the source memory location as long as the source objects supports an atomic read. Another difference is that we restrict the destination of the copy to be single-writer. Herlihy's proof that **memory-to-memory move** has unbounded consensus number would also work with our **swcopy** primitive. This means the **Destination** objects defined in Section 5.1 also have consensus number infinity.

3 Preliminaries

A *Compare-and-Swap* (CAS) object stores a value that can be accessed through two operations, read and CAS. The read operation returns the value that is current stored in the CAS object. The CAS operation takes a new value and an expected value and atomically writes the new value into the CAS object if the value that is currently there is equal to the expected value. The CAS operation returns true if the write happens and false otherwise. We say a CAS operation is *successful* if it returns true and *unsuccessful* otherwise.

A *Load-Linked/Store-Conditional* (LL/SC) object stores a value and supports three operations: LL, VL (validate-link), and SC. Sometimes a CL (clear-link) operation is also supported which has no return value. Let O be an LL/SC object. An LL on O simply returns the current value of O. An SC on O performed by process p_i takes a new value and writes it into O if there has not been a CL on O or a successful SC on O since the last LL on O by process p_i. We say that an SC is *successful* if it writes a value into O and *unsuccessful* otherwise. Successful SC operations return true and unsuccessful ones return false. A VL operation by process p_i returns true if there has not been a CL on O or a successful SC on O since the last LL on O by process p_i. A *weak LL/SC* object supports wLL, VL, CL (optional), and SC, and it behaves like an LL/SC object except a wLL on O is allowed to return **empty** if the next SC on O by the same process is guaranteed to be unsuccessful. We

say that a **wLL** is unsuccessful if it returns **empty**. Otherwise, we say the **wLL** is *successful*. When a process performs an LL, the LL is considered to be *outstanding* until the process performs a CL or an SC on the same object. Similarly, when a process performs a successful wLL, the wLL is considered to be *outstanding* until the process performs a CL or an SC on the same object.

We work in the standard asynchronous shared memory model [8] with P processes communicating through base objects that are either registers, or CAS objects. Processes may fail by crashing. All base objects are word-sized and we assume they are large enough to store pointers into memory. We do not hide any extra bits in pointers.

In our model, an *execution* (or equivalently, *execution history*) is an alternating sequence of *configurations* and *steps* C_0, e_1, C_1, e_2, C_2, $\dots$, where C_0 is an *initial configuration*. Each step is a shared operation on a base object. Configuration C_i consists of the values of every base object, and the state of every process after the step e_i is applied to configuration C_{i-1}.

If configuration C precedes configuration C' in an execution, the *execution interval* from C to C' is the set of all configurations and steps between C and C', inclusive. Similarly, the *execution interval* of an operation is the set of all configurations and steps from the first step of that operation to the last step of that operation. The execution interval for an *incomplete operation* is the set of all configurations and steps starting from the first step of that operation.

We say the implementation of an object is *linearizable* [17] if, for every possible execution and for each operation on that object in the execution, we can pick a configuration or step in its execution interval to be its linearization point, such that the operation appears to occur instantaneously at this point. In other words, all operations on the object must behave as if they were performed sequentially, ordered by their linearization points. If multiple operations have the same linearization point, then an ordering must be defined among these operations.

All implementations that we discuss will be *bounded wait-free*. This means that we can give an upper bound on how many steps it takes to complete each operation.

4 Weak LL/SC from CAS

As a subroutine, our **swcopy** operation makes use of multi-word weak LL/SC objects. Recall from Section 3 that weak LL/SC supports three operations **wLL**, **VL** and **SC**, and works the same way as regular LL/SC except that **wLL** is allowed to return **empty**.

In this section, we present a constant time algorithm for multi-word weak LL/SC which works when there is at most one outstanding **wLL** per process. We support a constant time CL operation which can be used to limit the number of outstanding **wLL** operations. Our algorithm assumes that **VL**, CL, and **SC** are only performed on an object if the process has an outstanding **wLL** on that object. This version is sufficient to implement the other algorithms in our paper. To implement M Weak LL/SC objects, each spanning L words, our algorithm takes $O((M + P^2)L)$ space. The proof of correctness for this algorithm can be found in the full version of the paper [11]. The high level idea is to use a layer of indirection and use an algorithm similar to Hazard Pointers [25] to get an upper bound on the memory usage. Many concurrent algorithms use the same high level idea [24, 2, 3].

Each **WeakLLSC** object is represented using a pointer, **buf**, to an L-word buffer storing the current value of the object. A **wLL** operation simply reads **buf** and returns the value that it points to. To perform an **SC**, the process first allocates a new L-word buffer, writes the new value in it, and then tries to write the address of this buffer into **buf** with a CAS. The expected value of this CAS is the value that the process read from **buf** during its previous **wLL**. The problem with this algorithm is that it uses an unbounded amount of space.

Our goal is to recycle buffer objects so that we use at most $O(M + P^2)$ of them. We recycle buffers with a variant of Hazard Pointers that is *worst-case* constant time rather than *expected* constant time. Before accessing a buffer, a **wLL** operation has to first protect it by writing its address to an announcement array. To make sure that its announcement happened "in time", the **wLL** operation re-reads **buf** and makes sure it is the same as what was announced. If **buf** has changed, then the **wLL** operation can return **empty** because it must have been concurrent with a successful **SC** and it can linearize immediately before the linearization point of the **SC**. If **buf** is equal to the announced pointer, then the buffer has been protected and the **wLL** operation can safely read from it.

A **VL** operation by process p_i simply checks if **buf** is equal to the buffer announced by its previous **wLL** operation. If so, it returns true, otherwise, it returns false. A **CL** operation by process p_i simply clears any buffer announced by p_i.

For the purpose of the **SC** operation, each process maintains two lists of buffers: a free list (**flist**) and a retired list (**rlist**). In an **SC** operation, the process allocates by removing a buffer from its local free list. If the CAS instruction performed by the **SC** is successful, it adds the old value of the CAS to its retired list. Each process's free list starts off with $2P$ buffers and we maintain the invariant that the number of buffers in the free list plus the number of buffers in the retired list always equals $2P$. When the free list becomes empty and the retired list hits $2P$ buffers, the process moves some buffers from the retired list to the free list. To decide which buffers are safe to reuse, the process scans the announcement array (the scan doesn't have to be atomic) and moves a buffer from the retired list to the free list if it was not seen in the array. In a later paragraph, we show how this step can be performed in worst-case $O(P)$ time. Since the process sees at most P different buffers in the announcement array during its scan, its free list's size is guaranteed to be at least P after this step. A process performs this expensive step at most once every P **SC** operations because the process has at least P free buffers after this step.

Pseudo-code is shown in Listing 1. In the pseudo-code, we use **A[i].read** and **A[i].write** to read from and write to the announcement array **A**. Since each element of the announcement array is a pointer type, **read** and **write** are trivially implemented using the corresponding atomic instruction. We wrap these instructions in function calls so that the code can be reused in Section 6. The argument from the previous paragraph also implies that **flist** cannot be empty on line 43, so we do not run the risk of dereferencing an invalid pointer on line 44. In the pseudo-code, we use **T*** to denote a pointer to an object of type T and **Value[L]** to denote an array of L word-sized values. If **var** is a variable, **&var** is used to denote the address of that variable.

Initialization. Each **WeakLLSC** object starts off pointing to a different Buffer object and each free list is initialized with $2P$ distinct Buffers. Buffers in the free lists are not pointed to by any of the **WeakLLSC** objects and no Buffer appears in two free lists. This property is maintained as the algorithm executes.

Linear-time set difference. In the pseudo-code, both **rlist** and **reserved** represent lists of buffers. The operation **rlist** $\setminus$ **reserved** on line 57 computes the difference between the two lists. What makes the original hazard pointers algorithm *expected* rather than *worst-case* constant time is that it uses a hash table to perform this step. Another approach in the literature is to have each process maintain an array of size B where B is the number of buffers [2, 3]. The array is used to keep track of the buffers that appear in **reserved**. Since $B \in \Theta(M + P^2)$, having an array per process requires too much space in our case, so instead,

Listing 1 Amortized constant time implementation of L-word Weak LL/SC from CAS. Code for process p_i is shown.

```
1  shared variables:
2    Buffer* A[P]; // announcement array

4  local variables:
5    list<Buffer*> flist;
6    list<Buffer*> rlist;
7    // initial size of flist is 2P
8    // rlist is initially empty

10 struct Buffer {
11   // Member Variables
12   Value[L] val;
13   int pid;
14   bool seen;

16   void init(Value[L] initialV) {
17     copy initialV into val
18     pid = -1; seen = 0; } };

20 struct WeakLLSC {
21   // Member Variables
22   Buffer* buf;

24   // Constructor
25   WeakLLSC(Value[L] val) {
26     buf = new Buffer();
27     buf->init(val); } // non-atomic

29   void CL() { A[i].write(NULL); }

30   optional<Value[L]> wLL() {
31     Buffer* tmp = buf;
32     A[i].write(tmp);
33     if(buf == tmp)
34       return tmp->val; //non-atomic
35     else return empty; }

37   bool VL() {
38     Buffer* old = A[i].read();
39     return buf == old; }

41   bool SC(Value[L] newV) {
42     Buffer* old = A[i].read();
43     Buffer* newBuf = flist.remove();
44     newBuf->init(newV);
45     bool b = CAS(&buf, old, newBuf);
46     if(b) retire(old);
47     else flist.add(newBuf);
48     A[i].write(NULL);
49     return b; }

51   void retire(Buffer* old) {
52     rlist.add(old);
53     if(rlist.size() == 2*P) {
54       list<Buffer*> reserved = [];
55       for(int j = 0; j < P; j++)
56         reserved.add(A[j].read());
57       newlyFreed = rlist \ reserved;
58       rlist.remove(newlyFreed);
59       flist.add(newlyFreed); }}};
```

we borrow space from the buffers to perform the marking step. This is done by adding enough space for a process id, **pid**, and a bit, **seen**, to each **Buffer** object. To perform the set difference **rlist** \ **reserved**, the process first visits each buffer **B** in **rlist** and prepares the buffer by setting **B.pid** to its own process id and setting **B.seen** to false. Then, the process loops through **reserved** and for each buffer, it sets **seen** to true if **pid** equals its own process id. Next, the process loops through **rlist** again and constructs a list of buffers that have not been seen. This list is the result of the set difference. Finally, the process has to reset everything by setting **B.pid** to $\perp$ for each **B** in **rlist**.

Deamortization. So far, the algorithm we have described takes *amortized* constant time. To deamortize it, each process can maintain two sets of retired list and free lists. Each time the process removes from one free list, it performs a constant amount of work towards populating the other. This is a commonly used technique for deamortizing garbage collection cost [9]. A different deamortization approach was developed by Aghazadeh, Golab and Woelfel [2] in the context of a different problem.

Space complexity. The algorithm uses P shared space for the announcement array, $O(P^2)$ local space for all the retired and free lists, and $O((M+P^2)L)$ shared space for all the buffers and **WeakLLSC** objects. Therefore, its total space usage is $O((M+P^2)L)$. In addition, it only uses pointer-width read, write, CAS as atomic operations, so it fulfills the claims in Result 3.

5 Single-Writer Atomic Copy

The copy primitive, **swcopy**, can be used to atomically read a value from some source memory location and write it into a **Destination** object. Our **Destination** objects are single-writer and we allow the source memory location to be modified by any instruction (e.g. write, fetch-and-add, swap, CAS, etc). The sequential specifications of **swcopy** and **Destination** objects are given below.

▶ **Definition 1.** *A* **Destination** *object supports 3 operations* **read**, **write** *and* **swcopy** *with the following sequential specifications:*

- **read()**: *returns the current value in the* **Destination** *object (initially* $\perp$*).*
- **write(Value v)**: *sets* **v** *as the current value of the* **Destination** *object.*
- **swcopy(Value* addr)**: *reads the value pointed to by* **addr** *and sets it as the current value of the* **Destination** *object.*

Processes can perform **read** *operations at any time, but no two* **write**/**swcopy** *operations can be concurrent.*

Destination objects are very useful in announcement array based algorithms where it is beneficial to read and announce atomically. Section 5.1 describes our implementation and we prove correctness in Section 5.2.

5.1 Algorithm for Single-Writer Atomic Copy

In this section, we show how to implement **Destination** objects that support **read**, **write**, and **swcopy** in $O(1)$ time and $O(M+P^2)$ space (where M is the number of **Destination** objects). Our algorithm only requires pointer-width read, write and CAS instructions.

We represent a **Destination** object **D** internally using a triplet, **D.val**, **D.ptr**, and **D.old**. When there is no **swcopy** in progress, **D.val** stores the current value of the **Destination** object. When there is a copy in progress, **D.ptr** stores a pointer to the location that is being copied from. Operations that see a copy in progress will help complete the copy. Finally, **D.old** stores the previous value of the **Destination** object. The variables **D.val** and **D.ptr** are stored together in a multi-word **WeakLLSC** object (defined in Section 4). This allows us to read from and write to them atomically as well as prevent any potential ABA problems. The downside is that the only way to read **D.val** or **D.ptr** is through a **wLL** operation which can repeatedly fail due to concurrent **SC** operations. For this reason, we keep **D.old** in a separate word, so that the readers can return **D.old** if they fail too many times on **wLL**. Readers will only perform **SC** operations that change **D.ptr** from not **NULL** to **NULL**. Therefore, the writer's **wLL** will be successful whenever **D.ptr** is **NULL**. We will maintain the invariant that **D.ptr** is **NULL** whenever there is no ongoing **swcopy**. We also ensure that **D.ptr** changes exactly twice during each **swcopy**. The first change writes a valid pointer and the second change resets it back to **NULL**.

A **swcopy(Value* src)** on **Destination** object **D** begins by backing up the current value from **D.val** into **D.old**. At this point, **D.ptr** is guaranteed to be **NULL**, so the writer can successfully read **D.val** with a **wLL**. The **swcopy** proceeds by writing **src** into **D.ptr** with

Listing 2 Atomic copy (single-writer). Code for process p_i.

```
1  struct Data {
2    Value val;
3    Value* ptr;};

5  struct Destination {
6    // Member Variables
7    Value old;
8    WeakLLSC<Data> data;
9    // data is initially ⟨⊥, NULL⟩

11   void swcopy(Value *src) {
12     // This wLL() cannot fail
13     old = data.wLL().val;
14     data.SC(⟨⊥, src⟩);
15     Value val = *src;
16     optional<Data> d = data.wLL();
17     if(d != empty && d.ptr != NULL)
18       data.SC(⟨val,NULL⟩);
19     else if(d != empty)
20       data.CL(); }
21   void write(Value new_val) {
22     // This wLL() cannot fail
23     old = data.wLL().val;
24     data.SC(⟨new_val, NULL⟩); }

26   Value read() {
27     optional<Data> d = data.wLL();
28     if(d == empty) {
29       d = data.wLL();
30       if(d == empty) return old;}
31     if(d.ptr == NULL) {
32       data.CL();
33       return d.val; }
34     Value v = *(d.ptr);
35     if(data.SC(⟨v, NULL⟩)) return v;
36     d = data.wLL();
37     data.CL();
38     if(d != empty && d.ptr == NULL)
39       return d.val;
40     return old; } };
```

an **SC**. Finally, it reads the value **v** pointed to by **src** and tries to write (**v**, **NULL**) into (**D.val**, **D.ptr**) with an **SC**. It is not a problem if the **SC** fails because that means another process has helped complete the copy.

To **read** from **D**, a process begins by trying to read the pair (**D.val**, **D.ptr**) with a **wLL**. If it fails on this **wLL** twice, then it is safe to return **D.old** because the value of D has been updated at least once during this **read**. Now we focus on the case where one of the **wLL**s succeed and reads (**D.val**, **D.ptr**) into local variables (**val**, **ptr**). If **ptr** is NULL, then **val** stores the current value, which the **read** returns. If **ptr** is not NULL, then there is a concurrent **swcopy** operation and the **read** tries to help by reading the value **v** referenced by **ptr** and writing (**v**, **NULL**) into (**D.val**, **D.ptr**) with an **SC**. If the **SC** is successful, then the **read** returns **v**. Otherwise, the process performs one last **wLL**. If it is successful and sees that **D.ptr** is **NULL**, then it returns **D.val**. Otherwise, it is safe to return **D.old**.

The **write** operation is the most straightforward to implement. Since each **Destination** object only has a single writer, a **write** operation simply uses a **wLL** and an **SC** to store the new value into **D.val**. There cannot be any successful **SC** operations concurrent with the **wLL** because the other processes can only succeed on an **SC** during a **swcopy** operation. Therefore, the **wLL** and **SC** performed by the **write** will both always succeed. The **write** operations also needs to keep **D.old** up to date so it updates it before performing the **SC**.

Pseudo-code is shown in Listing 2. Calls to CL are inserted appropriately so that there is at most one outstanding **wLL** per process. By simple inspection of the pseudo-code, we can verify that each operation takes constant time. To implement M **Destination** objects, it uses M **WeakLLSC** objects, each spanning two words, and $O(M)$ pointer-width read, write, CAS objects. Using the algorithm from Result 3 to implement the **WeakLLSC** objects, we get an overall space usage of $O(M + P^2)$, which satisfies the properties in Result 2.

5.2 Correctness Proof

We begin by defining the linearization points of **write** and **swcopy**. The linearization point of **read** is more complicated, so we will defer its definition until later. Each **write** operation is linearized on line 24. For **swcopy** operations, we will prove in Claim 4 that there exists exactly one **SC** instruction S during the **swcopy** that sets **data.ptr** to **NULL** and that this instruction is either executed by line 18 of the **swcopy** or line 35 of a concurrent **read** R. If S is executed by line 18, then the **swcopy** is linearized when it executes line 15. Otherwise, the **swcopy** is linearized on line 34 of R. We show in Claim 5 that this linearization point is contained in the execution interval of the **swcopy**. Note that partially complete **swcopy** operations without an **SC** instruction setting **data.ptr** to **NULL** are not linearized.

For the purposes of this proof, we will focus on an execution E consisting of operations on a single **Destination** object **D**. For simplicity, we will write **data.ptr** instead of **D.data.ptr** and **swcopy** instead of **D.swcopy**. At each configuration C in E, we define the *current value* of **D** to be the value written by the last modifying operation (either a **write** or a **swcopy**) linearized before C. To show that the algorithm in Listing 2 is correct, it suffices to show that the value returned by each **read** operation is the value of **D** at some step during the **read**. The **read** is linearized at that step.

We first prove two useful claims about the structure of the algorithm. Throughout the proof, it is important to remember that there can only be one **write** or **swcopy** operation active at any time. We say that a pointer is *valid* if it is not **NULL**.

▷ Claim 2. Suppose the **SC** performed by a **read** operation is successful, then **data.ptr** was valid at all configurations between line 31 of the **read** and the **SC**.

Proof. Let R be a **read** operation with a successful **SC** operation S on line 35. Let L be the successful **wLL** operation corresponding to S. L was either executed on line 27 of R or line 29 of R. Since S is successful, **data.ptr** cannot have changed between L and S. If **data.ptr** was **NULL** in this interval, then the if statement on line 31 would have evaluated to true, and S would not have been executed. Therefore, **data.ptr** is valid at all configurations between L and S, which includes all configurations between line 31 of R and S. ◁

▷ Claim 3. Suppose the **SC** on line 18 of a **swcopy** operation is successful, then **data.ptr** is valid at all configurations between line 14 of the **swcopy** and the **SC**.

Proof. Let Y be a **swcopy** operation with a successful **SC** operation S on line 18. For S to be executed, the if statement on line 17 must evaluate to true, which means that the **wLL** operation L on line 16 must have been successful. Since S is a successful **SC**, **data.ptr** cannot have changed between L and S. Again, due to the if statement on line 17, **data.ptr** is valid in this interval.

It remains to show that **data.ptr** is valid between lines 14 and 16. Suppose for contradiction that **data.ptr** is **NULL** in this interval. The only operation that can change **data.ptr** to be valid is on line 14 of **swcopy**, so **data.ptr** would have remained **NULL** until the end of Y. This contradicts the fact that **data.ptr** is valid between L and S. Therefore **data.ptr** is valid at all configurations between line 14 of Y and S. ◁

The following two claims show that the linearization points of each **swcopy** operation is well-defined and lie within its execution interval.

▷ Claim 4. There is exactly one successful **SC** instruction during a **swcopy** Y that sets **data.ptr** to **NULL** and this **SC** instruction is either from line 18 of Y or line 35 of some **read**. Furthermore, this **SC** instruction is executed after the first **SC** of Y.

Proof. Let Y_i be the ith **swcopy** operation in E. The order is well defined because there can be only one **swcopy** operation active at a time. We proceed by induction on i, alternating between two different propositions. Let P_i be the proposition that **data.ptr** equals **NULL** at the start of Y_i. Let Q_i be the proposition that Claim 4 holds for Y_i. P_1 acts as our base case and for the inductive step, we show that P_i implies Q_i and that Q_i implies P_{i+1}.

For the base case, we know that **data.ptr** is initialized to **NULL** and it can only be changed to something that is valid by the first **SC** of a **swcopy** operation. Therefore, **data.ptr** remains **NULL** until the first **swcopy** operation.

To show that P_i implies Q_i, we use the same argument to argue that **data.ptr** is **NULL** between the first **wLL/SC** pair performed by Y_i. By Claim 2, no **SC** operation from a **read** can succeed between the first **wLL/SC** pair of Y_i. This means the first **SC** performed by Y_i (on line 14) is guaranteed to succeed and set **data.ptr** to something valid. Between the first **SC** of Y_i and the end of Y_i, the only two operations that could possibly change Y_i are the **SC** on line 18 of Y_i and line 35 of a **read** operation. During this interval, if there are no successful **SC** operations from line 35, then the **SC** on line 18 of Y_i is guaranteed to execute and succeed. This shows that there is at least one successful **SC** from line 18 or line 35 between the first **SC** and the end of Y_i. By Claim 3, the **SC** on line 18 cannot succeed if **data.ptr** is **NULL**, and similarly for the **SC** on line 35 (Claim 2). Since the **SC**s on lines 18 and 35 both set **data.ptr** to **NULL**, at most one such **SC** can succeed between the first **SC** of Y_i and the end of Y_i. Therefore, P_i implies Q_i.

All that remains is to show that Q_i implies P_{i+1}. From Q_i, we know that **data.ptr** gets set to **NULL** between the first **SC** of Y_i and the end of Y_i. It will remain **NULL** until the first **SC** of Y_{i+1}, which means it is **NULL** at the beginning of Y_{i+1}. ◁

▷ Claim 5. The linearization point of each **swcopy** operation Y lies between the first **SC** and the end of Y.

Proof. A **swcopy** operation Y is either linearized at line 15 of its own operation or line 34 of a **read** operation R. Clearly, this lemma holds in the former case, so we focus on the latter.

By Lemma 4, we know that the **SC** operation S on line 35 of R happens between the first **SC** of Y and the end of Y. This means that the successful **wLL** operation L corresponding to S must have happened after the first **SC** of Y and before S. From the code, we can see that line 34 of R (which is the linearization point of Y) happens between L and S. By transitivity, the linearization point of Y happens between the first **SC** of Y and the end of Y. ◁

The next claim is useful for arguing that **data.ptr** is **NULL** at all configurations during a **write** operation and at all configurations between the beginning and the first **SC** of a **swcopy**.

▷ Claim 6. **data.ptr** can only be valid between the first **SC** of a **swcopy** and the end of the **swcopy**.

Proof. **data.ptr** is initially **NULL** and the only instruction that sets **data.ptr** to something valid is the first **SC** of a **swcopy** instruction. By Claim 4, we know that after this **SC** instruction and before the end of the **swcopy**, **data.ptr** is set back to **NULL**. Therefore, **data.ptr** can only be valid between the first **SC** of a **swcopy** and the end of the **swcopy**. ◁

Finally, we prove the main claim.

▷ Claim 7. If **data.ptr** is **NULL**, then **data.val** stores the current value of **D**.

Proof. We will prove this by induction on the execution history E. The fields of **D** are initialized so that **data.ptr** stores **NULL** and **data.val** stores the initial value of **D**. Therefore this claim holds for the initial configuration. Suppose, for induction, that this claim holds for some configuration C, we need to show that it holds for the next configuration C'. If **D.data.ptr** is valid in C', then the claim is vacuously true, so suppose **D.data.ptr** is **NULL** at C'. Let S be the step between C and C'. There are four cases for S; either (1) S is a successful **SC** operation from line 18, (2) S is a successful **SC** operation from 35, (3) S is a successful **SC** operation from line 24, or (4) S is not a successful **SC** on **data**.

In the first case, S is executed by a **swcopy** operation Y, which is linearized on line 15 of Y. The value written into **data.val** by S is equal to the value of the source location at the linearization point of Y. There cannot be any **swcopy** or **write** operation linearized between the linearization point of Y and S, so **data.val** stores the current value of **D** at C'.

For the second case, we first show that S is concurrent with a **swcopy** operation. Due to the if statement on line 31, S can only be successful if **data.ptr** is valid. By Claim 6, **data.ptr** can only be valid during a **swcopy** operation, which means that S must occur during some **swcopy** operation Y. By Claim 4, we know that Y is linearized on line 34 of the **read** operation that executed S. Since there can only be one **swcopy** or **write** at a time, there cannot be any other **swcopy** or **write** operation linearized between the linearization point of Y and S. Since the value written into **data.val** by S is equal to the value of the source location at the linearization point of Y, **data.val** stores the current value of **D** at C'.

For case (3), S is the linearization point of a **write** operation and S writes the value of that **write** operation into **data.val**. This means **data.val** stores the current value of **D** at C'.

Finally, for the fourth case, suppose S is not a successful **SC** on **data**. This means the value of **data.val** will remain unchanged between C and C'. By the inductive hypothesis, **data.val** stores the current value of **D** at C, so it suffices to show that there are no **write** or **swcopy** operation linearized at S. By Claims 2 and 3, **data.ptr** is valid at the linearization point of a **swcopy** operation. Since **data.ptr** is **NULL** both before and after S, no **swcopy** operation can be linearized at S. To show that no **write** operations can be linearized at S, it suffices to show that the **SC** at the linearization point of a **write** operation is always successful. Let W be a **write** operation by process p. The only **SC** operations on **data** that can be concurrent with W are from **read** operations. By Claim 6, **data.ptr** is **NULL** for the duration of W, and by Claim 2, no **SC** from a **read** operation can succeed during W. Therefore, both the **wLL** and the **SC** performed by W are guaranteed to succeed. ◁

Suppose R is a completed **read** operation that returns v. As previously noted, to prove that Listing 2 is a linearizable implementation of a **Destination** object, it suffices to show that there exists a step during R such that the value of the **Destination** object at that step is equal to v. We linearize R at that step. If there are multiple operations linearized at the same step, **read** operations are always linearized last. Note that there cannot be multiple **write** or **swcopy** operations linearized at the same step.

There are five possible return points for a **read** operation. If R returns on lines 35 or 39, then on lines 35 or 36 (respectively), we know that **data.val** equals v and **data.ptr** equals **NULL**. If R returns on line 33, then either on line 27 or line 29, **data.val** equals v and **data.ptr** equals **NULL**. By Claim 7, **data.val** stores the current value of the **Destination** object whenever **data.ptr** is **NULL**, so for these three return points there exists a step during R such that v is the current value.

Now suppose R returns on lines 30 or 40 (i.e. the case where R reads and returns the value in **D.old**). There must have been two successful **SC**s, S_1 and S_2, on **D.data** during R. In the case where R returns on line 30, these two successful **SC** operations occurred during

the **wLL**s on lines 27 and 29. In the case where R returns on line 40, S_1 was the one that caused the **SC** on line 35 to fail and S_2 occurred during the **wLL** on line 36. By Claims 2 and 3, there cannot be two successful **SC**s from lines 18 or 35 in a row without a successful **SC** from line 14 of **swcopy** or line 24 of **write** in between. Therefore, there must have been a successful **SC** either from line 14 of **swcopy** or line 24 of **write** during R. We'll use S to denote this **SC** operation. In both cases, the line immediately before S updates **D.old** by first performing a **wLL** on **data**. By Claim 6, **data.ptr** equals **NULL** during this **wLL** operation and since the only **SC** operations that could potentially cause it to fail are by **read** operations, by Claim 2, this **wLL** is guaranteed to succeed. By Claim 7, **data.val** stores the current value v' at the time of this **wLL** operation. This value gets written into **old**, so **old** stores the current value immediately after this step. Since there is only a single **write** or **swcopy** at a time, **old** still contains the current value immediately before S. R reads and returns the value of **old** at its last step so there are two cases. Either R reads v' from **old** or it reads something newer. If R reads v', then it returns the current value of **D** at the step immediately before S (which happens during R). If R reads something newer, then **old** must have been updated between S and the end of R. This can only happen on line 13 or on line 23, and we've already argued that **old** stores the current value of **D** on these two lines. Therefore, in either case, R returns a value that was the current value of **D** at some point during R.

6 LL/SC from CAS

Now we have all the tools to implement LL/SC from CAS (Result 1). In this section, we present an algorithm that works whenever there is at most one outstanding LL per process. In the full version of this paper [11], we show how to generalize this to support k outstanding LLs per process.

This algorithm is almost identical to our algorithm for weak LL/SC from CAS (Section 4). To ensure that the **LL** operation always succeeds, we use **swcopy** to atomically read and announce the current buffer (lines 31 and 32 of Listing 1). This means that the announcement array needs to be an array of **Destination** objects (from Section 5.1) rather than raw pointers. Other than that, the algorithm remains the same. Note that **Destination** objects internally use Weak LL/SC objects, which in turn use an announcement array. The announcement array used by the Weak LL/SC objects is different from the one used by the LL/SC objects. Listing 3 shows just the difference between this algorithm and the weak LL/SC algorithm from Listing 1.

This algorithm uses $O((M + P^2)L)$ pointer-width read, write, CAS objects just like in Listing 1, but it also uses P **Destination** objects for the announcement array. From Result 2, we know that P **Destination** objects can be implemented in constant time and $O(P^2)$ space, so this algorithm achieves the bounds in Result 1.

6.1 LL/SC Correctness Proof (outline)

In the proof of correctness for our **WeakLLSC** algorithm, the key property is that at the linearization point of a successful **wLL** operation on a **WeakLLSC** object **X** by process p_i, both **A[i]** and **X.buf** point to the same buffer. We linearize our **LL** operation from Listing 3 so that the same property holds. At the linearization point of the **swcopy** operation on line 7, **A[i]** and **buf** are equal, so we linearize the **LL** at this point. **SC** and **VL** operations are linearized just as they were in the **WeakLLSC** algorithm. In the full version of the paper, we describe how to adapt our **WeakLLSC** proof to work for this algorithm.

Listing 3 Amortized constant time implementation of L-word LL/SC from CAS. Code for process p_i is shown. The algorithm is exactly the same as Listing 1 except for the parts that are shown. Note that the type of the announcement array has changed, so the way we **read** from and **write** to the announcement array is also different.

```
1  Destination<Buffer*> A[P];

3  struct LLSC {
4    Buffer* buf;
5    ...
6    Value[L] LL() {
7      A[i].swcopy(&buf);
8      Buffer* tmp = A[i].read();
9      return tmp->val; }
10 };
```

7 Conclusion and Discussion

We introduced a new primitive called **swcopy** and described how to implement it efficiently. We used this primitive to implement constant time LL/SC from CAS in a way that is both space efficient and avoids the use of unbounded sequence numbers. We believe the **swcopy** primitive can simplify the design of many other concurrent algorithms and make reasoning about them more modular.

We restricted the **Destination** objects in Section 5 to be single-writer because it was sufficient for the use cases we considered. It's possible to generalize this interface to support writes and copy operations that are concurrent with each other. However, it is unclear what the desired behavior should be in this case. One option would be to give atomic copy "store" semantics where the value of the **Destination** object is determined by the last **write** or **copy** to that location. Another option would be to give atomic copy "CAS" semantics where the **copy** is only successful if the **Destination** object stores the expected value. The right choice of definition will likely depend on the potential application.

Another interesting extension of the atomic copy primitive is to have it apply a function on the value being copied before writing it into the destination. This is similar to a Read-Modify-Write instruction except the read and the write are on two different memory locations.

Composing our LL/SC from CAS algorithm with Jayanti and Petrovic's multi-word LL/SC from single-word LL/SC algorithm [21] yields an implementation of multi-word LL/SC from CAS that uses $\Theta(P^2k + PML)$ space. This space complexity is sometimes better than the space complexity of our algorithm. In particular, If $k = M = O(1)$ and $L = \Theta(P)$, then the combined algorithm uses only $\Theta(P^2)$ space whereas our algorithm uses $\Theta(P^3)$ space.

References

1 Yehuda Afek, Dalia Dauber, and Dan Touitou. Wait-free made fast. In *Symposium on Theory of Computing (STOC)*, pages 538–547, 1995.

2 Zahra Aghazadeh, Wojciech Golab, and Philipp Woelfel. Making objects writable. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 385–395, 2014.

3 Zahra Aghazadeh and Philipp Woelfel. Space-and time-efficient long-lived test-and-set objects. In *International Conference on Principles of Distributed Systems (OPODIS)*, pages 404–419. Springer, 2014.
4 Zahra Aghazadeh and Philipp Woelfel. Upper bounds for boundless tagging with bounded objects. In *International Symposium on Distributed Computing (DISC)*, pages 442–457, 2016.
5 James H Anderson and Mark Moir. Universal constructions for large objects. In *International Workshop on Distributed Algorithms (WDAG)*, pages 168–182. Springer, 1995.
6 James H Anderson and Mark Moir. Universal constructions for multi-object operations. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 184–193, 1995.
7 Maya Arbel-Raviv and Trevor Brown. Reuse, don't recycle: Transforming lock-free algorithms that throw away descriptors. *arXiv preprint arXiv:1708.01797*, 2017.
8 Hagit Attiya and Jennifer Welch. *Distributed computing: fundamentals, simulations, and advanced topics*, volume 19. John Wiley & Sons, 2004.
9 Henry G Baker Jr. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, 1978.
10 Greg Barnes. A method for implementing lock-free shared-data structures. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 261–270, 1993.
11 Guy E. Blelloch and Yuanhao Wei. LL/SC and atomic copy: Constant time, space efficient implementations using only pointer-width CAS, 2019. arXiv:1911.09671.
12 Guy E. Blelloch and Yuanhao Wei. Concurrent reference counting and resource management in wait-free constant time, 2020. arXiv:2002.07053.
13 Simon Doherty, Maurice Herlihy, Victor Luchangco, and Mark Moir. Bringing practical lock-free synchronization to 64-bit applications. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 31–39. ACM, 2004.
14 Faith Ellen and Philipp Woelfel. An optimal implementation of fetch-and-increment. In *International Symposium on Distributed Computing (DISC)*, pages 284–298. Springer, 2013.
15 Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.
16 Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(5):745–770, 1993.
17 Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.
18 IBM. IBM System/370 Extended Architecture, Principles of Operation, publication no. sa22-7085. 1983.
19 Amos Israeli and Lihu Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 151–160. ACM, 1994.
20 Prasad Jayanti and Srdjan Petrovic. Efficient and practical constructions of LL/SC variables. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 285–294. ACM, 2003.
21 Prasad Jayanti and Srdjan Petrovic. Efficient wait-free implementation of multiword LL/SC variables. In *IEEE International Conference on Distributed Computing Systems (ICSCS)*, pages 59–68. IEEE, 2005.
22 Prasad Jayanti and Srdjan Petrovic. Efficiently implementing a large number of LL/SC objects. In *International Conference on Principles of Distributed Systems (OPODIS)*, pages 17–31. Springer, 2005.
23 Prasad Jayanti and Srdjan Petrovic. Efficiently implementing LL/SC objects shared by an unknown number of processes. In *International Workshop on Distributed Computing (IWDC)*, pages 45–56. Springer, 2005.
24 Maged M Michael. ABA prevention using single-word instructions. *IBM Research Division, RC23089 (W0401-136), Tech. Rep*, 2004.

25 Maged M Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, 2004.

26 Maged M Michael. Practical lock-free and wait-free LL/SC/VL implementations using 64-bit CAS. In *International Symposium on Distributed Computing (DISC)*, pages 144–158. Springer, 2004.

27 Mark Moir. Practical implementations of non-blocking synchronization primitives. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 219–228, 1997.

Message Complexity of Population Protocols

Talley Amir
Yale University, New Haven, CT, USA
https://cpsc.yale.edu/people/talley-amir
talley.amir@yale.edu

James Aspnes
Yale University, New Haven, CT, USA
https://www.cs.yale.edu/homes/aspnes/
james.aspnes@gmail.com

David Doty
University of California, Davis, CA, USA
https://web.cs.ucdavis.edu/~doty/
doty@ucdavis.edu

Mahsa Eftekhari
University of California, Davis, CA, USA
https://eftekhari.cs.ucdavis.edu/
mhseftekhari@ucdavis.edu

Eric Severson
University of California, Davis, CA, USA
https://www.math.ucdavis.edu/~severson/
eseverson@ucdavis.edu

Abstract

The standard population protocol model assumes that when two agents interact, each observes the entire state of the other. We initiate the study of **message complexity** for population protocols, where an agent's state is divided into an externally-visible message and externally-hidden local state.

We consider the case of $O(1)$ message complexity. When time is unrestricted, we obtain an exact characterization of the stably computable predicates based on the number of internal states $s(n)$: If $s(n) = o(n)$ then the protocol computes semilinear predicates (unlike the original model, which can compute non-semilinear predicates with $s(n) = O(\log n)$), and otherwise it computes a predicate decidable by a nondeterministic $O(n \log s(n))$-space-bounded Turing machine. We then introduce novel $O(\text{polylog}(n))$ expected time protocols for junta/leader election and general purpose broadcast correct with high probability, and approximate and exact population size counting correct with probability 1. Finally, we show that the main constraint on the power of bounded-message-size protocols is the size of the internal states: with unbounded internal states, any computable function can be computed with probability 1 in the limit by a protocol that uses only *1-bit* messages.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases population protocol, message complexity, space-optimal

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.6

Related Version https://arxiv.org/abs/2003.09532

Supplementary Material https://archive.softwareheritage.org/browse/origin/directory/?origin_url=https://github.com/eftekhari-mhs/population-protocols

Funding The second author was supported by NSF award CCF-1650596. The third, fourth, and fifth authors were supported by NSF awards 1619343, 1844976, 1900931.

© Talley Amir, James Aspnes, David Doty, Mahsa Eftekhari, and Eric Severson;
licensed under Creative Commons License CC-BY
34th International Symposium on Distributed Computing (DISC 2020).
Editor: Hagit Attiya; Article No. 6; pp. 6:1–6:18
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Population protocols, introduced by Angluin, Aspnes, Diamadi, Fischer, and Peralta [6], are a class of algorithms that model ad hoc networks of finite-state mobile agents. At each step, a pair of agents is picked uniformly at random to interact, each observing the other's state and updating its own state in response. The original model [6] limited agents to $O(1)$ states, independent of the population size n. This limited the computational power (to only semilinear predicates [7]) and the time efficiency in performing fundamental tasks (e.g. the linear-time lower bound for leader election [28]). Recent work uses $\omega(1)$ states, yielding more time-efficient algorithms for fundamental tasks (e.g. [1–4, 15, 30–32, 41]). Is the improved performance a result of higher communication throughput or greater storage capacity?

Whereas the original model supposes that agents can view the entirety of the other's local state upon interacting, we introduce a new variant of this model which distinguishes a segment of the agent's state that is externally visible to its interacting partner, called the **message**. This variant generalizes previous work in the context of consensus that examines the particular case of **binary signaling** [5, 36], where the message is limited to a single bit. We study the computational power of population protocols that have $O(1)$ message complexity and varying local state complexity, ranging from $O(1)$ to unbounded.

1.1 Motivation

The population protocol framework was conceived to model passively mobile ad hoc sensor networks. In this setting the amount of communication bandwidth can be a tighter constraint than the local computation performed by a sensor. These two constraints – bandwidth efficiency and energy efficiency – are viewed as distinct in the networking literature. In some scenarios it makes more sense to optimize for one or the other, or to strike a balance [23,33,42]. The restriction to $O(1)$ messages but $\omega(1)$ internal states is germane when the communication in an interaction is more costly than the accompanying local computation.

Synthetic chemistry is another domain in which population protocols are an appropriate abstract model of computation. This is a subclass of chemical reaction networks, which are known to have similar computational power [21, 38]. Using a physical primitive known as **DNA strand displacement** [44], *every* chemical reaction network with $O(1)$ species (**states** in the language of population protocols) can be theoretically implemented by a set of DNA complexes [39], justifying the use of chemical reactions as an implementable programming language. This approach has been used to synthesize nontrivial chemical systems in the wet lab, resulting in pure DNA implementations of a chemical oscillator [40] and the "approximate majority" population protocol [9, 22]. Some theoretical [37] and experimental [43] systems are able to assemble unbounded-length heteropolymers, best modeled using arbitrarily many states (exponential in the polymer length) but only $O(1)$ messages rendering the smaller "locally visible region" near one or both ends of the polymer.

Finally, our model of $\omega(1)$ internal states and $O(1)$ external messages is a natural mathematical intermediate between the original $O(1)$-state model and the more recent $\omega(1)$-state model. Population protocols with superconstant states are provably more powerful [20], so it is intrinsically interesting to determine how powerful this new intermediate model is.

1.2 Our Contribution

Table 1 Summary of positive results: Above, the event of "not error" means that the answer is correct *and* the stated time and state bounds hold, unless the error probability is 0, in which case it refers only to the output being correct. In that case, the time and state bounds are in expectation, but still hold with high probability: in all cases, the probability of error is $O(1/n)$. (It should be straightforward to extend to $O(1/n^k)$ for any k, but for simplicity in proof statements we fix $k = 1$. We rely on [14, Theorem 1] that, as stated, holds only for a fixed exponent k, though it seems a more detailed analysis could achieve arbitrary k.) Note that when the probability of computing the correct output is 1 (i.e. the protocol stabilizes), the Time column denotes time to convergence. State complexities are accurate with high probability. $|M|$ is the number of messages, either a constant larger than 1, or exactly 2 (1-bit). Compute $\log n$ means computing either $\lfloor \log n \rfloor$ or $\lceil \log n \rceil$. In the first row t_P is the expected convergence time for P.

Problem solved	Pr[error]	Time	States	$\lvert M \rvert$	Leader
Simulate $s(n)$-state open protocol (Corollary 3.4)	0	$O(t_P n^2 \log s(n))$	$O(s(n)^2)$	$O(1)$	Yes
Junta election (Theorem 4.1)	> 0	$O(\log^2 n)$	$O(\log^2 n)$	1-bit	No
Compute n (Theorem 4.2)	> 0	$O(\log^2 n)$	$O(n \log^2 n)$	$O(1)$	Yes
Compute $\log n$ (Corollary 4.3)	> 0	$O(\log^2 n)$	$O(\log n)$	$O(1)$	Yes
Stably compute n (Corollary 4.4)	0	$O(\log^2 n)$	$O(n^4 \log^4 n)$	$O(1)$	Yes
Leaderlessly compute n (Corollary 4.5)	> 0	$O(\log^2 n)$	$O(n \text{ polylog}(n))$	$O(1)$	No
Leaderlessly compute $\log n$ (Corollary 4.5)	> 0	$O(\log^2 n)$	$O(\text{polylog}(n))$	$O(1)$	No
Compute d-input predicate (Corollary 4.6)	> 0	$O(d \log^2 n)$	$O(n^d \log^2 n)$	$O(1)$	Yes
TM simulation (Theorem 5.2)	0	unbounded	unbounded	1-bit	No

We introduce this new variant of population protocols and show three main results:

We first (Section 3) completely resolve the question of the computational power of $O(1)$ messages, with Theorem 3.2. In the positive direction, with poly(n) states ($O(\log n)$ bits), we give a simulation of $\Omega(1)$-bit messages (Theorems 3.3 and 3.5). Corollary 3.9 is an asymptotically sharp negative result: $O(1)$-message, $o(n)$-state ($\log n - \omega(1)$ bits) protocols compute only semilinear predicates.

Secondly (Section 4), we focus on time-efficient computation. We develop novel $O(\log^2 n)$-time algorithms for junta election (the key primitive to leader election) and exact population size counting (naturally suited to this model, where $O(n)$ local states and $O(1)$ messages are the minimal power to make this problem solvable). The counting protocol can specialize with fewer states to estimate the size (count $\log n$), and also generalize with more states to count the entire input configuration (so any predicate can be locally computed).

Thirdly (Section 5), we explore the extreme limits of the model where message complexity is limited to 1 bit. We construct a 1-bit broadcast primitive, showing it is powerful enough to simulate a Turing Machine with probability 1 correctness using unbounded local memory.

1.3 Comparison to existing work and new techniques required

Most protocols using $\omega(1)$ states [1–3, 12, 13, 16–19, 25, 30, 31, 34, 35, 41] crucially use $\omega(1)$-size messages. Key transitions in such protocols involve comparing two integers/ids of size $\omega(1)$ in a single step, which is not possible with $O(1)$-size messages. Sending a superconstant-

size message over multiple interactions is not efficient (though it is a trick we employ for unbounded time results such as Theorem 3.3), since there is not enough time for the two agents to wait for another interaction (which takes $\Theta(n)$ expected time), nor is there any way to distinguish each other in future interactions. We introduce new techniques that rely on timing of internal counters to get around this limitation.

Our *JuntaElection* protocol (see paper's extended version) is our primary fast leaderless protocol, used to make other leader-driven protocols leaderless. It elects a **junta**, a group of $O(\sqrt{n})$ agents, in $O(\log^2 n)$ time. As with many other existing protocols [15,30,31,41], this is used for a junta-driven **phase clock** [8] allowing agents to synchronize in a downstream computation. The cited protocols have agents choose an integer "level" ℓ, propagating by epidemic the maximum level ($\Theta(\log\log n)$ [15,30,31] as in our case, or $\Theta(\log n)$ [41]). Agents who chose the maximum level are in the junta. Lacking the ability to communicate the levels in 1-bit messages, we rely on **timing** of agents' internal counters to detect whether a higher level exists: Agents with level ℓ count up to $\approx 4^\ell$, (roughly) telling all other agents to continue counting and stop at $\approx 4^\ell$, unless another agent (with high probability with a higher level) tells them to continue counting. The actual details require intricate choice of timing and analysis to conclude that all agents stop at the same counter value with high probability. We push the technique of communication via timing further, showing that *1-bit* messages suffice to elect a leader, broadcast arbitrary messages, and simulate a Turing Machine.

The **majority** problem is that of deciding which of two opinions in a population is more numerous. One population protocol [5] distinguishes between external messages and internal state, using 1-bit messages (**binary signaling**) to achieve consensus in expected $O(nr\log nr)$ interactions; however, this protocol uses $O(r)$ internal states, where r is a tunable parameter independent of n and can thus be considered constant, making this a $O(1)$-state solution to the majority problem. Other existing protocols [2, 4, 12, 13, 18] use an $O(1)$-message "doubling/cancelling" technique, which works on top of a synchronization primitive, but these protocols use $\omega(1)$ messages to achieve synchronization. Our $O(1)$-message junta-election protocol can be composed with the doubling/cancelling technique to give a high-probability, $O(1)$-message majority protocol, which, unlike [2,4,13,18], is **uniform**, requiring no estimate of n.

Indeed, all of our protocols are uniform in this sense, in contrast to several existing $\omega(1)$-state protocols [1–4,13,16,18,19,31,34,35,41]. Many of our protocols could be simplified greatly by allowing nonuniformity. Briefly, an estimate of $\log n$ within a constant factor would allow agents to synchronize themselves using a **leaderless phase clock** based on counting to $c\log n$ for some large constant c. the lack of such synchronization is a major challenge in devising correct, efficient $O(1)$-message protocols.

2 Model

We write $\log n$ to denote $\log_2 n$, and $\ln n$ to denote $\log_e n$. The original population protocol model [6] involves a population of n **agents**, each of which holds a **state** in a **state space** Q. Interactions between agents update the states of both agents according to a **transition function** $\delta : Q \times Q \to Q \times Q$, where interactions are asymmetric: in each interaction, one of the agents is the **initiator** of the interaction, and one the **responder**.

We consider a refinement of the model in which the state of an agent is explicitly divided into an internal component that is not visible to other agents, and an external component that is. The internal component of the state is drawn from the state space I and the external component, or **message**, is drawn from a message space M. The set of states Q is the

Cartesian product $I \times M$. The transition function δ is modified to enforce the restriction that an agent in an interaction cannot observe the internal state of the other agent: δ is now a function from $Q \times M \times \{\mathsf{initiator}, \mathsf{responder}\}$ to Q. When an agent in state $q_1 = \langle i_1, m_1 \rangle$ initiates an interaction with an agent in state $q_2 = \langle i_2, m_2 \rangle$, the new states of the agents are given by $q_1' = \langle i_1', m_1' \rangle = \delta(q_1, m_2, \mathsf{initiator})$ and $q_2' = \langle i_2', m_2' \rangle = \delta(q_2, m_1, \mathsf{responder})$.

The set of producible states $Q(n)$ and the set of producible messages $M(n)$ can both depend on n. The function $s : \mathbb{N} \to \mathbb{N}$ defined as $s(n) = |Q(n)|$ is the **state complexity** of a population protocol. The function $n \mapsto |M(n)|$ is the **message complexity**. If $|I| = 1$ and each agent's state is merely defined by its message (the original model [6] and its superconstant state generalization), we say the protocol is **open**, so $|Q(n)| = |M(n)|$ for all n. We will mostly be interested in population protocols with modest state complexity (at most polynomial in n, and often only polylogarithmic in n) and constant message complexity. Given two functions $s, m : \mathbb{N} \to \mathbb{N}$, a **$s(n)$-state, $m(n)$-message population protocol** is one with state complexity s and message complexity m. Note that the complexity bounds we discuss are *worst-case*: $s(n)$ is the most number of states that can be produced in any population of size n under any execution. We place high probability bounds on the state complexity (e.g. *JuntaElection* where agents generate a geometric random variable which may take on any positive integer value). These do not consider the set of producible states, so our impossibility results (Theorem 3.8) on state and message complexity do not apply.

Problems solved by population protocols. A **configuration** gives the state of all agents. Population protocols have some problem-dependent notion of "correct" configurations. For example, for **leader election** a configuration with a single leader is correct. For computation of a **predicate** $\phi : \mathbb{N}^d \to \{\mathsf{yes}, \mathsf{no}\}$ (a.k.a., **decision problem**), the initial state of each agent is from a d-element subset Σ of states, states are partitioned into two subsets representing "yes" and "no", and a configuration is correct if all agents give the answer $\phi(\vec{i})$, where $\vec{i} \in \mathbb{N}^d$ represents the initial counts of agents in each state in Σ. A population protocol is **leader-driven** if its states have a Boolean field $\mathsf{leader} \in \{L, F\}$ (i.e. the state set $Q = \{L, F\} \times Q'$), such that in every valid initial configuration, exactly one agent has $\mathsf{leader} = L$.

Time complexity. For measuring time complexity, we assume **random scheduling**, where at each interaction two agents are chosen uniformly at random from all $n(n-1)$ possible ordered pairs of agents. Time complexity is defined by **parallel time**, the number of interactions divided by $n/2$ which we henceforth simply refer to as **time**. This definition reflects the average number of interactions in which an agent participates, and reflects an assumption that agents effectively interact in parallel, even though for simplicity of analysis this parallelism is modeled by interleaving interactions sequentially.

Convergence/stabilization. A configuration $\vec{c}$ is **stably correct** if every configuration reachable from $\vec{c}$ is correct. An execution $\mathcal{E} = (\vec{c}_0, \vec{c}_1, \ldots)$ is picked at random according to the scheduler explained above. We say $\mathcal{E}$ **converges** (respectively, **stabilizes)** at interaction $i \in \mathbb{N}$ if $\vec{c}_{i-1}$ is not correct (resp., stably correct) and for all $j \geq i$, $\vec{c}_j$ is correct (resp., stably correct). The **(parallel) convergence/stabilization time** of a protocol is the number of interactions to converge/stabilize, divided by $n/2$. Convergence can happen strictly before stabilization, although a protocol with finite reachability (i.e. for each $\vec{c}$, finitely many configurations are reachable from $\vec{c}$) converges from $\vec{c}$ with probability $p \in [0, 1]$ if and only if it stabilizes from $\vec{c}$ with probability p. For a computational task T equipped with some definition of "correct", we say that a protocol **stably computes T with probability p** if, with probability p, it stabilizes (equivalently, converges).[1]

[1] Let C and S respectively be the set of stabilizing and converging executions. Then $\Pr[C \setminus S] = 0$. Suppose a protocol converges in an execution $(\vec{c_0}, \vec{c_1}, \ldots)$ at interaction i. If it did not stabilize, then for

3 Computability with unrestricted time

In this section we study $s(n)$-state, $O(1)$-message protocols where time is not restricted. Theorem 3.2 is our main result in this section, which completely characterizes the power of such protocols in terms of the number of bits required to store the states.

Let $\mathsf{CMPP}(f(n))$ be the set of all predicates stably computed by an $s(n)$-state, $O(1)$-message population protocol, where $s(n) = 2^{O(f(n))}$ (using $O(f(n))$ bits of memory). Let $\mathsf{SNSPACE}(g(n))$ be the set of all predicates $\phi : \mathbb{N}^d \to \{0, 1\}$ decidable by a nondeterministic $O(g(n))$-space-bounded Turing machine, when inputs are given in unary.[2] The results of [20] considered a similar complexity class $\mathsf{PMSPACE}(f(n))$ of stably computable predicates using $O(f(n))$ bits of memory and $O(f(n))$ bit messages.[3] Let SL be the set of all **semilinear** predicates [7]. Their main result is the following characterization:

▶ **Theorem 3.1** ([20]). *Let $f : \mathbb{N} \to \mathbb{N}$. If $f(n) = o(\log \log n)$, then $\mathsf{PMSPACE}(f(n)) = \mathsf{SL}$. If $f(n) = \Omega(\log n)$, then $\mathsf{PMSPACE}(f(n)) = \mathsf{SNSPACE}(n \cdot f(n))$.*

Since the memory is expressed in Theorem 3.1 as number of *bits* (exponentially smaller than number of **states**), the multiplicative constants hidden in the Big-O notation become polynomial-factor terms in number of states. Theorem 3.2 is a similar dichotomy theorem for $O(1)$-message population protocols, which is sharper in that it holds for *all* values of $f(n)$. The proof is in the full paper version, and follows from Theorems 3.3, 3.5, 3.8, and 3.1.

▶ **Theorem 3.2.** *Let $f : \mathbb{N} \to \mathbb{N}$. If $f(n) = o(\log n)$, then $\mathsf{CMPP}(f(n)) = \mathsf{SL}$, otherwise $\mathsf{CMPP}(f(n)) = \mathsf{SNSPACE}(n \cdot f(n))$.*

3.1 Leader-driven $O(s(n)^2)$-state, $O(1)$-message protocols can simulate open $s(n)$-state protocols

In this section we show that $O(s(n)^2)$-state, $O(1)$-message, leader-driven protocols can simulate $s(n)$-state open protocols (whether leader-driven or not). Thus, allowing a leader and ignoring quadratic differences in state complexity, there is no difference whatsoever between the computational power of $O(1)$-message protocols and open protocols. Theorem 3.3 proves the general case of $m(n)$-message protocols. Corollary 3.4 is the special case of **open** protocols, where $s(n) = m(n)$. The simulation incurs a time slowdown of factor $n^2 \log m(n)$, where n is the population size and $m(n) \leq s(n)$ is the message complexity of the simulated protocol, so it ports (non-sublinear) computability results from the open protocol model.

Intuitively, the construction of Theorem 3.3 chooses two agents to "mark" as $\mathsf{initiator}$ and $\mathsf{responder}$, which then successively pass a bit string as they interact, until they have transmitted the full message of size $\log m(n)$ bits. Crucially, starting with a leader allows only one simulated transition to be taking place at a time.

all $j > i$, some incorrect configuration $\vec{d_j}$ would be reachable from $\vec{c_j}$. Let $p_j > 0$ denote the probability of reaching $\vec{d_j}$ from $\vec{c_j}$. The set of reachable configurations is bounded with probability 1, so $\min_{j>i} p_j$ is well-defined and positive. The probability of never reaching any $\vec{d_j}$ is then 0.

[2] In [20] these are called **symmetric** predicates on the assumption that the d counts in $\vec{i} \in \mathbb{N}^d$ are presented to the Turing machine as a $\|\vec{i}\|$-length string of symbols from an input alphabet Σ with $|\Sigma| = d$, with the same answer on all permutations of the string.

[3] In fact, to obtain their positive result for large space bounds, they do not need fully open protocols. Their simulation of nondeterministic $nf(n)$-space-bounded Turing machines just requires $O(\log n)$ bit messages to exchange unique IDs, even if $f(n) = \omega(\log n)$.

▶ **Theorem 3.3.** *For every $s(n)$-state, $m(n)$-message protocol P, there is a leader-driven, $O(s(n) \cdot m(n))$-state, $O(1)$-message protocol S that simulates P, and each interaction of P takes expected $O(n^2 \log m(n))$ interactions of S to simulate.*

The next corollary applies to **open** protocols, where each agent's message is its full state.

▶ **Corollary 3.4.** *For every $s(n)$-state, open population protocol P, there is a leader-driven, $O(s(n)^2)$-state, $O(1)$-message population protocol S that simulates P, and each interaction of P takes expected $O(n^2 \log s(n))$ interactions of S to simulate.*

It is known that $\Omega(\log n)$-state open protocols have computational power beyond that of $O(1)$-state protocols (limited to semilinear predicates [7] and functions [21]), and Corollary 3.4 grants this same computational power to leader-driven $O(1)$-message protocols. Theorem 3.8 in subsection 3.4 shows that Corollary 3.4 crucially depends on the assumption of an initial leader in the simulating protocol, by demonstrating that **leaderless** $O(1)$-message, $o(n)$-state protocols are no more powerful than $O(1)$-state open protocols.

3.2 Leader election can be composed with leader-driven, $s(n)$-state, $O(1)$-message protocols using $O(n^3 \log n)$ state overhead

Leader election is possible in linear time with 1-bit messages by "fratricide": $\ell, \ell \to \ell, f$. A downstream leader-driven protocol P will not work unaltered if composed with this leader election, because the presence of multiple leaders prior to convergence causes incorrect transitions of P. A straightforward fix using $O(n)$ messages involves exact size counting via transitions $\ell_i, \ell_j \to \ell_{i+j}, f_{i+j}$ (requiring $\Omega(n)$ messages) and each transition between agents with respective values n and $i < n$ resets the latter agent to its initial state in P. As soon as the last agent is reset with value n, the protocol faithfully executes a **tail** of an execution of P from $\vec{i}$, i.e. an execution starting at a configuration $\vec{c}$ reachable from $\vec{i}$. Thus if P is correct with probability 1, the composed protocol is also correct with probability 1. Theorem 3.5 demonstrates a similar "composition by resetting" strategy using $O(1)$ messages.

▶ **Theorem 3.5.** *For any leader-driven, $s(n)$-state, $O(1)$-message protocol P, there is a leaderless, $O(s(n)n^3 \log n)$-state, $O(1)$-message protocol S that, after $O(n \log n)$ expected time, executes a tail of an execution of P.*

Theorem 3.5 depends crucially on using $\geq n$ states, since Theorem 3.8 shows leaderless, $O(1)$-message, $o(n)$-state protocols are no more powerful than $O(1)$-state open protocols.

3.3 Deterministic Broadcast

The construction used in our composable leader election protocol (see extended paper) can be modified to also give the leader the ability to stably broadcast a message to the entire population. After the last restart, the leader agent a counts the entire population by moving them between phases. We can view these phases as deterministically synchronized rounds (each expected time $O(n \log n)$ [10]). Add a field $\mathsf{bit} \in \{0, 1\}$ to the message. The leader a can then communicate a bit string to the population by sending one bit during each round. This lets the population stably compute the population size n, by having the leader send n as a bit string in $O(\log n)$ rounds (stabilizing in expected $O(n \log^2 n)$ time). It uses $O(\log n)$ state overhead to store the bits it has broadcast, so $O(n^3 \log^2 n)$ states total. Thus we conclude:

▶ **Corollary 3.6.** *There is an $O(n^3 \log^2 n)$-state, $O(1)$-message protocol that stably computes the population size n (storing in every agents state), in expected $O(n \log^2 n)$ time.*

3.4 Leaderless $o(n)$-state, $O(1)$-message protocols compute only semilinear predicates

Theorem 3.8 is broad and does not apply to a particular “mode of computation” (e.g., deciding predicates [6, 7], computing functions [11, 21, 27], leader election [15, 29]). It does, however, assume a problem-specific notion of **valid** initial configurations.[4] We say a protocol is **additive** if the set of valid initial configurations is closed under addition. This rules out, for instance, protocols with an initial leader. Indeed, Corollary 3.9 is false if an initial leader is allowed, by applying Theorem 3.3 to let a leader-driven $O(1)$-message protocol simulate any $o(n)$-state open protocol that stably computes a non-semilinear predicate/function.[5]

A lower bound result in [20] shows that with an absolute space bound of $o(\log n)$ states, their model is limited to only stably computing the semilinear predicates.[6] The core of their argument bounds the number of reachable memory states.

▶ **Theorem 3.7** ([20]). *Let $s : \mathbb{N} \to \mathbb{N}$ and consider an additive, $s(n)$-state, open population protocol. Then either $s(n) = O(1)$ or $s(n) = \Omega(\log n)$.*

As a corollary, if $s(n) = o(\log n)$, then $s(n)$ is in fact constant, reducing to the original $O(1)$-state model, which can only stably compute semilinear predicates [7]. We use a similar proof technique to show an exponentially stronger result in the model of $O(1)$ messages.

▶ **Theorem 3.8.** *Let $s : \mathbb{N} \to \mathbb{N}$ and consider an additive, $s(n)$-state, $O(1)$-message population protocol. Then either $s(n) = O(1)$ or $s(n) = \Omega(n)$.*

Proof sketch. A fixed population $\vec{i}_c$ suffices to produce any of the $O(1)$ messages. Consider a population $\vec{i}_n$ of size n. If $s(n) \neq O(1)$, then for some state b not producible from $\vec{i}_n$, b **is** producible by sending some message m to a state a producible from $\vec{i}_n$ (though a and m cannot appear *simultaneously* in a configuration reachable from $\vec{i}_n$). By combining $\vec{i}_n$ with $\vec{i}_c$, we have a population of size $n + O(1)$ that can produce b. Thus the number of producible states grows at least linearly with n. ◀

Population protocols using $O(1)$ states compute only semilinear predicates [7], resulting in the following corollary. Since we require additivity of valid initial configurations, the corollary applies only to leaderless protocols.

▶ **Corollary 3.9.** *If a leaderless, $o(n)$-state, $O(1)$-message protocol stably computes a predicate ϕ, then ϕ is semilinear.*

4 Computability with polylogarithmic time complexity

In this section we study $O(1)$-message protocols with “fast” computation (polylog(n) time).

---

[4] For example, for leader election, all agents have the same initial state. For computation of predicates [6] or functions [11, 21], all agents represent “input” from a constant alphabet, with possibly an extra leader.

[5] For example, transitions $(i;\ell),(i;\ell) \to (i+1;\ell),(i+1;f)$ and $(j;\ell),(i;f) \to (j;\ell),(j;f)$, which starting from all agents in state $(1,\ell)$, give each agent the value $\lfloor \log n \rfloor$.

[6] Theorem 14 of [20] states “$o(\log\log n)$” bits, which implies $o(\log n)$ states, though the converse does not hold. However, inspecting their proof reveals that the result holds up to $\log(n) - 1$ states.

4.1 High-probability junta election using 1-bit messages

In this section, we describe a uniform protocol using 1-bit messages that, with high probability, elects a "junta" of $O(\sqrt{n})$ agents in polylogarithmic time. The protocol also lets each agent compute an integer $k \in \mathbb{N}^+$ that is the same for all agents and is one of $\lfloor \log \log n \rfloor$, $\lceil \log \log n \rceil$, or $\lceil \log \log n \rceil + 1$. Thus 2^k is an estimate of $\log n$ within a multiplicative factor 2.

Furthermore, *JuntaElection* is composable, in that we can use the protocol as a black box to initialize other protocols that require either a junta for a phase clock, or an approximation of $\log n$ (e.g. for a leaderless phase clock). Thus any nonuniform protocol that requires k-bit messages can be composed with *JuntaElection* to achieve a uniform protocol using $(k+1)$-bit messages with an additive time overhead of $O(\log^2 n)$. For example, we can compose *JuntaElection* with the the leader election protocol of [30] using $\frac{1}{2}$-coin flips to convert the $O(\sqrt{n})$-size junta to size 1, i.e. elect a leader, in expected $O(\log^2 n)$ time and $O(1)$ messages, or with majority protocols that use $O(1)$ messages for doubling/cancelling phases, synchronized by the junta-driven phase clock [12]. Our protocol has a positive probability of failure. It is yet unknown whether there is an $O(1)$-message protocol that stably approximates $\log n$ or elects a junta of size n^ϵ for some $\epsilon \in (0,1)$ in sublinear stabilization time.

High-level description of protocol. The protocol is described formally in the extended paper. Intuitively, it works as follows. Most leader/junta election protocols generate an **id**, where the agents generating the maximum id are the junta. In our protocol, we generate an id called **level**, but $O(1)$ messages prevent direct communication of levels, so we employ a timing-based strategy to communicate the maximum level using only messages `Go` and `Stop`.

Each agent initially generates a local geometric random variable G (number of fair coin flips until the first heads, i.e., an immediate heads results in $G = 1$) and computes its **level** as $\lceil \log G \rceil$. (We can also use synthetic coin techniques [1] to simulate fair coin flips and increment their level from i to $i+1$ as they flip 2^i consecutive tails.)

We define consecutive disjoint intervals $G_0, R_0, G_1, R_1, \ldots \subset \mathbb{N}$ (**green** and **red**) partitioning the natural number line. We call R_i's last element $d_i = \max R_i$ a **door**. (See Figure 1, formal definition below.) Each agent keeps a local counter, initially 0, that is incremented on some interactions. An agent is **in round** i if its counter is in $G_i \cup R_i$. The goal is to get every agent to count up until the round equal to the maximum level k generated by any agent and stop its counter at d_k. An agent with level l in round i is **eager** if $i < l$ and **cautious** otherwise. Intuitively, eager agents race through doors until their own level, telling all other agents to keep going, but become cautious at and beyond their own level, advancing past a door into the next round only if another agent tells them to do so (via a message $m = $ `Go`). More formally, an eager agent always sends a message of `Go` and increments its counter on every interaction. A cautious agent sends message `Go` if and only if its counter is in G_i for some i, increments its counter on every interaction in $G_i \cup R_i \setminus \{d_i\}$ unconditionally, and increments its counter beyond d_i if and only if the other agent's message is `Go`. Agents drop out of the junta when they leave their own level, so (assuming no agent leaves the maximum level) those who generated the maximum level are the eventual junta.

To formally define the intervals, let $c \in \mathbb{N}^+$. Each G_i, with $|G_i| = c4^i$, is called a **green** interval, R_i, with $|R_i| = \frac{3c}{2}4^i$, a **red** interval. Note that $d_i = \sum_{j=0}^{i-1}(|G_j| + |R_j|) = c\left(1 + \frac{3}{2}\right)\frac{4^i - 1}{4-1} < \frac{5c}{6}4^i = \frac{5}{6}|G_i|$, so $|G_i|$ is larger than the union of all the previous intervals by a constant multiplicative factor. The max level k is $\Theta(\log \log n)$ with high probability, so its corresponding interval $|G_k| = \Theta(4^{\log \log n}) = \Theta(\log^2 n)$. Thus, with high probability, the agents use $O(\log^2 n)$ states for their counters and stop at the door d_k after $O(\log^2 n)$ time.

To compose *JuntaElection* with a downstream protocol P, agents can simply restart P whenever they move beyond a d_i, and then wait to start simulating P until they reach the next d_{i+1} (Restarting is a common technique in distributed computing for composition and is not original to this paper, e.g., [30].) In the early stages of *JuntaElection*, the downstream protocol gets restarted many times, but eventually, all agents will move past d_{k-1}, after which they will restart the downstream protocol for the last time. The agents will all simultaneously be in the last interval $G_k \cup R_k$ before stopping at d_k. Thus all simulated downstream interactions of P will be between agents that agree on k.

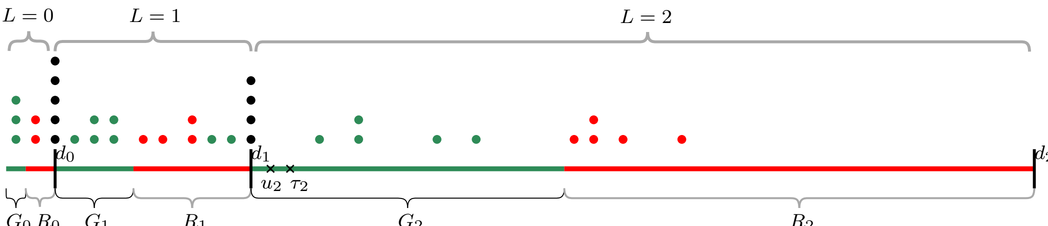


Figure 1 Agents, represented as dots, increment their counters through the $G_0, R_0, G_1, R_1, G_2, R_2$ intervals. Agents in green intervals or any interval before their own level have message `Go`. Agents in red intervals at their own level or later have message `Stop`. At the end of a red interval (the door d_i, shown with black horizontal line) at their own level or later, the agents (black dots) wait to increment their counter until they see a message `Go`. The special times marked u_i, τ_i are used in proving correctness.

▶ **Theorem 4.1.** *With probability $1 - O(1/n)$, JuntaElectionuses $O(\log^2 n)$ states and elects a junta of size $O(\sqrt{n})$ in $O(\log^2 n)$ time, after which v.*`count` *$= d_k$ for all agents v, where $k \in \{\lfloor \log\log n \rfloor, \lceil \log\log n \rceil, \lceil \log\log n \rceil + 1\}$.*

Proof sketch. We must show the agents are synchronized. When the interval lengths are $\Omega(\log n)$, we argue that the agents' local counters are bounded within the same interval. The main challenge is reasoning about agents that may be stuck at a door.

Our argument shows that a constant fraction $n/4$ of agents stay synchronized in each green interval, up until near the max level. Then, we argue that during the later green intervals, straggler agents are able to catch up, because they have a constant probability of passing through each door and the length of the green interval is more than the sum of all previous intervals. We then show the entire population in synchronized within the last few intervals. Thus all agents will have a `Stop` message when the population reaches the final door d_k, and the agents will stop their counters at d_k. See extended paper for full proof. ◀

Our proof techniques require setting $|G_i| = 700 \cdot 4^i$. However, simulation results show successful convergence when $|G_i| = 16 \cdot 2^i$ (see extended paper). Scaling the intervals this way lets $|G_M| = \Theta(\log n)$, so the protocol takes $O(\log n)$ time and $O(\log n)$ internal states.

4.2 Leader-driven, $O(\log^2 n)$-convergence-time exact size counting

In this section we give a $O(\log^2 n)$ time, high-probability protocol for a problem that is natural for agents with superconstant memory: exact population size counting. The probability of error can be reduced to 0 with standard techniques; see Corollary 4.4. This problem has been studied in the context of open protocols in both the exact [17, 26] and approximate [17, 25] settings, where it is known that open protocols can approximate n within multiplicative factor 2, by computing either $\lfloor \log n \rfloor$ or $\lceil \log n \rceil$, using $O(\log n \log\log n)$ states and $O(\log^2 n)$

time [17]. Open protocols can compute the **exact** value of n, using $O(n \log n \log \log n)$ states and $O(\log n)$ time [17]. Both protocols can be made probability-1, with a $O(\log n)$ factor increase in states, i.e. $O(\log^2 n \log \log n)$ states for calculating $\lfloor \log n \rfloor$ or $\lceil \log n \rceil$, and $O(n \log^2 n \log \log n)$ states for exactly computing n. However, note that our results below are leader-driven, so direct comparison with the leaderless results of [17] is not appropriate.

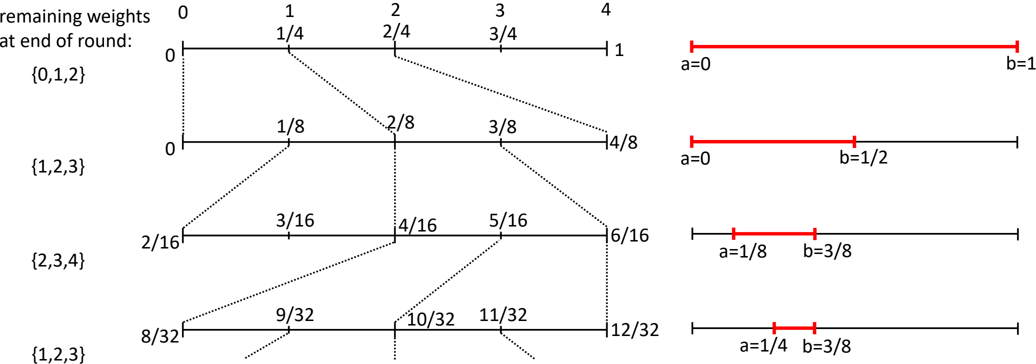


Figure 2 Update rule for fast exact counting protocol. All agents start with a mass of 0 and weight $w = 0$, except the leader, who starts with mass $= 1$ and $w = 4$. They conduct averaging on weight w for one round, at which point (with high probability) three consecutive weights remain. The figure shows how the remaining masses map to the next subinterval, with the weight w updating to $2(w - w_{\min})$ where $w_{\min}$ is the minimum value of w at the end of the *Averaging* phase. The right side shows the subintervals to scale. Each agent updates its internal state to represent the interval $[a, b]$. Once $[a, b]$ contains only a single number of the form $\frac{1}{n}$, the protocol terminates, and each agent knows the value n. The first $\log n$ rounds would always have 0 as the minimum remaining weights, but we allow other values to show concretely how the updating rule works.

▶ **Theorem 4.2.** *There is an $O(1)$-message leader-driven population protocol that, with probability $1 - O(1/n)$, exactly counts the population size n (storing it in each agent's internal state), in $O(\log^2 n)$ time and using $O(n \log^2 n)$ states.*

Proof sketch. The full proof and pseudocode for *ExactCounting* are in the extended paper. The proof uses the "fast averaging" technique employed by other population protocols [4, 17, 26, 34, 35], where each agent holds an integer and computes the transition $i, j \to \lfloor \frac{i+j}{2} \rfloor, \lceil \frac{i+j}{2} \rceil$. In the $O(1)$-message setting this will not work exactly as described.

Intuitively, the leader will distribute 1 unit of what we can imagine is a continuous mass into the population. Rational-valued averaging of this mass would result in each agent converging to $1/n$, from which n can be computed, $O(1)$ messages cannot represent arbitrary rationals. Instead, we allow agents to communicate a few bits of their number at a time, while ensuring that before moving on, they agree on an interval containing the true average, which shrinks by half each round, synchronized by a leader-driven phase clock [8]).

Figure 2 shows the updating rule. Each agent's state represents an interval $[a, b] \subseteq [0, 1]$, where $b - a = 2^{-r}$ during round $r \in \mathbb{N}$ (initialized to $r = 0$). a is a dyadic rational, initialized to $a = 0.0$, containing $r + 2$ bits after the binary point. One message field $W = \{0, 1, 2, 3, 4\}$ describes varying amounts of extra weight. The value $w \in W$ counts for $\frac{w}{4 \cdot 2^r}$ units of mass in round r. An agent is interpreted as having mass $= a + \frac{w}{4 \cdot 2^r} \in [a, b]$ (note representing $\frac{w}{4 \cdot 2^r}$ is what requires $r + 2$ bits after the binary point). The leader is initialized with $w = 4$ (and mass $= 0 + \frac{4}{4 \cdot 1} = 1$), and the followers are initialized with $w = 0$ (and mass $= 0$).

The full proof shows that, with high probability, every agent will always have the same value of a. This implies, via the averaging rule for weights, that mass is conserved and the sum of mass in the population is 1. Thus for all agents at all times, it holds that the true average $\frac{1}{n}$ stays within the interval $[a, b]$. Once the interval contains only a single integer reciprocal $\frac{1}{n}$, the protocol terminates with all agents knowing n. ◀

Terminating *ExactCounting* early gives a space efficient protocol for estimating $\log n$:

▶ **Corollary 4.3.** *A leader-driven, $O(1)$-message protocol, with probability $1 - O(1/n)$, computes $r \in \{\lfloor \log n \rfloor, \lceil \log n \rceil\}$, in $O(\log^2 n)$ time using $O(\log n)$ states.*

Proof sketch. We run *ExactCounting* until the interval $[a, b]$ contains exactly one power of two 2^{-k}, and then output k, unless it contains no powers of two, in which case we output arbitrarily either of the powers of 2 contained in the interval of the **previous** round. If $n = 2^k$, then $k = \log n$ exactly. Otherwise, since the interval contains no other power of 2, but it contains $1/n$, then $k \in \{\lfloor \log n \rfloor, \lceil \log n \rceil\}$. ◀

By the standard technique of running in parallel with a slower deterministic counting protocol, we can convert our exact counting protocol to have probability 0 of error while retaining fast convergence time (see extended paper for proof).

▶ **Corollary 4.4.** *There are $O(1)$-message, leader-driven population protocols that, with probability 1, respectively count the exact population size n and estimate it by computing $\lfloor \log n \rfloor$ or $\lceil \log n \rceil$, both with expected $O(\log^2 n)$ convergence time and $O(n \log^2 n)$ stabilization time. With probability $1 - O(1/n)$, they use $O(n^4 \log^4 n)$ and $O(\log^2 n)$ states, respectively.*

The next corollary shows that *ExactCounting*can be made leaderless by composing with the leader election protocol derived from *JuntaElection*. Proofs are in the extended paper.

▶ **Corollary 4.5.** *There is a leaderless, $O(1)$-message population protocol that exactly counts the population size n in $O(\log^2 n)$ time and $O(n \operatorname{polylog} n)$ states, succeeding with probability $1-O(1/n)$. There is also a leaderless, $O(1)$-message population protocol that computes $\lfloor \log n \rfloor$ or $\lceil \log n \rceil$ in $O(\log^2 n)$ time and $O(\operatorname{polylog} n)$ states, succeeding with probability $1 - O(1/n)$.*

4.3 Leader-driven, $O(\log^2 n)$-time predicate computation

We can use techniques from Theorem 4.2 to show how to compute, using a leader and with high probability, any predicate on a constant alphabet Σ, up to the space bounds allowed by the agents. We assume that there is one leader agent, and that every other agent has a state from a fixed alphabet Σ. Exactly the semilinear predicates are computable with probability 1 by $O(1)$-state open protocols [7] (with $> \log n$ states, more predicates are possible [20]).

▶ **Corollary 4.6.** *Let $d \in \mathbb{N}^+$ and let Σ be a d-symbol input alphabet. Then there is an $O(1)$ message leader-driven population protocol that, with probability $1 - O(1/n)$, exactly counts the input vector $\vec{i} \in \mathbb{N}^d$ (storing it in each agent's internal state), in $O(d \log^2 n)$ time and using $O(n^d \log^2 n)$ states.*

Proof sketch. Agents first run the exact counting protocol of Theorem 4.2 to store locally the value n. Agents then use a similar strategy to this protocol to count how many agents have input x for each symbol $x \in \Sigma$. Having now stored the entire initial population's input in their internal state, they can simply compute any computable predicate ϕ locally. ◀

If an agent can locally store the entire initial configuration, it can compute any predicate computable by the transition function δ. We required that δ be computable by a Turing Machine with $O(\log s(n))$ bits of memory, to make our model comparable with [20]. Thus we can compute all predicates computable by $O(\log n)$ bit space-bounded Turing Machines.

5 Computability with 1-bit messages

We will show that with 1-bit messages, it is possible to simulate a synchronous system that provides a 1-bit broadcast channel. This will be used to simulate more complex systems. We sacrifice stabilization for convergence and rely on unbounded counters to ensure convergence in the limit with probability 1. Let us begin by defining the simulated system.

A **synchronous broadcast system** consists of n synchronous agents that carry out a sequence of **rounds**. In a broadcast round, each agent generates a 1-bit outgoing message. These messages are combined using the OR function to produce the outcome for this round.

Broadcast operations can be used to detect conditions such as the presence of a leader, or ordinary message transmission if a unique agent is allowed to broadcast in a particular round. However, because broadcast operations are symmetric, they cannot be used for symmetry breaking. For the purpose of electing a leader, we assume that agents have the ability to flip coins; once we have a leader, further agents may be recruited for particular roles using an auxiliary protocol that allows the leader to select a single agent from the population in some round. The broadcast and selection protocols are mutually exclusive: either all agents participate in a broadcast in some round or all agents participate in selection. This is possible by showing that all agents eventually agree on the round number forever with probability 1.

Simulating this model in a population protocol requires (a) enforcing synchrony across agents, so that each agent updates its state consistently with the round structure; (b) implementing the broadcast channel that computes the OR of the agents' outputs; and (c) implementing the selection protocol. We show how to do this in the following section.

5.1 Implementing the core primitives

Broadcasts are implemented by epidemics that propagate 1 messages, separated by barrier phases in which all agents display 0. Selection is implemented by having the leader display a 1 to the first agent it meets. Both protocols depend on the number of steps at each agent being approximately synchronized with high probability; after $t(n/2)$ steps, all agents' step counts should be within the range $t \pm O(\sqrt{t \log n})$ with high probability; the time to carry out a broadcast is also $O(\log n)$ with high probability (see proofs in extended paper). By increasing the length of each round over time, the total probability across all rounds of an error occurring in either the broadcast or selection protocol due to out-of-sync agents or slow broadcasts converges to a finite value. Applying the Borel-Cantelli lemma then shows that there is a round after which no further failures occur with probability 1.

5.1.1 Details

Observe that the probability that a particular agent i participates in an interaction is exactly $2/n$, and that the events that i participates in distinct interactions are independent. If we let X_i^t be the indicator variable that agent i participates in the t-th interaction, then $S_i^t = \sum_{j=1}^{t} X_i^t$ is a sum of independent Bernoulli random variables, and obeys the Chernoff bound $\Pr\left[|S_i^t - \mu| > \mu\delta\right] < 2e^{-\mu\delta^2/3}$, where $\mu = \mathrm{E}\left[S_i^t\right] = 2t/n$ and $0 \le \delta \le 1$.

The execution of each agent is organized as a sequence of rounds, where each round r for $r = 1, 2, \ldots$ consists of exactly $5r^2$ steps. The first $2r^2$ steps will be a **barrier phase** during which the agent displays message 0 and updates its state during an interaction only by incrementing its step counter. The remaining $3r^2$ steps will be an **interaction phase** in which the agents may execute one of two protocols. In a **broadcast phase**, each agent will propagate an epidemic represented by message 1, recording if it observed such an epidemic and possibly initiating the epidemic itself if instructed to do so by the protocol. In a **selection phase**, a leader agent displays 1 for its first encounter, and the agent interacting with the leader receives a special mark. The choice of broadcast/selection phase is determined by the controlling protocol and is the same for all agents. As in a barrier phase, an agent in an interaction phase continues to update its step counter with each interaction.

The controlling protocol updates the state of the agent at the end of each round. Each agent v has a state $v.\mathsf{state}$ that is one of broadcasting (agent is initiating a broadcast of value 1) receiving (agent is waiting to detect a 1), received (agent has detected a 1), selecting (agent is attempting to select another agent), candidate (agent is a candidate for selection), selected (agent has been selected), or idle (agent has selected another agent and is now waiting for the end of the round). We assume the controlling protocol assigns consistent values to the agents in each phase: if one or more agents start in state broadcasting, the rest should start in state receiving; while if some agent starts in state selecting, the rest should start in state candidate. Pseudocode for this protocol and its proof of correctness are in the extended paper.

5.2 Convergent computation of arbitrary symmetric functions

Early rounds produce incorrect results, so we need an error-recovery mechanism. We describe a basic protocol for electing a leader and having it gather inputs from the other agents. This allows the leader to compute the output of an arbitrary symmetric function and broadcast it to the other agents. The protocol guarantees termination with probability 1 even in executions where some of the rounds exhibit errors in the underlying broadcast mechanism. By restarting the protocol when it terminates, we guarantee that the protocol eventually runs without errors, thus converging to the correct output.

Each agent v has a Boolean field $v.\mathsf{leader}$ that marks it as a leader (or candidate leader) and a field $v.\mathsf{processed}$ that marks whether it has reported its input $v.\mathsf{input}$ to the leader. Agents cycle through 7 rounds, where the round number is $r \bmod 7$, organized as follows:

Round 0 Any leader broadcasts 1. A non-leader that receives 0 sets its leader bit. This round allows recovery from states with no leaders.

Round 1 Any leader broadcasts 1 with probability 1/2. A leader that does not broadcast but receives a 1 clears its leader bit.

Round 2 Any agent that cleared its leader bit in the previous round broadcasts 1. This causes any remaining leaders that receive a 1 to restart the information-gathering protocol and causes any non-leaders that receive a 1 to clear their processed bits. Broadcasting a 1 in this round is also used by the leader to restart the protocol after completion.

Round 3 Any agent v with $v.\mathsf{processed} = 1$ broadcasts 1. This is used by the leader and other agents to detect unprocessed inputs.

Round 4 If a leader received a 1 in the previous round, and there is no transmission in progress from a non-leader agent, the leader executes a selection operation. The selected agent sets its processed bit and transmits its input if its processed bit is not already set. If the processed bit is set, the agent transmits nothing in the following two rounds.

Rounds 5 and 6 These are used to transmit either (a) one bit of a selected agent's input, or (b) one bit of the protocol output. In either case the bit is encoded as two bits using the convention $01 = 0$, $10 = 1$, $00 = \mathsf{stop}$. Note that the absence of a broadcast in both rounds

is interpreted as stop, which both allows a selected agent to signal it has already been processed and guarantees eventual termination after an agent finishes transmitting its input even if some of the broadcasts are garbled. Two agents may transmit simultaneously (e.g. if there are multiple surviving leaders), thus agents must be prepared to handle receiving 11. Either we can decide to have agents interpret it as a fixed value: $11 = 1$, or interpret it as a signal to restart the protocol by broadcasting a 1 in the next Round 2.

The protocol terminates when the leader has collected all inputs (detected by the absence of a signal in Round 3) and transmits the computed output to all agents (using Rounds 5 and 6 over however many iterations are needed). We assume that the computed output has finite length for any combination of inputs. After transmitting the output, the leader broadcasts a 1 in Round 2 to restart the information-gathering component of the protocol.

▶ **Lemma 5.1.** *In any execution with finite errors in the underlying broadcast protocol, with probability* 1*, the above protocol converges to a single leader and then restarts infinitely often.*

Lemma 5.1 is proven correct by demonstrating that, as defined, this protocol elects exactly one leader by Round 2 which correctly processes all non-leader agents with probability 1 by the end of Round 6. The full proof can be found in the paper's extended version.

Once the protocol restarts with a single leader, any subsequent error-free execution produces correct output. This follows from the fact that the leader collects the input from every agent exactly once. Since each agent records as its output the last output broadcast by the leader, this causes all agents to converge to holding the correct output with probability 1.

Because the leader has unbounded states, it can simulate an arbitrary Turing machine. This allows the output to converge to the value of any computable symmetric function. The restriction to symmetric functions follows from uniformity of the agents in the initial configuration, but can be overcome, assuming inputs include indexes. We have thus shown:

▶ **Theorem 5.2.** *For any computable symmetric function f, there is a population protocol using* 1*-bit messages and unbounded internal states that starts in an initial configuration where each agent i is distinguished only by its input x_i, that converges to having each agent holding output $f(x_1, \ldots, x_n)$.*

Our construction exploits the unbounded state at each agent to allow the leader to simulate the entire computation. While probability-1 convergence requires unbounded state in the limit (otherwise there is a nonzero probability that any round fails), it may be desirable to put off expanding the state as long as possible. In the extended paper, we argue that with some small tweaks, the construction can be adapted to distribute the contents of a Turing machine tape of s bits across all agents of the population as in [20], reducing the storage overhead at each agent for the Turing machine computation to $O(s/n + \log s)$ bits.

6 Open problems

Probability-1 computation. Many of our protocols have a positive probability of error. Common techniques for achieving zero error probability in $\omega(1)$-state protocols require $\omega(1)$ messages. Based on this, we conjecture that probability-1 leader election using $O(1)$ messages requires $\Omega(n)$ time to stabilize. This is known to hold for $O(1)$ states [28], though sublinear-time **convergence** is possible with $O(1)$ states [32].

Time lower bounds. A tool for time lower bounds (e.g. probability-1 leader election [1,28]) is a "density lemma" [1,24] showing that when the state complexity is $\leq \frac{1}{2} \log \log n$, all states appear in "large" count. This is false for $s(n) > \log \log n$, which is the key to the fastest

space-optimal leader election protocols [15,30,31]. A density lemma applies to the **messages** of $O(1)$-message protocols, no matter the state complexity (derivable from [25, Lemma 4.2]). Does this imply that $O(1)$-message leader election requires linear stabilization time?

Power of 1-bit messages with $O(1)$-states. $O(1)$-state **open** protocols can stably compute exactly the semilinear predicates [7]. Can all semilinear predicates be stably computed with 1-bit messages? A related question is whether there is a direct simulation of $O(1)$-message protocols by 1-bit message protocols (similar to Theorem 3.3).

Efficient predicate computation. Corollary 4.6 can be used to efficiently compute any computable predicate $\phi : \mathbb{N}^d \to \{0, 1\}$ but requires storing the entire initial configuration locally in each agent ($\Theta(n^d)$ states). Corollary 3.6 can be used to compute any computable predicate storing unique IDs in each agent ($O(n)$ states), but it is slow since communication is routed through a leader. What predicates can be computed time- *and* space-efficiently?

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.
2. Dan Alistarh, James Aspnes, and Rati Gelashvili. Space-optimal majority in population protocols. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2221–2239. SIAM, 2018.
3. Dan Alistarh and Rati Gelashvili. Polylogarithmic-time leader election in population protocols. In *International Colloquium on Automata, Languages, and Programming*, pages 479–491. Springer, 2015.
4. Dan Alistarh, Rati Gelashvili, and Milan Vojnović. Fast and exact majority in population protocols. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, pages 47–56, 2015.
5. Dana Angluin, James Aspnes, and Dongqu Chen. A population protocol for binary signaling consensus. Technical Report YALEU/DCS/TR-1527, Yale University Department of Computer Science, 2016.
6. Dana Angluin, James Aspnes, Zoë Diamadi, Michael J. Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed Computing*, pages 235–253, March 2006.
7. Dana Angluin, James Aspnes, and David Eisenstat. Stably computable predicates are semilinear. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Principles of Distributed Computing*, PODC '06, page 292–299, New York, NY, USA, 2006. Association for Computing Machinery. doi:10.1145/1146381.1146425.
8. Dana Angluin, James Aspnes, and David Eisenstat. Fast computation by population protocols with a leader. *Distributed Computing*, 21(3):183–199, September 2008.
9. Dana Angluin, James Aspnes, and David Eisenstat. A simple population protocol for fast robust approximate majority. *Distributed Computing*, 21(2):87–102, July 2008.
10. James Aspnes, Joffroy Beauquier, Janna Burman, and Devan Sohier. Time and Space Optimal Counting in Population Protocols. 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 13:1–13:17, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.OPODIS.2016.13.
11. Amanda Belleville, David Doty, and David Soloveichik. Hardness of Computing and Approximating Predicates and Functions with Leaderless Population Protocols. In *ICALP 2017:*

44th International Colloquium on Automata, Languages, and Programming, volume 80, pages 141:1–141:14, 2017.

12 Petra Berenbrink, Robert Elsässer, Tom Friedetzky, Dominik Kaaser, Peter Kling, and Tomasz Radzik. Majority & stabilization in population protocols. *arXiv preprint arXiv:1805.04586*, 2018.

13 Petra Berenbrink, Robert Elsässer, Tom Friedetzky, Dominik Kaaser, Peter Kling, and Tomasz Radzik. A population protocol for exact majority with $O(\log^{5/3} n)$ stabilization time and $\Theta(\log n)$ states. In *32nd International Symposium on Distributed Computing (DISC 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

14 Petra Berenbrink, Tom Friedetzky, Dominik Kaaser, and Peter Kling. Tight & simple load balancing. In *2019 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2019, Rio de Janeiro, Brazil, May 20-24, 2019*, pages 718–726. IEEE, 2019.

15 Petra Berenbrink, George Giakkoupis, and Peter Kling. Optimal time and space leader election in population protocols. In *STOC 2020: 52nd Annual ACM Symposium on Theory of Computing*, 2020. to appear.

16 Petra Berenbrink, Dominik Kaaser, Peter Kling, and Lena Otterbach. Simple and efficient leader election. In *1st Symposium on Simplicity in Algorithms (SOSA 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

17 Petra Berenbrink, Dominik Kaaser, and Tomasz Radzik. On counting the population size. In Peter Robinson and Faith Ellen, editors, *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*, pages 43–52. ACM, 2019. doi:10.1145/3293611.3331631.

18 Andreas Bilke, Colin Cooper, Robert Elsässer, and Tomasz Radzik. Brief announcement: Population protocols for leader election and exact majority with $O(\log^2 n)$ states and $O(\log^2 n)$ convergence time. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 451–453, 2017.

19 Janna Burman, David Doty, Thomas Nowak, Eric E Severson, and Chuan Xu. Efficient self-stabilizing leader election in population protocols. *arXiv preprint arXiv:1907.06068*, 2019.

20 Ioannis Chatzigiannakis, Othon Michail, Stavros Nikolaou, Andreas Pavlogiannis, and Paul G. Spirakis. Passively mobile communicating machines that use restricted space. In *Proceedings of the 7th ACM ACM SIGACT/SIGMOBILE International Workshop on Foundations of Mobile Computing*, FOMC '11, page 6–15, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/1998476.1998480.

21 Ho-Lin Chen, David Doty, and David Soloveichik. Deterministic function computation with chemical reaction networks. *Natural Computing*, 13(4):517–534, 2014. Special issue of invited papers from DNA 2012. doi:10.1007/s11047-013-9393-6.

22 Yuan-Jyue Chen, Neil Dalchau, Niranjan Srinivas, Andrew Phillips, Luca Cardelli, David Soloveichik, and Georg Seelig. Programmable chemical controllers made from DNA. *Nature Nanotechnology*, 8(10):755–762, 2013.

23 L. H. Diakite and L. Yu. Energy and bandwidth efficient wireless sensor communications for improving the energy efficiency of the air interface for wireless sensor networks. In *2013 IEEE Third International Conference on Information Science and Technology (ICIST)*, pages 1426–1429, March 2013. doi:10.1109/ICIST.2013.6747805.

24 David Doty. Timing in chemical reaction networks. In *SODA 2014: Proceedings of the 25th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 772–784. SIAM, 2014.

25 David Doty and Mahsa Eftekhari. Efficient size estimation and impossibility of termination in uniform dense population protocols. In Peter Robinson and Faith Ellen, editors, *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*, pages 34–42. ACM, 2019. doi:10.1145/3293611.3331627.

26 David Doty, Mahsa Eftekhari, Othon Michail, Paul G. Spirakis, and Michail Theofilatos. Brief announcement: Exact size counting in uniform population protocols in nearly logarithmic time. In *DISC 2018: 32nd International Symposium on Distributed Computing*, 2018.

27 David Doty and Monir Hajiaghayi. Leaderless deterministic chemical reaction networks. *Natural Computing*, 14(2):213–223, 2015. Preliminary version appeared in DNA 2013.

28 David Doty and David Soloveichik. Stable leader election in population protocols requires linear time. *Distributed Computing*, 31(4):257–271, 2018. Special issue of invited papers from DISC 2015.

29 R. Elsässer and T. Radzik. Recent results in population protocols for exact majority and leader election. *Bulletin of the EATCS*, 126, 2018. URL: `http://bulletin.eatcs.org/index.php/beatcs/article/view/549/546`.

30 Leszek Gasieniec and Grzegorz Stachowiak. Fast space optimal leader election in population protocols. In *SODA*, pages 2653–2667, 2018. `doi:10.1137/1.9781611975031.169`.

31 Leszek Gasieniec, Grzegorz Stachowiak, and Przemyslaw Uznanski. Almost logarithmic-time space optimal leader election in population protocols. In *SPAA*, pages 93–102, 2019. `doi:10.1145/3323165.3323178`.

32 Adrian Kosowski and Przemysław Uznański. Brief announcement: Population protocols are fast. In *PODC*, pages 475–477, 2018. URL: `https://dl.acm.org/citation.cfm?id=3212788`.

33 R. Lu, X. Lin, H. Zhu, X. Liang, and X. Shen. Becan: A bandwidth-efficient cooperative authentication scheme for filtering injected false data in wireless sensor networks. *IEEE Transactions on Parallel and Distributed Systems*, 23(1):32–43, January 2012. `doi:10.1109/TPDS.2011.95`.

34 Y. Mocquard, E. Anceaume, and B. Sericola. Optimal proportion computation with population protocols. In *2016 IEEE 15th International Symposium on Network Computing and Applications (NCA)*, pages 216–223, October 2016.

35 Yves Mocquard, Emmanuelle Anceaume, James Aspnes, Yann Busnel, and Bruno Sericola. Counting with population protocols. In *14th IEEE International Symposium on Network Computing and Applications*, pages 35–42, 2015.

36 E. Perron, D. Vasudevan, and M. Vojnovic. Using three states for binary consensus on complete graphs. In *IEEE INFOCOM 2009*, pages 2527–2535, April 2009. `doi:10.1109/INFCOM.2009.5062181`.

37 Lulu Qian, David Soloveichik, and Erik Winfree. Efficient Turing-universal computation with DNA polymers. In *DNA 2010: Proceedings of The Sixteenth International Meeting on DNA Computing and Molecular Programming*, volume 6518 of *Lecture Notes in Computer Science*. Springer, 2010.

38 D. Soloveichik, M. Cook, E. Winfree, and J. Bruck. Computation with finite stochastic chemical reaction networks. *Natural Computing*, 7:615–633, 2008.

39 David Soloveichik, Georg Seelig, and Erik Winfree. DNA as a universal substrate for chemical kinetics. In *DNA14*, pages 57–69, 2008. `doi:10.1007/978-3-642-03076-5_6`.

40 Niranjan Srinivas, James Parkin, Georg Seelig, Erik Winfree, and David Soloveichik. Enzyme-free nucleic acid dynamical systems. *Science*, 358(6369):eaal2052, 2017.

41 Yuichi Sudo, Fukuhito Ooshita, Taisuke Izumi, Hirotsugu Kakugawa, and Toshimitsu Masuzawa. Logarithmic expected-time leader election in population protocol model. In *International Symposium on Stabilizing, Safety, and Security of Distributed Systems*, pages 323–337. Springer, 2019.

42 P. Udayakumar, R. Vyas, and O. P. Vyas. Energy efficient election protocol for wireless sensor networks. In *2013 International Conference on Circuits, Power and Computing Technologies (ICCPCT)*, pages 1028–1033, March 2013. `doi:10.1109/ICCPCT.2013.6529026`.

43 Damien Woods, David Doty, Cameron Myhrvold, Joy Hui, Felix Zhou, Peng Yin, and Erik Winfree. Diverse and robust molecular algorithms using reprogrammable DNA self-assembly. *Nature*, 567:366–372, 2019. `doi:10.1038/s41586-019-1014-9`.

44 Bernard Yurke, Andrew J. Turberfield, Allen P. Mills, Jr., Friedrich C. Simmel, and Jennifer L. Nuemann. A DNA-fuelled molecular machine made of DNA. *Nature*, 406:605–608, 2000.

Distributed Computation with Continual Population Growth

Da-Jung Cho
University of Kassel, Germany
https://www.dajungcho-toc.com
dajung.cho@uni-kassel.de

Matthias Függer
CNRS, LSV, ENS Paris-Saclay, Université Paris-Saclay, Inria, Gif-sur-Yvette, France
http://www.lsv.fr/~mfuegger/
mfuegger@lsv.fr

Corbin Hopper
ENS Paris-Saclay, Gif-sur-Yvette, France
Université Paris-Saclay, CNRS, Orsay, France
corbin.hopper@mail.mcgill.ca

Manish Kushwaha
Université Paris-Saclay, INRAE, AgroParisTech, Micalis Institute, Jouy-en-Josas, France
https://scholar.google.com/citations?user=JsXHBggAAAAJ
manish.kushwaha@inrae.fr

Thomas Nowak
Université Paris-Saclay, CNRS, Orsay, France
https://www.lri.fr/~nowak/
thomas.nowak@lri.fr

Quentin Soubeyran
École polytechnique, Institut Polytechnique de Paris, Route de Saclay, Palaiseau, France
Université Paris-Saclay, CNRS, Orsay, France
quentin.soubeyran.16@polytechnique.org

Abstract

Computing with synthetically engineered bacteria is a vibrant and active field with numerous applications in bio-production, bio-sensing, and medicine. Motivated by the lack of robustness and by resource limitation inside single cells, distributed approaches with communication among bacteria have recently gained in interest. In this paper, we focus on the problem of population growth happening concurrently, and possibly interfering, with the desired bio-computation. Specifically, we present a fast protocol in systems with continuous population growth for the majority consensus problem and prove that it correctly identifies the initial majority among two inputs with high probability if the initial difference is $\Omega(\sqrt{n \log n})$ where n is the total initial population. We also present a fast protocol that correctly computes the NAND of two inputs with high probability. We demonstrate that combining the NAND gate protocol with the continuous-growth majority consensus protocol, using the latter as an amplifier, it is possible to implement circuits computing arbitrary Boolean functions.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases microbiological circuits, majority consensus, birth-death processes

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.7

Related Version The full version with proofs is available at https://arxiv.org/abs/2003.09972.

Acknowledgements We acknowledge support from the Digicosme working group HicDiesMeus, Ile-de-France region's DIM-RFSI, INRAE's MICA department, and the CNRS project ABIDE. We thank Joel Rybicki for feedback on an earlier version.

© Da-Jung Cho, Matthias Függer, Corbin Hopper, Manish Kushwaha, Thomas Nowak, and Quentin Soubeyran;
licensed under Creative Commons License CC-BY
34th International Symposium on Distributed Computing (DISC 2020).
Editor: Hagit Attiya; Article No. 7; pp. 7:1–7:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

In the past few decades, synthetic biology has laid considerable focus on the re-programming of cells as computing machines. They have been engineered to sense a range of inputs (metabolites [25, 32], light [35], oxygen [1], pH [31]) and process them to produce desired outputs according to defined processing codes (primarily digital [11, 20], but occasionally analog [8]). Some potential applications of the cellular machines include production of metabolic compounds of interest [24], bio-remediation of toxic environments [37], sensing of disease bio-markers [32], and therapeutic intervention by targeted effector delivery [1]. Yet, the ability of single cells to reliably process multiple inputs is acutely constrained by their limited resources.

Adding too many processes into one cell leads to resource-stress and eventually the code is lost due to mutation, a baseline error mechanism present in all living systems. This has, in part, encouraged the notion of distributing the computational tasks across multiple cells [27, 36], to reduce resource-stress and improve robustness. The value of the idea is corroborated by the success of the division of labor seen in multi-cellular organisms that have naturally evolved from their unicellular ancestors [17, 26]. While task-distribution in cell populations solves some problems, it immediately leads to other challenges that must be tackled for the successful implementation of any complex distributed program. Some of these challenges include: the orthogonality/specificity of communication signals, the rate and bandwidth of communication channels, cellular growth and its effect on signal amplification or dissipation, and effect of cross-talk between different signals.

In this work we focus on amplification and Boolean function computation in distributed systems whose agents are duplicating bacteria. A central problem in this setting is to maintain a consistent state of circuit values among the bacteria, a problem that has been studied in distributed computing for decades in different contexts [18]. Starting from a mathematical computing model, analysis of a system's behavior has led to correctness proofs and performance bounds of proposed solutions, also shedding light on how protocol parameters influence the quality of the outcome. In distributed systems with biological agents, the cellular behavior is usually expressed in the language of chemical reaction networks (CRNs). A CRN is defined by a set of reactions, each consuming members of one or several species and producing members of others at a given rate.

The two most commonly used kinetics for CRNs are deterministic and stochastic approaches. The deterministic approach models the kinetics of a CRN as systems of ordinary differential equations (ODEs) with continuous real-valued concentrations of each species, whereas the stochastic approach models the CRN as a continuous-time Markov chain with discrete integer-valued counts of each species. While ODE modeling can capture important behavioral characteristics, in particular expected-value large-population limits, some phenomena can only be explained by stochastic-process kinetics. In particular, ODE kinetics cannot elucidate the probability of certain population-level events occurring in a system of two competing species, e.g., the extinction of one species due to a series of random events. The stochastic-process kinetics of CRNs are much more common in distributed computing, in particular in population protocols [3], where reactions are restricted to two reactants and two products with constant size populations, but also in computability results in more general CRNs [7, 33].

Consistent cell states by competition among cells. Competition among species naturally lends itself to solving consensus-type problems. Angluin *et al.* [3] analyzed a population protocol with three states: A, B, and blank. Encounters of opposing species A and B lead

to one of them becoming blank, and blank species that encounter a non-blank species copy its state. The population protocol by Angluin *et al.* [2] alternates phases of state duplication and cancellation, separated by a clock signal generated by a dedicated leader species. These protocols, however, rely on constant size populations and the latter on a dedicated leader, rendering them impractical for implementations in bacterial cultures.

Birth-death processes track species counts within a population with "birth" and "death" events over time. For each such population state there are transitions that move from one population state to the other with respect to "birth" and "death" events. Birth-death processes have been used to model competition, predation, or infection in evolutionary biology, ecology, genetics, and queueing theory [22, 30].

An early mention of problems requiring a stochastic analysis of two competing species is by Volterra [38] and Feller [10] although only the growth of a single species is analyzed therein. For an overview of single species birth-death Markov chains, see, e.g., [5]. Extensions for multiples species, with applications to genetic mutations, are found in the literature on competition and branching processes [4, 16, 28]. For example, Ridler-Rowe [29] considers a stochastic process between two competing species. However, the process in that work differs from ours in that death reactions are $A + B \longrightarrow A$ and $A + B \longrightarrow B$, leaving a winner after an encounter between two competing individuals. The paper presents an approximation for long-term distributions and bounds the probability that starting from initial A, B sizes, species A goes extinct. However, the analysis is for initial population sizes approaching infinity, only, and assumes an initial gap between species counts that is linear in the population size. By contrast our analysis holds for finite population sizes n, and requires a gap of $\Omega(\sqrt{n \log n})$, only. A complementary approach for the same asymmetric process proposed in [13] is to numerically solve a finite size cut-off of the infinite linear equation systems.

Computation in birth systems. In this work, we introduce and study *protocols for birth systems* where all species inherently duplicate. Such protocols are thus different from population protocols, which have population sizes that remain constant over the course of an execution. Further, our protocols do not rely on exact species counts, they are not leader-based, and they require small and constant state space per cell, lending themselves readily for future biological implementation.

For simplicity we assume that all duplication reactions of our birth systems have the same rate. We leave the question of natural selection due to differing growth rates to future work. In particular, we study two protocols within birth systems.

(i) We introduce the A-B protocol for two species A and B and show that it solves majority consensus with high probability: If the initial difference between A and B sizes Δ grows weakly with the population size n according to $\Delta = \Omega(\sqrt{n \log n})$, then the protocol identifies the initial majority with high probability. Since it amplifies the difference between the two species, we also refer to the A-B protocol as an *amplifier*. Further, we will show that the protocol reaches consensus in expected constant time. The protocol's reactions are deceptively simple. Besides the obligatory birth reactions $A \longrightarrow 2A$ and $B \longrightarrow 2B$, it comprises a single death reaction $A + B \longrightarrow \emptyset$.

(ii) We demonstrate how to implement the components of *feed-forward Boolean circuits*. Each Boolean gate in our implementation is a NAND gate, followed by an amplifier. Note that while we focus on the universal NAND gate for the sake of a lighter notation, our construction and its analysis holds for any arbitrary two-input Boolean function. The latter will be important for optimization and follow-up with biological implementations.

Signals between the NAND gates are encoded using two species each, the difference of which determines whether a signal is a logical 0, 1, or neither. A NAND gate is a protocol that maps two input signals X and Y to an output signal Z that is the logical NAND of X and Y.

While NAND gates are used to implement the circuit's Boolean behavior, the successive amplifiers regenerate the gate's output signal by amplifying the difference between the two output signal species. Repeated, successive invocation of the NAND protocol followed by the amplifier protocol for time $O(\log n)$, where n is the total initial population, can finally be used to compute the circuit's output values layer by layer.

Organization. The rest of the paper is organized as follows: In Section 2, we define the computational model. In Section 3, we introduce and analyze our protocol for majority consensus. In Section 4, we define and analyze the NAND gate protocol. In Section 5, we present simulations of the A-B protocol as well as a biologically plausible implementation of the NAND gate with amplifiers. Finally, Section 6 summarizes our results.

2 Model

We write $\mathbb{N} = \{0, 1, \dots\}$, $\mathbb{N}^+ = \mathbb{N} \setminus \{0\}$, and $\mathbb{R}_0^+ = [0, \infty)$. When analyzing our protocols, we employ the term "with high probability" relative to the total initial population. That is, event E happens with high probability if there exists some $c > 0$ such that $\mathbb{P}(E) = 1 - O(1/n^c)$, where n is the total initial population.

2.1 Chemical Reaction Networks

We use the standard stochastic kinetics for chemical reaction networks. A reader familiar with the model can safely skip this subsection.

A *chemical reaction network* is described by a set $\mathcal{S}$ of species and a set of reactions. A *reaction* is a triple $(\mathbf{r}, \mathbf{p}, \alpha)$ where $\mathbf{r}, \mathbf{p} \in \mathbb{N}^{\mathcal{S}}$ and $\alpha \in \mathbb{R}_0^+$. The species with positive count in $\mathbf{r}$ are called the reaction's *reactants* and this with positive count in $\mathbf{p}$ are called its *products*. The parameter α is called the reaction's *rate constant*. A *configuration* of a CRN is simply an element of $\mathbb{N}^{\mathcal{S}}$. A reaction $(\mathbf{r}, \mathbf{p}, \alpha)$ is *applicable* to configuration $\mathbf{c}$ if $\mathbf{r}(S) \le \mathbf{c}(S)$ for all $S \in \mathcal{S}$. We will write $\mathbf{r} \xrightarrow{\alpha} \mathbf{p}$ to denote a reaction $(\mathbf{r}, \mathbf{p}, \alpha)$. For instance, the reaction $(\{A, B\}, \{2B, C\}, \alpha)$ will simply be denoted $A + B \xrightarrow{\alpha} 2B + C$. Here, we used the shorthand notations $\{A, B\}$ and $\{2B, C\}$ for functions $\mathcal{S} \to \mathbb{N}$. For instance, the notation $\{2B, C\}$ represents the function $\mathbf{p} : \mathcal{S} \to \mathbb{N}$ defined by $\mathbf{p}(B) = 2$, $\mathbf{p}(C) = 1$, and $\mathbf{p}(S) = 0$ for all other species $S \notin \{B, C\}$.

The *stochastic kinetics* of a CRN are a continuous-time Markov chain (see a textbook [5] for auxiliary definitions). Given some volume $v \in \mathbb{R}_0^+$, which we will normalize to $v = 1$, the propensity of a reaction $(\mathbf{r}, \mathbf{p}, \alpha)$ in configuration $\mathbf{c}$ is equal to $\frac{\alpha}{v} \prod_{S \in \mathcal{S}} \binom{\mathbf{c}(S)}{\mathbf{r}(S)}$, where $\binom{\mathbf{c}(S)}{\mathbf{r}(S)}$ denotes the binomial coefficient of $\mathbf{c}(S)$ and $\mathbf{r}(S)$. The binomial coefficient is 1 if $\mathbf{r}(S) = 0$, i.e., if the species S is not a reactant of the reaction. It is 0 if $\mathbf{r}(S) > \mathbf{c}(S)$. The propensity of a non-applicable reaction is thus 0. For example, the propensity of reaction $A + B \xrightarrow{\alpha} 2B + C$ in configuration $\mathbf{c}$ is equal to $\frac{\alpha}{v} \cdot \mathbf{c}(A) \cdot \mathbf{c}(B)$. The propensity of $A \xrightarrow{\gamma} 2A$ is equal to $\frac{\gamma}{v} \cdot \mathbf{c}(A)$. The new configuration after an applicable reaction is equal to $\mathbf{c}' = \mathbf{c} - \mathbf{r} + \mathbf{p}$.

We will use the notation $Q(x, y)$ for the propensity of the transition from state x to state y in a continuous-time Markov chain. To each continuous-time Markov chain corresponds a discrete-time Markov chain that only keeps track of the sequence of state changes, but not of their timing. We will write $P(x, y)$ for the transition probability from state x to state y in the discrete-time chain. We have the formula $P(x, y) = Q(x, y) / \sum_z Q(x, z)$.

2.2 Birth Systems

A *protocol for a birth system*, or protocol, with input species $\mathcal{I}$ and output species $\mathcal{O}$, for finite, not necessarily disjoint, sets $\mathcal{I}$ and $\mathcal{O}$ is a CRN specified as follows. Its set of species $\mathcal{S}$ comprises input/output species $\mathcal{I} \cup \mathcal{O}$ and a finite set of internal species $\mathcal{L}$. Further, the protocol defines the initial species counts X_0 for internal and output species $X \in \mathcal{L} \cup \mathcal{O}$ and a finite set of reactions $\mathcal{R}$ on the species in $\mathcal{S}$. For each species $X \in \mathcal{S}$, there is a duplication reaction of the form $X \xrightarrow{\gamma} 2X$. All duplication reactions have the same rate constant $\gamma > 0$.

Given a protocol and an initial species count for its inputs, an execution of the protocol is given by the stochastic process of the CRN with species $\mathcal{S}$, reactions $\mathcal{R}$, and respective initial species counts.

3 Majority Consensus

The A-B protocol is defined for two species, A and B, both of which are inputs and outputs. It contains, apart from the obligatory duplication reactions, the single reaction of A and B eliminating each other with rate constant $\delta > 0$. The complete list of reactions of the A-B protocol is thus:

$$A \xrightarrow{\gamma} 2A \qquad B \xrightarrow{\gamma} 2B \qquad A + B \xrightarrow{\delta} \emptyset$$

We say that *consensus* is reached if one of the two species becomes extinct. If the initial population counts differ, we say that *majority consensus* is reached if consensus is reached and the species that was initially in majority is not extinct. If the initial counts of both species are equal, then majority consensus is reached when exactly one species is extinct.

We show that the A-B protocol reaches consensus in constant time and majority consensus with high probability.

▶ **Theorem 1.** *For initial population $n = A(0) + B(0)$ and initial gap $\Delta = |A(0) - B(0)|$, the A-B protocol reaches consensus in expected time $O(1)$ and in time $O(\log n)$ with high probability. It reaches majority consensus with probability $1 - e^{-\Omega(\Delta^2/n)}$.*

From Theorem 1 we immediately obtain a bound on the initial gap sufficient for majority consensus with high probability.

▶ **Corollary 2.** *For initial population n and initial gap Δ, if $\Delta = \Omega\left(\sqrt{n \log n}\right)$, then the A-B protocol reaches majority consensus with high probability.*

Without duplication reactions, it is obvious that the A-B protocol reaches consensus and that majority consensus is always reached if the two species have different initial population counts. We are thus not only able to show that we can achieve majority consensus in spite of continual population growth via duplication reactions of all species, but also that a sub-linear gap in the initial population counts suffices. The required initial gap of $\Omega(\sqrt{n \log n})$ matches that of the best protocols without obligatory duplications [3, 6].

We will prove Theorem 1 in the following sections; first the time upper bound, then correctness with high probability.

3.1 Markov-Chain Model

The A-B protocol evolves as a continuous-time Markov chain with state space $S = \mathbb{N}^2$. Its state-transition rates are $Q\big((A, B), (A + 1, B)\big) = \gamma A$, $Q\big((A, B), (A, B + 1)\big) = \gamma B$, and $Q\big((A, B), (A - 1, B - 1)\big) = \delta AB$. Note that the death transition $(A, B) \to (A - 1, B - 1)$

has rate zero if $A = 0$ or $B = 0$. Both axes $\{0\} \times \mathbb{N}$ and $\mathbb{N} \times \{0\}$ are absorbing, and so is the state $(A, B) = (0, 0)$. This chain is regular, i.e., its sequence of transition times is unbounded with probability 1. Indeed, as we will show, the discrete-time chain reaches consensus with probability 1, from which time on the chain is equal to a linear pure-birth process, which is regular.

The corresponding discrete-time jump chain has the same state space $S = \mathbb{N}^2$ and the state-transition probabilities $P\big((A, B), (A+1, B)\big) = \frac{\gamma A}{\gamma(A+B)+\delta AB}$, $P\big((A, B), (A, B+1)\big) = \frac{\gamma B}{\gamma(A+B)+\delta AB}$, and $P\big((A, B), (A-1, B-1)\big) = \frac{\delta AB}{\gamma(A+B)+\delta AB}$ if $A > 0$ or $B > 0$. The axes as well as state $(A, B) = (0, 0)$ is absorbing, as in the continuous-time chain.

As a convention, we will write $X(t)$ for the state of the continuous-time process X at time t, and X_k for the state of the discrete-time jump process after k state transitions. The time to reach consensus is the earliest time T such that $A(T) = 0$ or $B(T) = 0$.

3.2 Time to Reach Consensus

In this section we prove the first part of Theorem 1, i.e., the bounds on the time to reach consensus, both in expected time and with high probability. For that, we will employ a coupling of the A-B protocol Markov chain with a single-species birth-death process. We show that the A-B protocol reaches consensus when the single-species process reaches its extinction state and then bound this time in the single-species process. Figure 1 visualizes the idea of the proof.

We denote the single-species process by $M(t)$. It is a birth-death chain with state space $S = \mathbb{N}$ and transition rates $Q(M, M+1) = \gamma M$ and $Q(M, M-1) = \delta M^2$. State 0 is absorbing. Note that the death rate δM^2 depends quadratically on the current population M, and not linearly like the birth rate γM. The reason is that we want $M(t)$ to bound the minimum of the populations $A(t)$ and $B(t)$ and that the death transition in the A-B protocol is quadratic in this minimum.

We will crucially use the fact that $\mathbb{P}\big(M(t) = 0\big) \leq \mathbb{P}\big(A(t) = 0 \vee B(t) = 0\big)$ for all times t. This, together with a bound on the time until $M(t) = 0$, then gives a bound on the time until consensus in the A-B protocol chain.

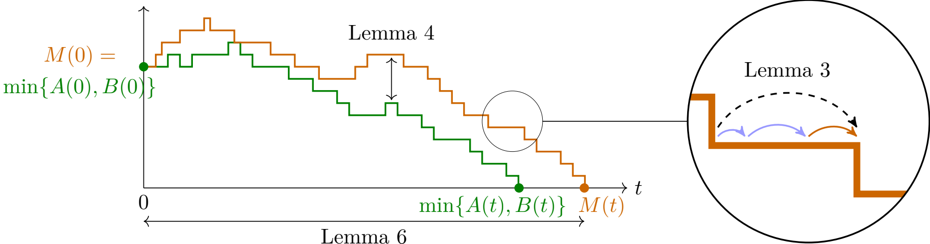


Figure 1 Idea of the proof: Construction of a continuous-time coupling of the A-B protocol and the single species birth-death M chain. Stuttering steps are mapped to effective steps (Lemma 3). An execution of the coupling process fulfills the deterministic guarantee $\min\{A(t), B(t)\} \leq M(t)$ for all times $t \geq 0$ (Lemma 4). From the coupling it follows that $\mathbb{P}\big(M(t) = 0\big) \leq \mathbb{P}\big(A(t) = 0 \vee B(t) = 0\big)$ for the uncoupled processes (Lemma 5). The time until consensus then follows from the time until extinction in the M chain (Lemma 6).

Continuous-time coupling. The coupling is defined as follows. For sequences $(\xi_k)_{k\geq 1}$ of i.i.d. (independent and identically distributed) uniform random variables in the unit interval $[0, 1)$ and $(\eta_k)_{k\geq 1}$ of i.i.d. exponential random variables with normalized rate 1, we define

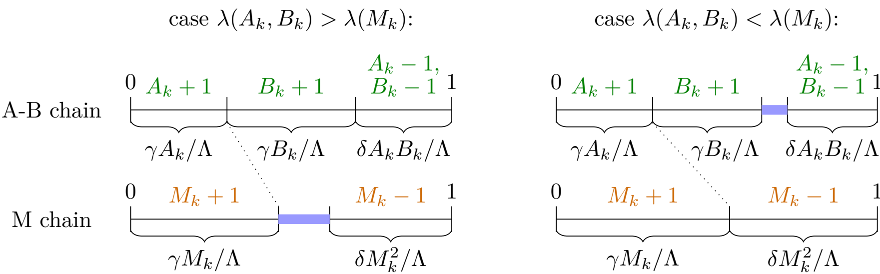


Figure 2 Continuous-time coupling of the A-B chain and the single-species birth-death M-chain, given that $A_k \le B_k$, with $\Lambda = \Lambda(A_k, B_k, M_k)$. The intervals for the cases of ξ_{k+1} and their effect on the A-B chain and the M-chain are shown in green and orange, respectively. Cases that lead to stuttering steps are shown in blue. The dotted relation between intervals is proven in Lemma 4.

the coupled process $(A(t), B(t), M(t))$ as follows. Initially, $M(0) = \min\{A(0), B(0)\}$. For $k \ge 0$, the $(k+1)^{\text{th}}$ transition happens after time $\eta_k/\Lambda(A_k, B_k, M_k)$ where $\Lambda(A, B, M) = \max\{\lambda(A, B), \lambda(M)\}$ is the maximum of the sums of transition rates of the individual chains in states (A, B) and M, respectively, i.e., $\lambda(A, B) = \gamma(A+B) + \delta AB$ and $\lambda(M) = \gamma M + \delta M^2$. The new state $(A_{k+1}, B_{k+1}, M_{k+1})$ of the coupled chain is then determined by the following update rules. The state $(0, 0, 0)$ is absorbing. Otherwise, if $A_k \le B_k$, then:

$$(A_{k+1}, B_{k+1}) = \begin{cases} (A_k + 1, B_k) & \text{if } \xi_{k+1} \in \left[0\,,\ \frac{\gamma A_k}{\Lambda(A_k,B_k,M_k)}\right) \\ (A_k, B_k + 1) & \text{if } \xi_{k+1} \in \left[\frac{\gamma A_k}{\Lambda(A_k,B_k,M_k)}\,,\ \frac{\gamma A_k + \gamma B_k}{\Lambda(A_k,B_k,M_k)}\right) \\ (A_k - 1, B_k - 1) & \text{if } \xi_{k+1} \in \left[1 - \frac{\delta A_k B_k}{\Lambda(A_k,B_k,M_k)}\,,\ 1\right) \\ (A_k, B_k) & \text{otherwise} \end{cases} \tag{1}$$

If $A_k > B_k$ then the roles of A_k and B_k in (1) are exchanged. The update rule for M_{k+1} is:

$$M_{k+1} = \begin{cases} M_k + 1 & \text{if } \xi_{k+1} \in \left[0\,,\ \frac{\gamma M_k}{\Lambda(A_k,B_k,M_k)}\right) \\ M_k - 1 & \text{if } \xi_{k+1} \in \left[1 - \frac{\delta M_k^2}{\Lambda(A_k,B_k,M_k)}\,,\ 1\right) \\ M_k & \text{otherwise} \end{cases}$$

Analysis for time until consensus. Note, that in the coupling "stuttering steps" for (A_k, B_k) or M_k are possible in the definition of the coupled process, making the underlying discrete-time jump chains of, e.g., chain $(A(t), B(t))$ and the A-B protocol, potentially differ. Indeed, the event $(A_{k+1}, B_{k+1}) = (A_k, B_k)$ is possible with positive probability if $\lambda(A_k, B_k) < \lambda(M_k)$, and $M_{k+1} = M_k$ has positive probability if $\lambda(M_k) < \lambda(A_k, B_k)$; see Figure 2. The following elementary Lemma 3, however, shows that the continuous-time chain $(A(t), B(t))$ and the A-B protocol chain have identical transition rates, and are thus identically distributed. The same holds true for the continuous-time chain $M(t)$ and the birth-death M chain.

▶ **Lemma 3.** *Let $T_1, T_2, \ldots$ be a sequence of i.i.d. exponential random variables with rate parameter λ and let k be an independent geometric random variable with success probability p. Then $T = T_1 + \cdots + T_k$ is exponentially distributed with rate parameter $p\lambda$.*

By construction of the coupled process, the single-species birth-death process $M(t)$ indeed dominates the minimum of the species population counts $A(t)$ and $B(t)$ in the following way:

▶ **Lemma 4.** *In the coupled process,* $\min\{A(t), B(t)\} \leq M(t)$ *for all times* $t \geq 0$.

Lemma 4 allows to compare the probabilities of extinction in the single-species chain and of consensus in the A-B protocol chain:

▶ **Lemma 5.** $\mathbb{P}(M(t) = 0) \leq \mathbb{P}(A(t) = 0 \vee B(t) = 0)$ *for all times* $t \geq 0$.

It thus suffices to prove bounds on the time until the population goes extinct in the single-species M chain. For that, we leverage known results on birth-death processes, which are not applicable to the two-species A-B protocol chain.

▶ **Lemma 6.** *If* T *denotes the time until extinction in the single-species process* $M(t)$*, then* $\mathbb{E}\,T = O(1)$.

Proof. The birth rate in state $M(t) = i$ is equal to $\alpha(i) = i\gamma$ and the death rate is equal to $\beta(i) = i^2\delta$. From known general results on birth-death processes [15, p. 149] we obtain, when starting from initial population $M(0) = M$, that

$$\begin{aligned}\mathbb{E}\,T &= \sum_{k=1}^{\infty} \frac{\alpha(1)\cdots\alpha(k-1)}{\beta(1)\cdots\beta(k-1)} \cdot \frac{1}{\beta(k)} + \sum_{j=1}^{M-1}\sum_{k=j+1}^{\infty} \frac{\alpha(j+1)\cdots\alpha(k-1)}{\beta(j+1)\cdots\beta(k-1)} \cdot \frac{1}{\beta(k)} \\ &= \sum_{j=1}^{M}\sum_{k=j-1}^{\infty} \frac{\alpha(j)\cdots\alpha(k)}{\beta(j)\cdots\beta(k)} \cdot \frac{1}{\beta(k+1)} = \sum_{j=1}^{M}\sum_{k=j-1}^{\infty} \frac{\gamma^{k-j+1}}{\delta^{k-j+1}k!/(j-1)!} \cdot \frac{1}{(k+1)^2\delta}\end{aligned}$$

Setting $\alpha = \gamma/\delta$, we have

$$\begin{aligned}\mathbb{E}\,T &= \frac{1}{\delta}\sum_{j=1}^{M}\sum_{k=j-1}^{\infty} \alpha^{k-j+1}\frac{(j-1)!}{(k+1)!(k+1)} = \frac{1}{\delta}\sum_{j=1}^{M}\frac{(j-1)!}{\alpha^j}\sum_{k=j}^{\infty}\frac{\alpha^k}{k!k} \\ &= \frac{1}{\delta}\sum_{j=1}^{M}\frac{(j-1)!}{\alpha^j}\cdot\frac{\alpha^j}{j!j}\sum_{k=j}^{\infty}\frac{\alpha^{k-j}}{k!/j!\cdot k/j} \leq \frac{1}{\delta}\sum_{j=1}^{M}\frac{(j-1)!}{\alpha^j}\cdot\frac{\alpha^j}{j!j}\sum_{k=j}^{\infty}\frac{\alpha^{k-j}}{(k-j)!}\end{aligned}$$

since for $k \geq j \geq 1$, it is $k!/j! \geq (k-j)!$ and $k/j \geq 1$. Thus,

$$\mathbb{E}\,T \leq \frac{1}{\delta}\sum_{j=1}^{M}\frac{(j-1)!}{\alpha^j}\cdot\frac{\alpha^j}{j!j}\cdot e^\alpha = \frac{e^\alpha}{\delta}\sum_{j=1}^{M}\frac{1}{j^2} \leq \frac{e^\alpha\pi^2}{6\delta} = O(1)\ .$$

This concludes the proof. ◀

Denoting with T_{AB} the earliest time t such that $A(t) = 0$ or $B(t) = 0$, and with T_M the earliest time t such that $M(t) = 0$, Lemma 5 is equivalent to $\mathbb{P}(T_M \leq t) \leq \mathbb{P}(T_{AB} \leq t)$, which, in turn, is equivalent to $\mathbb{P}(T_M > t) \geq \mathbb{P}(T_{AB} > t)$. Using the formula $\mathbb{E}\,T = \int_0^\infty \mathbb{P}(T > t)\,dt$, we further have

$$\mathbb{E}(T_M) = \int_0^\infty \mathbb{P}(T_M > t)\,dt \geq \int_0^\infty \mathbb{P}(T_{AB} > t)\,dt = \mathbb{E}(T_{AB})\ .$$

Combining this with Lemma 6, shows that the expected time until consensus in the A-B protocol is also $O(1)$. For the high-probability result in the first part of Theorem 1, we simply make $\Theta(\log n)$ consecutive tries to achieve extinction in an interval of constant time:

▶ **Lemma 7.** *If* T *denotes the time until extinction in the singles-species process* $M(t)$*, then there exists a constant* C *such that* $\mathbb{P}(T \leq C\log_2 n) = 1 - O(1/n)$.

A simple combination of Lemmas 5 and 7 completes the proof of the first part of Theorem 1.

Figure 3 Discrete-time coupling of A-B chain and two Yule processes X and Y with $\lambda(A_k, B_k) = \gamma(A_k + B_k) + \delta A_k B_k$. Cases for ξ_{k+1} that lead to stuttering steps are shown in blue. The interval relations indicated by the dotted lines are proven by induction in Lemma 8.

3.3 Probability of Reaching Majority Consensus

We now turn to the proof of the second part of Theorem 1, i.e., the bound on the probability to achieve majority consensus. We use a coupling of the A-B protocol chain with a different process than for the time bound. Namely we couple it with two parallel independent Yule processes. A Yule process, also known as a pure birth process, has this single state-transition rule $X \to X + 1$ with linear transition rate γX. Since we already showed the upper bound on the time until consensus, it suffices to look at the discrete-time jump process. In particular, the coupling we define is discrete-time.

Discrete-time coupling. For an i.i.d. sequence $(\xi_k)_{k\geq 1}$ of uniformly distributed random variables in the unit interval $[0, 1)$, we define the coupled process (A_k, B_k, X_k, Y_k) by $A_0 = X_0$, $B_0 = Y_0$, and

$$(A_{k+1}, B_{k+1}) = \begin{cases} (A_k - 1, B_k - 1) & \text{if } \xi_{k+1} \in \left[0\,,\ \frac{\delta A_k B_k}{\gamma(A_k+B_k)+\delta A_k B_k}\right) \\ (A_k + 1, B_k) & \text{if } \xi_{k+1} \in \left[\frac{\delta A_k B_k}{\gamma(A_k+B_k)+\delta A_k B_k}\,,\ 1 - \frac{\gamma B_k}{\gamma(A_k+B_k)+\delta A_k B_k}\right) \\ (A_k, B_k + 1) & \text{if } \xi_{k+1} \in \left[1 - \frac{\gamma B_k}{\gamma(A_k+B_k)+\delta A_k B_k}\,,\ 1\right) \end{cases}$$

$$(X_{k+1}, Y_{k+1}) = \begin{cases} (X_k, Y_k) & \text{if } \xi_{k+1} \in \left[0\,,\ \frac{\delta A_k B_k}{\gamma(A_k+B_k)+\delta A_k B_k}\right) \\ (X_k + 1, Y_k) & \text{if } \xi_{k+1} \in \left[\frac{\delta A_k B_k}{\gamma(A_k+B_k)+\delta A_k B_k}\,,\ 1 - \frac{\gamma(A_k+B_k)}{\gamma(A_k+B_k)+\delta A_k B_k} \cdot \frac{Y_k}{X_k+Y_k}\right) \\ (X_k, Y_k + 1) & \text{if } \xi_{k+1} \in \left[1 - \frac{\gamma(A_k+B_k)}{\gamma(A_k+B_k)+\delta A_k B_k} \cdot \frac{Y_k}{X_k+Y_k}\,,\ 1\right) \end{cases}$$

if $\max\{A_k, B_k\} > 0$ and $\max\{X_k, Y_k\} > 0$. Otherwise the process remains constant. Figure 3 visualizes the construction.

Analysis for probability of reaching majority consensus. The crucial property of this coupling is that the initial minority in the A-B process cannot overtake the initial majority before the initial minority overtakes the initial majority in the parallel Yule processes.

▶ **Lemma 8.** *If $X_0 = A_0 \geq B_0 = Y_0$ and $X_k \geq Y_k$ for all $0 \leq k \leq K$, then $X_k - Y_k \leq A_k - B_k$ for all $0 \leq k \leq K$.*

▶ **Lemma 9.** *If $A_0 = X_0$ and $B_0 = Y_0$, then $\mathbb{P}(\exists k\colon A_k = B_k) \leq \mathbb{P}(\exists k\colon X_k = Y_k)$.*

Proof. By Lemma 8, if k is minimal such that $A_k = B_k$, then $X_k = Y_k$. ◀

As defined in the coupling the parallel Yule processes (X_k, Y_k) can have stuttering steps where $(X_{k+1}, Y_{k+1}) = (X_k, Y_k)$. However, this happens only finitely often almost surely. This allows us to analyze a version of the process (X_k, Y_k) without stuttering steps in the sequel.

▶ **Lemma 10.** *If $(\tilde{X}_k, \tilde{Y}_k)$ is the product of two independent pure-birth processes with $\tilde{X}_0 = X_0$ and $\tilde{Y}_0 = Y_0$, then $\mathbb{P}(\exists k\colon \tilde{X}_k = \tilde{Y}_k) = \mathbb{P}(\exists k\colon X_k = Y_k)$.*

By slight abuse of notation, we will use (X_k, Y_k) to refer to the parallel Yule processes without any stuttering steps.

Two parallel independent Yule processes are known to be related to a beta distribution, which we will use below. The regularized incomplete beta function $I_z(\alpha, \beta)$ is defined as $I_z(\alpha, \beta) = \int_0^z t^{\alpha-1}(1-t)^{\beta-1}\,dt \;/\; \int_0^1 t^{\alpha-1}(1-t)^{\beta-1}\,dt$.

▶ **Lemma 11.** *If $X_0 > Y_0$, then $\mathbb{P}\left(\exists k\colon X_k = Y_k\right) = 2 \cdot I_{1/2}(X_0, Y_0)$.*

We define the event "B wins" as A eventually becoming extinct. Then, we have:

▶ **Lemma 12.** *If $A_0 > B_0$, then $\mathbb{P}\left(\exists k\colon A_k = B_k\right) = 2 \cdot \mathbb{P}(B \text{ wins})$.*

Combining the previous two lemmas with the coupling, we get an upper bound on the probability that the A-B protocol fails to reach majority consensus. This upper bound is in terms of the regularized incomplete beta function.

▶ **Lemma 13.** *If $A_0 \geq B_0$, then the A-B protocol fails to reach majority consensus with probability at most $I_{1/2}(A_0, B_0)$.*

Proof. Setting $X_0 = A_0$ and $Y_0 = B_0$, and combining Lemmas 9, 11, and 12, we get $\mathbb{P}(B \text{ wins}) = \frac{1}{2} \cdot \mathbb{P}(\exists k\colon A_k = B_k) \leq \frac{1}{2} \cdot \mathbb{P}(\exists k\colon X_k = Y_k) = I_{1/2}(A_0, B_0)$. ◀

Due to Lemma 13, it only remains to upper-bound the term $I_{1/2}(\alpha, \beta)$. Lemma 14 provides such a bound.

▶ **Lemma 14.** *For $m, \Delta \in \mathbb{N}$, it holds that $I_{1/2}(m + \Delta, m) = \exp\left(-\Omega\left(\frac{\Delta^2}{m}\right)\right)$.*

Combining the above lemmas proves the second part of Theorem 1.

4 Boolean Gates

In terms of circuit design, the A-B protocol can be viewed as a differential signal amplifier. Differential signaling has applications in systems that require high resilience to noise, and thus an application for our inherently growing systems is natural.

In this section we study a protocol that allows to compute the logical Nand of two signals, however with a loss of signal quality at the output. The A-B protocol is then applied to regenerate the signal, obtaining a clear 0 or 1 with high probability. Note that the Nand gate protocol is easily generalized to arbitrary two-input Boolean functions, and so is its analysis.

We start with some notation. A *signal* is from a finite alphabet $\Sigma = \{X, Y, \dots\}$. At each time $t \geq 0$, a signal $X \in \Sigma$ has a value $x(t) \in \{0, 1, \perp\}$. Following a technique from clockless circuit design [34, 21] we encode the value of a signal as a dual-rail signal in the following way. For each signal X, there are two species X^0 and X^1. Intuitively, for $v \in \{0, 1\}$, a large count of $X^v(t)$ and a low count of $X^{\neg v}(t)$ encodes for $x(t) = v$. In fact, we will ask for a minimum gap in species counts between $X^v(t)$ and $X^{\neg v}(t)$. If the signal is neither 0 nor 1, we will say that it has value $\perp$. We will make the assumptions on the input signals precise in the sequel, and discuss guarantees on output signals when specifying the gate input/output behavior.

Let X^0, X^1 be species of a dual-rail encoding of signal X. For convenience we write $X(t)$ for $X^0(t) + X^1(t)$. For $n, \Delta \in \mathbb{N}$, we say signal X is *initially* (n, Δ)*-correct with value* $x \in \{0, 1\}$ if

$$X(0) \geq n \qquad \text{and} \qquad X^{\neg x}(0) \leq \frac{n - \Delta}{2} \quad .$$

The *initial gap* $X^x(0) - X^{\neg x}(0)$ of signal X is thus bounded by $X^x(0) - X^{\neg x}(0) = X^x(0) + X^{\neg x}(0) - 2X^{\neg x}(0) \geq \Delta$.

4.1 Dual-Rail NAND Gate

A dual-rail implementation of a NAND gate with input signals A, B and output signal Y is as a protocol with input species $\mathcal{I} = \{A^0, A^1, B^0, B^1\}$, output species $\mathcal{O} = \{Y^0, Y^1\}$, and no internal species. Initial counts for outputs that are not inputs are $Y^0(0) = Y^1(0) = 0$. Further, for all $a, b \in \{0, 1\}$ and $y = \neg(a \wedge b)$, the protocol contains a reaction

$$A^a + B^b \xrightarrow{\alpha} A^a + B^b + Y^y \quad ,$$

where $\alpha > 0$ is the gate's rate constant. Since all species are permanently replicating, we further have the obligatory duplication reactions $A^i \xrightarrow{\gamma} 2A^i$, $B^i \xrightarrow{\gamma} 2B^i$, and $Y^i \xrightarrow{\gamma} 2Y^i$ for $i \in \{0, 1\}$. Figure 4 depicts the NAND gate with the subsequent amplification protocol.

Figure 4 Dual-rail NAND gate with input signals A and B an output signal Y. Successive amplification of Y to signal Z shown in gray.

In Section 4.2 we will show that the NAND gate ensures the following input-output specification:

▶ **Theorem 15.** *Assume that the NAND gate's input signals A, B are dual-rail encoded signals, and that they are initially (n, Δ)-correct with values $a, b \in \{0, 1\}$, respectively, where $n \in \mathbb{N}^+$ and $\Delta \geq 0.62 \cdot \max\{A(0), B(0)\}$. Then with high probability, there exists some time $t = O(1)$ such that $Y(t) = n$ and $Y^y(t) - Y^{\neg y}(t) = \Omega(n)$ for the output signal Y where $y = \neg(a \wedge b)$ is the correct NAND output based on the initial values a, b of signals A and B, respectively.*

4.2 Gate Correctness and Performance

We now turn to the proof of Theorem 15. For our analysis we need a bound on the regularized incomplete beta function $I_{3/4}$.

▶ **Lemma 16.** *For $X \geq Y$, it is $I_{3/4}(X, Y) \leq \frac{1}{2} \exp\left(-\frac{(X-Y+1)^2}{4(Y-1)} + (X + Y - 1) \log \frac{3}{2} \right)$. In particular, for $m, \Delta \geq 0$, $I_{3/4}(m + \Delta, m) \leq \frac{1}{2} \exp\left(-\frac{(\Delta+1)^2}{4(m-1)} + \frac{2m+\Delta}{2}\right)$.*

The following lemma shows that for $z = 3/4$, the function $(x, y) \mapsto I_z(x, y)$ is non-decreasing in (x, y) along the discretized line with slope $1/3$.

▶ **Lemma 17.** *If $X \geq 3Y \geq 0$, then $I_{3/4}(X, Y) \leq I_{3/4}(X + 3, Y + 1)$.*

We are now in the position to show a lower bound on the probability for a discrete time Yule process with two species X and Y, that $\lim_{k\to\infty} X_k/(X_k+Y_k) < 3/4$, given that the initial values fulfill $X_0/(X_0+Y_0) > 3/4$ and that there is a step ℓ with $X_\ell/(X_\ell+Y_\ell) \leq 3/4$.

▶ **Lemma 18.** *Let X and Y be species from a Yule process. Assume that $X_0/(X_0+Y_0) > 3/4$ for the initial values. Then $\mathbb{P}\left(\lim_{k\to\infty} \frac{X_k}{X_k+Y_k} < \frac{3}{4} \;\middle|\; \exists \ell : \frac{X_\ell}{X_\ell+Y_\ell} \leq \frac{3}{4}\right) \geq \omega(X_0, Y_0)$ where $\omega(X_0,Y_0) = \inf\left\{I_{3/4}(x,y) \,\middle|\, x \geq X_0 \wedge y \geq Y_0 + 1 \wedge x \in 3y - \{0,1,2\}\right\}$. Moreover, $\omega(X_0,Y_0) > 0.444$*

Making use of Lemma 18, we next prove an upper bound on the probability that the two-species discrete-time Yule process X, Y, with an initial large majority of X, eventually hits a step where its relative population size drops to $\frac{3}{4}$ or below.

▶ **Lemma 19.** *Let X and Y be species from a Yule process. Assume that $\frac{X_0}{X_0+Y_0} > \frac{3}{4}$. Then $\mathbb{P}\left(\exists k : \frac{X_k}{X_k+Y_k} \leq \frac{3}{4}\right) < \frac{I_{3/4}(X_0,Y_0)}{0.444}$.*

The following lemma provides a lower bound on the probability that the dual-rail encoding of signals A and B, that are both initially (n,Δ)-correct, for $\Delta > n/2$, remains separated as their species grow.

▶ **Lemma 20.** *Let A^0, A^1 as well as B^0, B^1 be species of a dual-rail encoding of signals A and B. Assume that each species follows a Yule processes. If signals A and B are initially (n,Δ)-correct with $n, \Delta \in \mathbb{N}$ with $\Delta > \frac{n}{2}$ for some $a, b \in \{0,1\}$, then $\mathbb{P}\left(\forall t \geq 0 : \frac{A^a(t)}{A(t)} > \frac{3}{4} \wedge \frac{B^b(t)}{B(t)} > \frac{3}{4}\right)$ is lower bounded by*

$$\left(1 - \frac{1}{2 \cdot 0.444} \exp\left(\frac{1}{2}\left(-\frac{\Delta^2}{(n-\Delta)} + \max\{A(0), B(0)\}\right)\right)\right)^2 .$$

We next show in Lemma 21 that when the NAND gates has produced n output species Y^0 and Y^1, a certain gap $\Delta > 0$ is guaranteed with a probability that depends on n and Δ. However, instead of showing this for the original NAND gate, we first prove that the bound holds for an adapted version where Y^0 and Y^1 do not duplicate. We later extend the result to the original NAND gate in Lemma 22.

▶ **Lemma 21.** *Consider an adapted version of the NAND gate with dual-rail encoded input signals A, B and output signal Y. In the adapted version, species Y^0 and Y^1 do not duplicate. Further, assume that for some $a, b \in \{0,1\}$, $\forall t \geq 0 : \frac{A^a(t)}{A(t)} > \frac{3}{4} \wedge \frac{B^b(t)}{B(t)} > \frac{3}{4}$. Then, with $y = \neg(a \wedge b)$ being the correct Boolean output of the gate, for any $t \geq 0$ and $\Delta, n \in \mathbb{N}$ with $\Delta \leq n/8$, $\mathbb{P}\left(Y^y(t) - Y^{\neg y}(t) > \Delta \;\middle|\; Y(t) = n\right) \geq 1 - \exp\left(-\frac{\left(\frac{n}{8}-\Delta\right)^2}{2n}\right)$.*

▶ **Lemma 22.** *Consider the NAND gate with dual-rail encoded input signals A, B and output signal Y. If for some $a, b \in \{0,1\}$, $\forall t \geq 0 : \frac{A^a(t)}{A(t)} > \frac{3}{4} \wedge \frac{B^b(t)}{B(t)} > \frac{3}{4}$, $A(0) \geq n$, and $B(0) \geq n$ then, letting $y = \neg(a \wedge b)$ be the correct Boolean output of the gate, with high probability there exists a $t = O(1)$ such that $Y^y(t) - Y^{\neg y}(t) = \Omega(n)$ and $Y(t) = n$.*

We are now in the position to prove Theorem 15, showing the correctness of the NAND gate if each of the two dual-rail input signals has a sufficiently large gap between its rails.

Proof of Theorem 15. The theorem follows from Lemma 22 if its assumption holds with high probability. The latter follows from Lemma 20 if the exponent $\frac{1}{2}\left(-\frac{\Delta^2}{(n-\Delta)} + \max\{A(0), B(0)\}\right)$ is in $\Omega\left(-\max\{A(0), B(0)\}\right)$. We next show that this is the case.

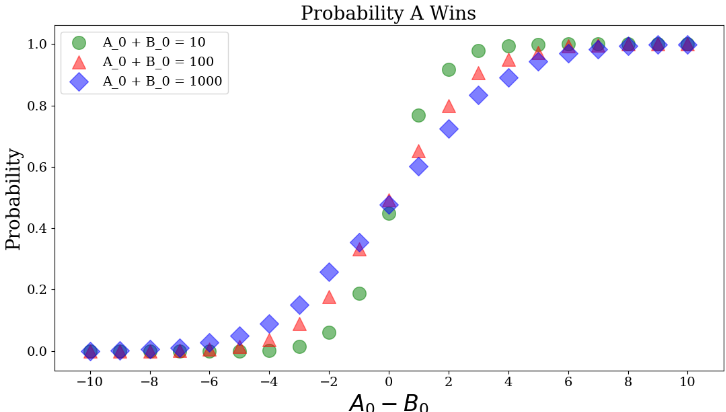


Figure 5 The probability that species A survives while species B goes extinct is sharply dependent on their initial difference in population count $A_0 - B_0$. The sharpness of the transition is inversely proportional to initial population size $A_0 + B_0$.

Let $M = \max\{A(0), B(0)\}$. From $\Delta \geq \mu M$ with $\mu = 0.62$ we have, $-\frac{\Delta^2}{n-\Delta} + M \leq -\frac{\mu^2 M^2}{M - \mu M} + M \leq M\left(1 - \frac{\mu^2}{(1-\mu)}\right)$. It thus remains to show that $\left(1 - \frac{\mu^2}{(1-\mu)}\right) < 0$. By algebraic manipulation, this is the case if $\mu \in \left(\frac{1}{2}(\sqrt{5} - 1), 1\right)$, which is true by assumption. The theorem follows. ◀

5 Simulations

5.1 A-B Protocol Simulations

Simulations corresponding to the A-B protocol complement the theoretical results above. The A-B protocol is simulated in Figure 5 for the probability that species A survives, while species B goes extinct. The birth and death rates, γ and δ, are both set to 1. The probability that the protocol converges on A is primarily dependent on the difference in initial population size $A_0 - B_0$. Larger populations are only slightly less sensitive to the difference: Figure 5 demonstrates that the total population size across two orders of magnitude has a small effect compared to the difference between species. Indeed, this behavior qualitatively matches the bound in Theorem 1 with $-\Omega(\Delta^2/n)$ in the exponent.

The dependence of expected convergence time for the A-B protocol is explored over its reaction rate constants and initial conditions in Figure 6. Exponential changes in rate constants yield exponential changes in convergence time. As expected, the convergence time is more strongly dependent on the death rate constant δ, than the birth rate constant γ. Convergence time sharply increases if the initial concentrations of the two species A and B are proportional. The off-diagonal initial concentrations converge faster for larger population sizes since the absolute difference in concentrations is larger.

5.2 *In silico* Biological Implementation

While the studied model is a simplification, it represents core functions that constitute collective decision-making among biological species, and is readily adaptable for specific biological applications. If reactions are modified such that one of the two reactants does not change, the model could represent one-way messaging equivalent to a conjugation event

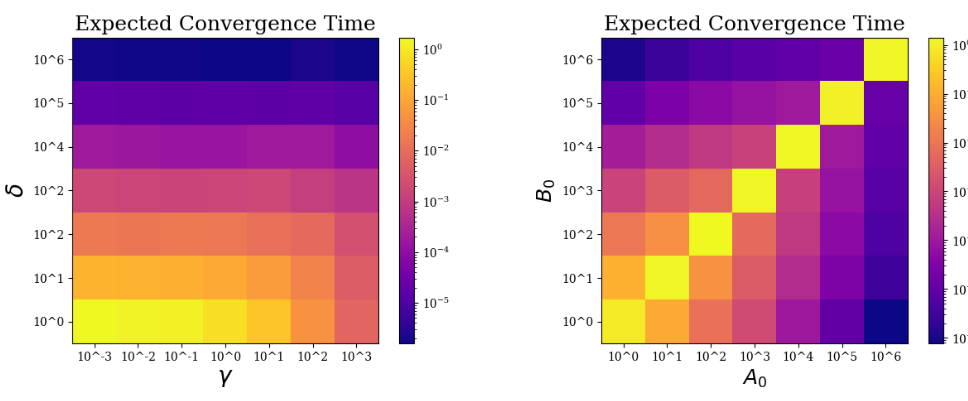


Figure 6 Log-scaled expected convergence time of the A-B protocol is represented by color. Corresponding values are shown on the adjacent vertical bar. **Left:** rate constants γ and δ with $A_0 = B_0 = 100$. **Right:** initial populations sizes with $\gamma = 0.01$ and $\delta = 1$.

between a sender and receiver bacterial cell [19]. Similarly, if the messages A and B are coded as free species diffusible between senders and receivers, it could represent communication between bacterial cells using bacteriophage particles as messages [23].

In this section, we discuss a plausible biological implementation with *E. coli* bacteria that use conjugation to communicate. Conjugation is a method of genetic communication in which circular DNA plasmids are transferred from a sender cell to a receiver cell. An F plasmid allows a cell to be a sender during conjugation. The receiver can be engineered to express a logical function using the received plasmid and its existing DNA, although the internal implementation is not detailed for this simulation. A conjugation reaction with a sender S and a receiver R is described by $R + S \xrightarrow{\delta} f(R, S) + S$, where δ is the conjugation rate constant. Both, the amplifier and the NAND gate follow this scheme. For the amplifier, $f(R, S) = \emptyset$ and for the NAND gate $f(R, S) = Y$, where Y is the gate's corresponding output species. While with wild-type F plasmids, *E. coli* are either senders (with F plasmid) or receivers (without F plasmid), there exist engineered systems that allow the same cell (with F plasmid) to be both a sender and a receiver [9, 19]. Note that a single cell still cannot act as both the sender and the receiver during a single reaction.

The growth of the *E. coli* is modeled by a logistic model with a carrying capacity of 10^9 cells. Reaction rate constants for duplication $\gamma = 0.016$ and for conjugation $\delta = 10^{-11}$ have been taken from Dimitriu *et al.* [9]. For our implementation, amplification of the gate's inputs and outputs was executed in parallel to the gate's protocol. The simulations discussed in the following suggest that sequential execution is not required for correctness and performance, greatly simplifying the biological design. If all possible gate reactions were used, inputs that lead to Y^1 would be more susceptible to noise since there are more possible input pairs leading to Y^1 than Y^0 in a NAND gate. This was alleviated by selecting a subset of all possible gate reactions in which three reactions lead to Y^1 and two reactions lead to Y^0.

For performance with many individuals, simulations are done using the τ-leaping approximation of stochastic simulation, in which multiple reactions occur during a dynamic time interval of τ, before updating reaction rates [12, 14]. The initial population size is set to 5×10^8, the carrying capacity to 1×10^9, and the initial input error to 10% of wrong input species per input. Despite the low rate of communication from conjugation, at least 80% of the output species are correct at all times for all input choices.

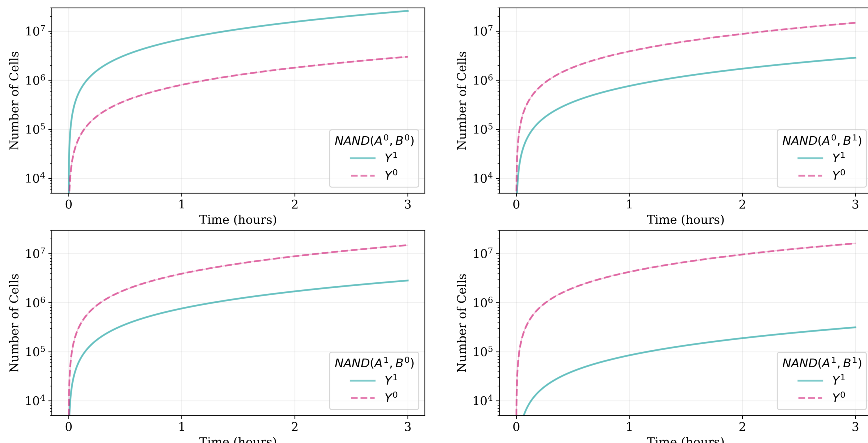


Figure 7 A biologically plausible implementation of a NAND gate with amplifiers on inputs and outputs. Initial population size is 5×10^8, carrying capacity of 10^9 cells. Reaction rate constants were set to $\gamma = 0.016$ and $\delta = 10^{-11}$ [9]. The output species is shown for each choice of inputs. The initial input error is 10%. All choices lead to correct, clearly separable outputs within half an hour. Confidence intervals from 30 sample simulations are smaller than the width of the lines.

6 Conclusions

We considered the majority consensus problem with continuous population growth in a stochastic setting, and established the A-B protocol between two competing species A and B with birth reactions $A \longrightarrow 2A$ and $B \longrightarrow 2B$, and death reaction $A + B \longrightarrow \emptyset$. In particular, the input of the A-B protocol are two species A and B with an initial total population size $n = A(0) + B(0)$ and an initial gap $\Delta = |A(0) - B(0)|$. We showed that the A-B protocol reaches majority consensus with high probability if the gap weakly grows with the population size according to $\Delta = \Omega(\sqrt{n \log n})$. Expected convergence time until consensus is constant and in $O(\log n)$ with high probability.

We further demonstrated how to use dual-rail gates to implement digital circuits computing arbitrary Boolean functions. As opposed to thresholds of a single species, dual-rail encoding is particularly useful in our birth systems as the A-B protocol allows us to amplify and thus regenerate such signals. As a dual-rail gate implementation, we presented the NAND gate protocol that takes two dual-rail encoded input signals and produces a corresponding dual-rail output signal. The protocol is simple, an important criterion for follow up in real-world biological implementations. We proved that, given a sufficiently large initial gap between the rails of the input signals, our gate produces the correct output with high probability in $O(\log n)$ time, where n is a lower bound on the initial input population size. In particular, our gate guarantees an output signal gap of $\Omega(n)$ if both inputs have a gap of at least 0.62 times their initial population size. By alternating execution of the NAND gate protocol and the A-B protocol, layer by layer, we finally arrive at computing the circuit's outputs.

Simulations show that the qualitative behavior of our protocols matches the behavior expected from the asymptotic bounds. While the studied A-B protocol and the NAND gate protocol are simplifications of biological implementations of consensus and gate evaluation protocols, we believe that our results give a signpost for future research on the successful implementation of complex distributed systems such as indirect inter-cellular communication via phages. We discussed a potential biological implementation based on communication by conjugation among engineered *E. coli*.

References

1. J. Christopher Anderson, Elizabeth J. Clarke, Adam P. Arkin, and Christopher A. Voigt. Environmentally controlled invasion of cancer cells by engineered bacteria. *Journal of Molecular Biology*, 355(4):619–627, January 2006. doi:10.1016/j.jmb.2005.10.076.
2. Dana Angluin, James Aspnes, and David Eisenstat. Fast computation by population protocols with a leader. *Distributed Computing*, 21(3):183–199, 2008.
3. Dana Angluin, James Aspnes, and David Eisenstat. A simple population protocol for fast robust approximate majority. *Distributed Computing*, 21(2):87–102, March 2008. doi:10.1007/s00446-008-0059-z.
4. L Billard. Competition between two species. *Stochastic Processes and their Applications*, 2(4):391–398, 1974.
5. Pierre Brémaud. *Markov Chains: Gibbs Fields, Monte Carlo Simulation, and Queues*. Springer, Heidelberg, 1999.
6. Anne Condon, Monir Hajiaghayi, David Kirkpatrick, and Ján Maňuch. Approximate majority analyses using tri-molecular chemical reaction networks. *Natural Computing*, 19:249–270, 2020.
7. Rachel Cummings, David Doty, and David Soloveichik. Probability 1 computation with chemical reaction networks. *Natural Computing*, 15:245–261, 2016.
8. Ramiz Daniel, Jacob R. Rubens, Rahul Sarpeshkar, and Timothy K. Lu. Synthetic analog computation in living cells. *Nature*, 497(7451):619–623, May 2013. doi:10.1038/nature12148.
9. Tatiana Dimitriu, Chantal Lotton, Julien Bénard-Capelle, Dusan Misevic, Sam P Brown, Ariel B Lindner, and François Taddei. Genetic information transfer promotes cooperation in bacteria. *Proceedings of the National Academy of Sciences*, 111(30):11103–11108, 2014.
10. William Feller. Die Grundlagen der volterraschen Theorie des Kampfes ums Dasein in wahrscheinlichkeitstheoretischer Behandlung (1939). In *Selected Papers I*, pages 441–470. Springer, 2015.
11. Matthias Függer, Manish Kushwaha, and Thomas Nowak. Digital circuit design for biological and silicon computers. In *Advances in Synthetic Biology*, pages 153–171. Springer, 2020.
12. Daniel T Gillespie. Approximate accelerated stochastic simulation of chemically reacting systems. *The Journal of Chemical Physics*, 115(4):1716–1733, 2001.
13. Antonio Gómez-Corral and M López García. Extinction times and size of the surviving species in a two-species competition process. *Journal of Mathematical Biology*, 64(1-2):255–289, 2012.
14. S. Hoops, S. Sahle, R. Gauges, C. Lee, J. Pahle, N. Simus, M. Singhal, L. Xu, P. Mendes, and U. Kummer. COPASI–a COmplex PAthway SImulator. *Bioinformatics*, 22(24):3067–3074, October 2006. doi:10.1093/bioinformatics/btl485.
15. Samuel Karlin and Howard M. Taylor. *A First Course in Stochastic Processes*. Academic Press, New York, 2 edition, 1975.
16. David G Kendall. Branching processes since 1873. *Journal of the London Mathematical Society*, 1(1):385–406, 1966.
17. Eric Libby and William C Ratcliff. Ratcheting the evolution of multicellularity. *Science*, 346(6208):426–427, 2014.
18. Nancy A Lynch. *Distributed algorithms*. Morgan Kaufmann, 1996.
19. John P Marken and Richard M Murray. Addressable," packet-based" intercellular communication through plasmid conjugation. *bioRxiv*, page 591552, 2019.
20. Tae Seok Moon, Chunbo Lou, Alvin Tamsir, Brynne C. Stanton, and Christopher A. Voigt. Genetic programs constructed from layered logic gates in single cells. *Nature*, 491(7423):249–253, October 2012. doi:10.1038/nature11516.
21. Chris J Myers. *Asynchronous Circuit Design*. John Wiley & Sons, 2001.
22. Artem S Novozhilov, Georgy P Karev, and Eugene V Koonin. Biological applications of the theory of birth-and-death processes. *Briefings in Bioinformatics*, 7(1):70–85, 2006.
23. Monica E Ortiz and Drew Endy. Engineered cell-cell communication via DNA messaging. *Journal of Biological Engineering*, 6(1):16, December 2012. doi:10.1186/1754-1611-6-16.

24 C. J. Paddon, P. J. Westfall, D. J. Pitera, K. Benjamin, K. Fisher, D. McPhee, M. D. Leavell, A. Tai, A. Main, D. Eng, D. R. Polichuk, K. H. Teoh, D. W. Reed, T. Treynor, J. Lenihan, H. Jiang, M. Fleck, S. Bajad, G. Dang, D. Dengrove, D. Diola, G. Dorin, K. W. Ellens, S. Fickes, J. Galazzo, S. P. Gaucher, T. Geistlinger, R. Henry, M. Hepp, T. Horning, T. Iqbal, L. Kizer, B. Lieu, D. Melis, N. Moss, R. Regentin, S. Secrest, H. Tsuruta, R. Vazquez, L. F. Westblade, L. Xu, M. Yu, Y. Zhang, L. Zhao, J. Lievense, P. S. Covello, J. D. Keasling, K. K. Reiling, N. S. Renninger, and J. D. Newman. High-level semi-synthetic production of the potent antimalarial artemisinin. *Nature*, 496(7446):528–532, April 2013. `doi:10.1038/nature12051`.

25 Amir Pandi, Mathilde Koch, Peter L Voyvodic, Paul Soudier, Jérôme Bonnet, Manish Kushwaha, and Jean-Loup Faulon. Metabolic perceptrons for neural computing in biological systems. *Nature Communications*, 10(1):1–13, 2019.

26 William C Ratcliff, R Ford Denison, Mark Borrello, and Michael Travisano. Experimental evolution of multicellularity. *Proceedings of the National Academy of Sciences*, 109(5):1595–1600, 2012.

27 Sergi Regot, Javier Macia, Núria Conde, Kentaro Furukawa, Jimmy Kjellén, Tom Peeters, Stefan Hohmann, Eulàlia de Nadal, Francesc Posas, and Ricard Solé. Distributed biological computation with multicellular engineered networks. *Nature*, 469(7329):207–211, December 2010. `doi:10.1038/nature09679`.

28 GEH Reuter. Competition processes. In *Proc. 4th Berkeley Symp. Math. Statist. Prob*, volume 2, pages 421–430, 1961.

29 CJ Ridler-Rowe. On competition between two species. *Journal of Applied Probability*, 15(3):457–465, 1978.

30 Thomas L Saaty. *Elements of Queueing Theory*, volume 34203. McGraw-Hill New York, 1961.

31 Sebastian R Schmidl, Felix Ekness, Katri Sofjan, Kristina N-M Daeffler, Kathryn R Brink, Brian P Landry, Karl P Gerhardt, Nikola Dyulgyarov, Ravi U Sheth, and Jeffrey J Tabor. Rewiring bacterial two-component systems by modular dna-binding domain swapping. *Nature Chemical Biology*, 15(7):690–698, 2019.

32 Shimyn Slomovic, Keith Pardee, and James J Collins. Synthetic biology devices for in vitro and in vivo diagnostics. *Proceedings of the National Academy of Sciences*, 112(47):14429–14435, 2015.

33 David Soloveichik, Matthew Cook, Erik Winfree, and Jehoshua Bruck. Computation with finite stochastic chemical reaction networks. *Natural Computing*, 7(4):615–633, February 2008. `doi:10.1007/s11047-008-9067-y`.

34 Jens Sparsø and Steve Furber. *Principles of asynchronous circuit design*. Springer, 2002.

35 Jeffrey J Tabor, Howard M Salis, Zachary Booth Simpson, Aaron A Chevalier, Anselm Levskaya, Edward M Marcotte, Christopher A Voigt, and Andrew D Ellington. A synthetic genetic edge detection program. *Cell*, 137(7):1272–1281, 2009.

36 Alvin Tamsir, Jeffrey J. Tabor, and Christopher A. Voigt. Robust multicellular computing using genetically encoded NOR gates and chemical 'wires'. *Nature*, 469(7329):212–215, December 2010. `doi:10.1038/nature09565`.

37 Pei Kun R Tay, Peter Q Nguyen, and Neel S Joshi. A synthetic circuit for mercury bioremediation using self-assembling functional amyloids. *ACS Synthetic Biology*, 6(10):1841–1850, 2017.

38 Vito Volterra. *Leçons sur la Theorie Mathematique de la Lutte pour la Vie*. Gauthier-Villars, Paris, 1931.

Who Started This Rumor? Quantifying the Natural Differential Privacy of Gossip Protocols

Aurélien Bellet
INRIA, Villeneuve d'Ascq, France
aurelien.bellet@inria.fr

Rachid Guerraoui
EPFL, Lausanne, Switzerland
rachid.guerraoui@epfl.ch

Hadrien Hendrikx
PSL, DIENS, INRIA, Paris, France
hadrien.hendrikx@inria.fr

Abstract

Gossip protocols (also called rumor spreading or epidemic protocols) are widely used to disseminate information in massive peer-to-peer networks. These protocols are often claimed to guarantee privacy because of the uncertainty they introduce on the node that started the dissemination. But is that claim really true? Can the source of a gossip safely hide in the crowd? This paper examines, for the first time, gossip protocols through a rigorous mathematical framework based on differential privacy to determine the extent to which the source of a gossip can be traceable. Considering the case of a complete graph in which a subset of the nodes are curious, we study a family of gossip protocols parameterized by a "muting" parameter s: nodes stop emitting after each communication with a fixed probability $1 - s$. We first prove that the standard push protocol, corresponding to the case $s = 1$, does not satisfy differential privacy for large graphs. In contrast, the protocol with $s = 0$ (nodes forward only once) achieves optimal privacy guarantees but at the cost of a drastic increase in the spreading time compared to standard push, revealing an interesting tension between privacy and spreading time. Yet, surprisingly, we show that some choices of the muting parameter s lead to protocols that achieve an optimal order of magnitude in both privacy and speed. Privacy guarantees are obtained by showing that only a small fraction of the possible observations by curious nodes have different probabilities when two different nodes start the gossip, since the source node rapidly stops emitting when s is small. The speed is established by analyzing the mean dynamics of the protocol, and leveraging concentration inequalities to bound the deviations from this mean behavior. We also confirm empirically that, with appropriate choices of s, we indeed obtain protocols that are very robust against concrete source location attacks (such as maximum a posteriori estimates) while spreading the information almost as fast as the standard (and non-private) push protocol.

2012 ACM Subject Classification Security and privacy → Privacy-preserving protocols

Keywords and phrases Gossip Protocol, Rumor Spreading, Differential Privacy

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.8

Related Version A full version of the paper is available at `https://arxiv.org/abs/1902.07138`.

Funding *Aurélien Bellet*: This research was supported by grants ANR-16-CE23-0016-01 and ANR-18-CE23-0018-03, by the European Union's Horizon 2020 Research and Innovation Program under Grant Agreement No. 825081 COMPRISE (`https://project.inria.fr/comprise/`), by a grant from CPER Nord-Pas de Calais/FEDER DATA Advanced data science and technologies 2015-2020.
Rachid Guerraoui: This research was supported by European ERC Grant 339539 - AOC.
Hadrien Hendrikx: This research was supported by the MSR-INRIA joint centre.

© Aurélien Bellet, Rachid Guerraoui, and Hadrien Hendrikx;
licensed under Creative Commons License CC-BY

1 Introduction

Gossip protocols (also called *rumor spreading* or *epidemic protocols*), in which participants *randomly* choose a neighbor to communicate with, are both simple and efficient means to exchange information in P2P networks [23, 39, 33, 8]. They are a basic building block to propagate and aggregate information in distributed databases [13, 9] and social networks [14, 27], to model the spread of infectious diseases [29], as well as to train machine learning models on distributed datasets [15, 12, 43, 35].

Some of the information gossiped may be sensitive, and participants sharing it may not want to be identified. This can for instance be the case of whistle-blowers or individuals that would like to exercise their right to freedom of expression in totalitarian regimes. Conversely, it may sometimes be important to locate the source of a (computer or biological) virus, or fake news, in order to prevent it from spreading before too many participants get "infected".

There is a folklore belief that gossip protocols inherently guarantee some form of *source anonymity* because participants cannot know who issued the information in the first place [26]. Similarly, identifying "patient zero" for real-world epidemics is known to be a very hard task. Intuitively indeed, random and local exchanges make identification harder. But to what extent? Although some work has been devoted to the design of source location strategies in specific settings [31, 38, 41], the general anonymity claim has never been studied from a pure *privacy* perspective, that is, independently of the very choice of a source location technique. Depending on the use-case, it may be desirable to have strong privacy guarantees (e.g., in anonymous information dissemination) or, on the contrary, we may hope for weak guarantees, e.g., when trying to identify the source of an epidemic. In both cases, it is crucial to precisely quantify the anonymity level of gossip protocols and study its theoretical limits through a principled approach. This is the challenge we take up in this paper for the classic case of gossip dissemination in a complete network graph.

Our first contribution is an information-theoretic model of anonymity in gossip protocols based on (ϵ, δ)-*differential privacy* (DP) [16, 17]. Originally introduced in the database community, DP is a precise mathematical framework recognized as the gold standard for studying the privacy guarantees of information release protocols. In our proposed model, the information to protect is the source of the gossip, and an adversary tries to locate the source by monitoring the communications (and their relative order) received by a subset of f curious nodes. In a computer network, these curious nodes may have been compromised by a surveillance agency; in our biological example, they could correspond to health professionals who are able to identify whether a given person is infected. Our notion of DP then requires that the probability of any possible observation of the curious nodes is almost the same no matter who is the source, thereby limiting the predictive power of the adversary regardless of its actual source location strategy. A distinctive aspect of our model is that the mechanism that seeks to ensure DP comes only from the *natural* randomness and partial observability of gossip protocols, not from additional perturbation or noise which affects the desired output as generally needed to guarantee DP [18]. We believe our adaptation of DP to the gossip context to be of independent interest. We also complement it with a notion of *prediction uncertainty* which guarantees that even unlikely events do not fully reveal the identity of the source under a uniform prior on the source. This property directly upper bounds the probability of success of any source location attack, including the maximum likelihood estimate.

We use our proposed model to study the privacy guarantees of a generic family of gossip protocols parameterized by a *muting parameter* s: nodes have a fixed probability $1 - s$ to stop emitting after each communication (until they receive the rumor again). In our

Table 1 Summary of results to illustrate the tension between privacy and speed. n is the total number of nodes and f/n is the fraction of curious nodes in the graph. $\delta \in [0,1]$ quantifies differential privacy guarantees (smaller is better). Spreading time is asymptotic in n.

	Muting param.	δ ensuring $(0,\delta)$-DP	Spreading time
Standard push (minimal privacy, maximal speed)	$s = 1$	1	$\mathcal{O}(\log n)$
Muting after infecting (maximal privacy, minimal speed)	$s = 0$	$\frac{f}{n}$	$\mathcal{O}(n \log n)$
Generic parameterized gossip (privacy vs. speed trade-off)	$0 < s < 1$	$s + (1-s)\frac{f}{n}$	$\mathcal{O}(\log(n)/s)$

biological parallel, this corresponds to the fact that a person stops infecting other people after some time. The muting parameter captures the ability of the protocol to forget initial conditions, thereby helping to conceal the identity of the source. In the extreme case where $s = 1$, we recover the standard "push" gossip protocol [39], and show that it is inherently *not* differentially private for large graphs. In contrast, we also show that, at the other end of the spectrum, choosing $s = 0$ leads to *optimal privacy guarantees* among all gossip protocols.

More generally, we determine *matching upper and lower bounds* on the privacy guarantees of gossip protocols. Essentially, our upper bounds on privacy are obtained by tightly lower bounding the probability that the source node contacts a curious node before another node does, and upper bounding the probability that this happens for a random node fixed in advance, in a way that holds for all gossip algorithms. Remarkably, despite the fact that the source node always has a non-negligible probability of telling the rumor to a curious node first, our results highlight the fact that setting $s = 0$ leads to strong privacy guarantees in several regimes, including the pure $(\epsilon, 0)$-DP as well as prediction uncertainty.

It turns out that, although achieving optimal privacy guarantees, choosing $s = 0$ leads to a very slow spreading time (*log-linear* in the number of nodes n). This highlights an interesting tension between *privacy* and *spreading time*: the two extreme values for the muting parameter s recover the two extreme points of this trade-off. We then show that more balanced trade-offs can be achieved: appropriate choices of the muting parameter lead to gossip protocols that are *near-optimally private* with a spreading time that is *logarithmic* in the size of the graph. In particular, the trade-off between privacy and speed shows up in the constants but, surprisingly, some choices of the parameter lead to protocols that achieve an optimal order of magnitude for both aspects. Our results on this trade-off are summarized in Table 1: for a proportion f/n of curious nodes, one can see that setting the muting parameter $s = f/n$ achieves almost optimal privacy (up to a factor 2) while being substantially faster than $s = 0$ (optimal up to a factor f/n). Similarly, if one wants to achieve $(0, \delta_0)$-differential privacy with $\delta_0 > 2f/n$, then it is possible to set $s = \delta_0/2$ and obtain a protocol that respects the privacy constraint with spreading time $\mathcal{O}(\log(n)/\delta_0)$. From a technical perspective, these privacy results are obtained by showing that only a small fraction of the possible observations by curious nodes have different probabilities when two different nodes start with the gossip. This requires to precisely evaluate the probability of well-chosen worst-case sequences, which is generally hard as randomness is involved both when nodes decide to stop spreading the rumor (with probability $1 - s$) and when they choose who to communicate with. Our parameterized gossip protocol can be seen as a population protocol [4], and we prove its speed by analyzing its mean dynamics and leveraging concentration inequalities to bound the deviations from the mean dynamics.

We support our theoretical findings by an empirical study of our parameterized gossip protocols. The results show that appropriate choices of s lead to protocols that are very robust against classical source location attacks (such as maximum a posteriori estimates) while spreading the information almost as fast as the standard (and non-private) push protocol. Crucially, we observe that our differential privacy guarantees are very well aligned with the ability to withstand attacks that leverage background information, e.g., targeting known activists or people who have been to certain places.

The rest of the paper is organized as follows. We first discuss related work and formally introduce our concept of differential privacy for gossip. Then, we study two extreme cases of our parameterized gossip protocol: the standard push protocol, which we show is not private, and a privacy-optimal but slow protocol. This leads us to investigate how to better control the trade-off between speed and privacy. Finally, we present our empirical study and conclude by discussing open questions.

For pedagogical reasons, we keep our model relatively simple to avoid unnecessary technicalities in the derivation and presentation of our results. For completeness, we discuss the impact of possible extensions (e.g., information observed by the adversary, malicious behavior, termination criterion) in the full version of this paper [6]. Due to space limitations, some detailed proofs are also deferred to the full version.

2 Background and Related Work

2.1 Gossiping

The idea of disseminating information in a distributed system by having each node *push* messages to a randomly chosen neighbor, initially coined the *random phone-call model*, dates back to even before the democratization of the Internet [39]. Such protocols, later called *gossip*, *epidemic* or *rumor spreading*, were for instance applied to ensure the consistency of a replicated database system [13]. They have gained even more importance when argued to model spreading of infectious diseases [29] and information dissemination in social networks [14, 27]. Gossip protocols can also be used to compute aggregate queries on a database distributed across the nodes of a network [34, 9], and have recently become popular in federated machine learning [32] to optimize cost functions over data distributed across a large set of peers [15, 12, 43, 35]. Gossip protocols differ according to their interaction schemes, i.e., *pull* or *push*, sometimes combining both [33, 36, 2].

In this work, we focus on the classical *push* form in the standard case of a *complete* graph with n nodes (labeled from 0 to $n-1$). We now define its key communication primitive. Denoting by I the set of informed nodes, `tell_gossip`(i, I) allows an informed node $i \in I$ to tell the information to another node $j \in \{0, ..., n-1\}$ chosen uniformly at random. `tell_gossip`(i, I) returns j (the node that received the message) and the updated I (the new set of informed nodes that includes j). Equipped with this primitive, we can now formally define the class of gossip protocols that we consider in this paper.

▶ **Definition 1** (Gossip protocols). *A gossip protocol on a complete graph is one that (a) terminates almost surely, (b) ensures that at the end of the execution the set of informed nodes I is equal to $\{0, ..., n-1\}$, and (c) can modify I only through calls to* `tell_gossip`.

2.2 Locating the Source of the Gossip

Determining the source of a gossip is an active research topic, especially given the potential applications to epidemics and social networks, see [31] for a recent survey. Existing approaches have focused so far on building *source location attacks* that compute or approximate the

maximum likelihood estimate of the source given some observed information. Each approach typically assumes a specific kind of graphs (e.g., trees, small world, *etc.*), dissemination model and observed information. In *rumor centrality* [41], the gossip communication graph is assumed to be fully observed and the goal is to determine the *center* of this graph to deduce the node that started the gossip. Another line of work studies the setting in which some nodes are *curious sensors* that inform a central entity when they receive a message [38]. Gossiping is assumed to happen at random times and the source node is estimated by comparing the different timings at which information reaches the sensors. The proposed attack is natural in trees but does not generalize to highly connected graphs. The work of [22] focuses on hiding the source instead of locating it. The observed information is a snapshot of who has the rumor at a given time. A specific dissemination protocol is proposed to hide the source but the privacy guarantees only hold for tree graphs.

We emphasize that the privacy guarantees (i.e., the probability not to be detected) that can be derived from the above work only hold under the specific attacks considered therein. Furthermore, all approaches rely on maximum likelihood and hence assume a uniform prior on the probability of each node to be the source. The guarantees thus break if the adversary knows that some of the nodes could not have started the rumor, or if he is aware that the protocol is run twice from the same source.

We note that other problems at the intersection of gossip protocols and privacy have been investigated in previous work, such as preventing unintended recipients from learning the rumor [25], and hiding the initial position of agents in a distributed system [28].

2.3 Differential Privacy

While we borrow ideas from the approaches mentioned above (e.g., we assume that a subset of nodes are curious sensors as in [38]), we aim at studying the fundamental limits of *any* source location attack by measuring the amount of information leaked by a gossip scheme about the identity of the source. For this purpose, a general and robust notion of privacy is required. *Differential privacy* [16, 18] has emerged as a gold standard for it holds independently of any assumption on the model, the computational power, or the background knowledge that the adversary may have. Differentially private protocols have been proposed for numerous problems in the fields of databases, data mining and machine learning: examples include computing aggregate and linear counting queries [18], releasing and estimating graph properties [37, 42], clustering [30], empirical risk minimization [10] and deep learning [1].

In this work, we consider the classic version of differential privacy which involves two parameters $\epsilon, \delta \geq 0$ that quantify the privacy guarantee [17]. More precisely, given any two databases $\mathcal{D}_1$ and $\mathcal{D}_2$ that differ in at most one record, a randomized information release protocol $\mathcal{P}$ is said to guarantee (ϵ, δ)-differential privacy if for any possible output S:

$$p(\mathcal{P}(\mathcal{D}_1) \in S) \leq e^{\epsilon} p(\mathcal{P}(\mathcal{D}_2) \in S) + \delta, \tag{1}$$

where $p(E)$ denotes the probability of event E. Parameter ϵ places a bound on the ratio of the probability of any output when changing one record of the database, while parameter δ is assumed to be small and allows the bound to be violated with small probability. When $\epsilon = 0$, δ gives a bound on the total variation distance between the output distributions, while $\delta = 0$ is sometimes called "pure" ϵ-differential privacy. DP guarantees hold regardless of the adversary and its background knowledge about the records in the database. In our context, the background information could be the knowledge that the source is among a subset of all nodes. Robustness against such background knowledge is crucial in some applications, for instance when sharing secret information that few people could possibly know or when the

source of an epidemic is known to be among people who visited a certain place. Another key feature of differential privacy is *composability*: if (ϵ, δ)-differential privacy holds for a release protocol, then querying this protocol two times about the same dataset satisfies $(2\epsilon, 2\delta)$-differential privacy. This is important for rumor spreading as it enables to quantify privacy when the source propagates multiple rumors that the adversary can link to the same source (e.g., due to the content of the message). We will see in Section 6 that these properties are essential in practice to achieve robustness to concrete source location attacks.

Existing differentially private protocols typically introduce additional *perturbation* (also called *noise*) to hide critical information [18]. In contrast, an original aspect of our work is that we will solely rely on the *natural* randomness and limited observability brought by gossip protocols to guarantee differential privacy.

3 A Model of Differential Privacy for Gossip Protocols

Our first contribution is a precise mathematical framework for studying the fundamental privacy guarantees of gossip protocols. We formally define the inputs of the gossip protocols introduced in Definition 1, the outputs observed by the adversary during their execution, and the privacy notions we investigate. To ease the exposition, we adopt the terminology of information dissemination, but we sometimes illustrate the ideas in the context of epidemics.

3.1 Inputs and Outputs

As described in Section 2.3, differential privacy is a probabilistic notion that measures the privacy guarantees of a protocol based on the variations of its *output* distribution for a change in its *input*. In this paper, we adapt it to our gossip context. We first formalize the *inputs* and *outputs* of gossip protocols, in the case of a *single piece of information to disseminate* (multiple pieces can be addressed through composition, see Section 2.3). At the beginning of the protocol, a single node, the source, has the information (the gossip, or rumor). This node defines the input of the gossip protocol, and it is the actual "database" that we want to protect. Therefore, in our context, input databases in Equation (1) have only 1 record, which contains the identity of the source (an integer between 0 and $n-1$). Therefore, all possible input databases differ in at most one record, and differential privacy aims at protecting the content of the database, i.e., which node started the rumor.

We now turn to the outputs of a gossip protocol. The execution of a protocol generates an ordered sequence S_{omni} of pairs (i, j) of calls to `tell_gossip` where $(S_{\text{omni}})_t$ corresponds to the t-th time the `tell_gossip` primitive has been called, i is the node on which `tell_gossip` was used and j the node that was told the information. If several calls to `tell_gossip` happen simultaneously, ties are broken arbitrarily. We assume that the messages are received in the same order that they are sent. This protocol can thus be seen as an epidemic population protocol model [4] in which nodes interact using `tell_gossip`. The sequence S_{omni} corresponds to the output that would be observed by an omniscient entity who could eavesdrop on all communications. It is easy to see that, for any execution, the source can be identified exactly from S_{omni} simply by retrieving $(S_{\text{omni}})_0$.

In this work, we focus on adversaries that monitor a set of *curious nodes* $\mathcal{C}$ of size f, i.e. they observe all communications involving a curious node. This model, previously introduced in [38], is particularly meaningful in large distributed networks: while it is unlikely that an adversary can observe the full state of the network at any given time, compromising or impersonating a subset of the nodes appears more realistic. The number of curious nodes is directly linked with the *release mechanism* of DP: while the final state of the system is

always the same (everyone knows the rumor), the information released depends on which messages were received by the curious nodes during the execution. Formally, the output disclosed to the adversary during the execution of the protocol, i.e., the information he can use to try to identify the source, is a subsequence of S_{omni} as defined below.

▶ **Assumption 2.** *The sequence S observed by the adversary through the (random) execution of the protocol is a (random) subsequence $S = ((S_{\text{omni}})_t | (S_{\text{omni}})_t = (i, j)$ with $j \in \mathcal{C})$, that contains all messages sent to curious nodes. The adversary has access to the relative order of tuples in S, which is the same as in S_{omni}, but not to the index t in S_{omni}.*

It is important to note that the adversary does not know which messages were exchanged between non-curious nodes. In particular, he does not know how many messages were sent in total at a given time. As we focus on complete graphs, knowing which curious node received the rumor gives no information on the source node. For a given output sequence S, we write $S_t = i$ to denote that the t-th `tell_gossip` call in S originates from node i. Omitting the dependence on S, we also denote $t_i(j)$ the time at which node j first receives the message (even for the source) and $t_d(j)$ the time at which j first communicates with a curious node.

The ratio f/n of curious nodes determines the probability of the adversary to gather information (the more curious nodes, the more information leaks). For a fixed f, the adversary only becomes weaker as the network grows bigger. Since we would like to study adversaries with fixed power, unless otherwise noted we make the following assumption.

▶ **Assumption 3.** *The ratio of curious nodes f/n is constant.*

Finally, we emphasize that we do not restrict the computational power of the adversary.

3.2 Privacy Definitions

We now formally introduce our privacy definitions. The first one is a direct application of differential privacy (Equation 1) for the inputs and outputs specified in the previous section. To ease notations, we denote by I_0 the source of the gossip (the set of informed nodes at time 0), and for any given $i \in \{0, ..., n-1\}$, we denote by $p_i(E) = p(E | I_0 = \{i\})$ the probability of event E if node i is the source of the gossip. The protocol is therefore abstracted in this notation. Denoting by $\mathcal{S}$ the set of all possible outputs, we say that a gossip protocol is (ϵ, δ)-differentially private if:

$$p_i(S) \leq e^{\epsilon} p_j(S) + \delta, \quad \forall S \subset \mathcal{S}, \ \forall i, j \in \{0, ..., n-1\}, \tag{2}$$

where $p(S)$ is the probability that the output belongs to the set S. This formalizes a notion of *source indistinguishability* in the sense that any set of output which is likely enough to happen if node i starts the gossip (say, $p_i(S) \geq \delta$) is almost as likely (up to a e^{ϵ} multiplicative factor) to be observed by the adversary regardless of the source. Note however that when $\delta > 0$, this definition can be satisfied for protocols that release the identity of the source (this can happen with probability δ). To capture the behavior under unlikely events, we also consider the complementary notion of *c-prediction uncertainty* for $c > 0$, which is satisfied if for a uniform prior $p(I_0)$ on source nodes and any $i \in \{0, ..., n-1\}$:

$$p(I_0 \neq \{i\} | S) / p(I_0 = \{i\} | S) \geq c, \tag{3}$$

for any $S \subset \mathcal{S}$ such that $p_i(S) > 0$. Prediction uncertainty guarantees that no observable output S (however unlikely) can identify a node as the source with large enough probability: it ensures that the probability of success of any source location attack is upper bounded

by $1/(1+c)$. This holds in particular for the maximum likelihood estimate. Prediction uncertainty does not have the robustness of differential privacy against background knowledge, as it assumes a uniform prior on the source. While it can be shown that $(\epsilon, 0)$-DP with $\epsilon > 0$ implies prediction uncertainty, the converse is not true. Indeed, prediction uncertainty is satisfied as soon as no output identifies any node with enough probability, without necessarily making all pairs of nodes indistinguishable as required by DP. We will see that prediction uncertainty allows to rule out some naive protocols that have nonzero probability of generating sequences which reveal the source with certainty.

Thanks to the symmetry of our problem, we consider without loss of generality that node 0 starts the rumor ($I_0 = \{0\}$) and therefore we will only need to verify Equations (2) and (3) for $i = 0$ and $j = 1$.

4 Extreme Privacy Cases

In this section, we study the fundamental limits of gossip in terms of privacy. To do so, we parameterize gossip protocols by a muting parameter $s \in [0, 1]$, as depicted in Algorithm 1. We thereby capture, within a generic framework, a large family of protocols that fit Definition 1 and work as follows. They maintain a set A of *active nodes* (initialized to the source node) which spread the rumor asynchronously and in parallel: this is modeled by the fact that at each step of the protocol, a randomly selected node $i \in A$ invokes the `tell_gossip` primitive to send the rumor to another node (which in turn becomes active), while i also stays active with probability s. This is illustrated in Figure 1. The muting parameter s can be viewed as a randomized version of *fanout* in [21].[1] Algorithm 1 follows the population protocol model [4], and is also related to the SIS epidemic model [29] but in which the rumor never dies regardless of the value of $s \in [0, 1]$ (there always remain some active nodes). Although we present it from a centralized perspective, we emphasize that Algorithm 1 is asynchronous and can be implemented by having active nodes wake up following a Poisson process.

In the rest of this section, we show that extreme privacy guarantees are obtained for extreme values of the muting parameter s.

4.1 Standard Push has Minimal Privacy

The natural case to study first in our framework is when the muting parameter is set to $s = 1$: this corresponds to the standard push protocol [39] in which nodes always keep emitting after they receive the rumor. Theorem 4 shows that, surprisingly, the privacy guarantees of this protocol become arbitrarily bad as the size of the graph increases (keeping the fraction of curious nodes constant).

▶ **Theorem 4** (Standard push is not differentially private). *If Algorithm 1 with $s = 1$ guarantees (ϵ, δ)-DP for all values of n and constant $\epsilon < \infty$, then $\delta = 1$.*

This result may seem counter-intuitive at first since one could expect that it would be more and more difficult to locate the source when the size of the graph increases. Yet, since the ratio of curious nodes is kept constant, this result comes from the fact that the event $\{t_d(0) \leq t_i(1)\}$ (node 0 communicates with a curious node before node 1 gets the message) becomes more and more likely as n grows, hence preventing any meaningful differential

[1] Unlike in the classic fanout, nodes start to gossip again each time they receive a message instead of deactivating permanently.

Algorithm 1 Parameterized Gossip.

Require: n {Number of nodes}, k {Source node}, s {Probability for a node to remain active}
Ensure: $I = \{0, \dots, n-1\}$ {All nodes are informed}
1: $I \leftarrow \{k\}$, $A \leftarrow \{k\}$
2: **while** $|I| < n$ **do**
3: Sample i uniformly at random from A
4: $A \leftarrow A \setminus \{i\}$ with probability $1 - s$
5: $j, I \leftarrow$ `tell_gossip`(i, I), $A \leftarrow A \cup \{j\}$
6: **end while**

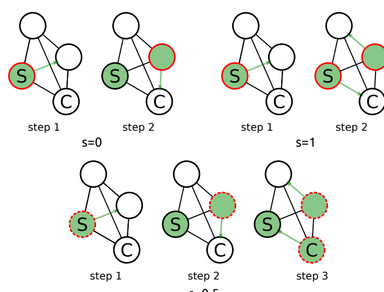


Figure 1 *Left:* Parameterized Gossip. *Right:* Illustration of the role of muting parameter s. `S` indicates the source and `C` a curious node. Green nodes know the rumor, and red circled nodes are active. When $s = 0$, there is only one active node at a time, which always stops emitting after telling the gossip. In the case $s = 1$, nodes always remain active once they know the rumor (this is the standard push gossip protocol [39]). When $0 < s < 1$, each node remains active with probability s after each communication.

privacy guarantee when n is large enough. The proof can be found in [6]. Theorem 4 clearly highlights the fact that the standard gossip protocol ($s = 1$) is not differentially private in general. We now turn to the other extreme case, where the muting parameter $s = 0$.

4.2 Muting After Infecting has Maximal Privacy

We now study the privacy guarantees of generic Algorithm 1 when $s = 0$. In this protocol, nodes forward the rumor to exactly one random neighbor when they receive it and then stop emitting until they receive the rumor again. Intuitively, this is good for privacy: the source changes and it is quickly impossible to recover which node started the gossip (as initial conditions are quickly forgotten). In fact, once the source tells the rumor once, the state of the system (the set of active nodes, which in this case is only one node) is completely independent from the source. A similar idea was used in the protocol introduced in [22].

The following result precisely quantifies the privacy guarantees of Algorithm 1 with parameter $s = 0$ and shows that it is *optimally private* among all gossip protocols (in the precise sense of Definition 1).

▶ **Theorem 5.** *Let $\epsilon \geq 0$. For muting parameter $s = 0$, Algorithm 1 satisfies (ϵ, δ)-differential privacy with $\delta = \frac{f}{n}\left(1 - \frac{e^\epsilon - 1}{f}\right)$ and c-prediction uncertainty with $c = \frac{n}{f+1} - 1$. Furthermore, these privacy guarantees are optimal among all gossip protocols.*

Proof of Theorem 5. We start by proving the fundamental limits on the privacy of any gossip protocol, and then prove matching guarantees for Algorithm 1 with $s = 0$.

(Fundamental limits in privacy) Proving a lower bound on the differential privacy parameters can be achieved by finding a set of possible outputs S (here, a set of ordered sequences) such that $p_0(S) \geq p_1(S)$. Indeed, a direct application of the definition of Equation (2) yields that given any gossip protocol, $S \subset \mathcal{S}$ and $w_0, w_1 \in \mathbb{R}$ such that $w_0 \leq p_0(S)$ and $p_1(S) \leq w_1$, if the protocol satisfies (ϵ, δ) differential privacy then $\delta \geq w_0 - e^\epsilon w_1$.

The proofs need to consider all the messages sent and then distinguish between the ones that are disclosed (sent to curious nodes) and the ones that are not.

Since $I = \{0\}$ then `tell_gossip` is called for the first time by node 0 and it is called at least once otherwise the protocol terminates with $I = \{0\}$, violating the conditions of Definition 1. We denote by $S^{(0)}$ the set of output sequences such that $S_0 = 0$ (i.e., 0 is the first to communicate with a curious node). We also define the event $T_0^c = \{t_d(0) \neq 0\}$ (the source does not send its first message to a curious node). For all $i \notin \mathcal{C} \cup \{0\}$, we have that $p_0(S_0 = i|T_0^c) \leq p_0(S_0 = 0|T_0^c)$ since $p_0(A_1 = \{0\}) = p_0(i \in A_1)$, where A_1 is the set of active nodes at time 1. From this inequality we get

$$\textstyle\sum_{i \notin \mathcal{C}} p_0(S_0 = 0|T_0^c) \geq \sum_{i \notin \mathcal{C}} p_0(S_0 = i|T_0^c) = 1 \geq \sum_{i \notin \mathcal{C}} p_0(S_0 = 1|T_0^c),$$

where the equality comes from the fact that $S_0 = i$ for some $i \notin \mathcal{C}$. The second inequality comes from the fact that $p_j(S_0 = i|T_0^c) = p_j(S_0 = k|T_0^c)$ for all $i, k \neq j$. Therefore, we have $p_0(S_0 = 0|T_0^c) \geq \frac{1}{n-f}$ and $p_0(S_0 = 1|T_0^c) \leq \frac{1}{n-f}$. Combining the above expressions, we derive the probability of $S^{(0)}$ when 0 started the gossip. We write $p_0(S^{(0)}) = p_0(S^{(0)}, t_d(0) = 0) + p_0(S^{(0)}, T_0^c)$ and then, since $p_0\big(S^{(0)}|t_d(0) = 0\big) = 1$:

$$p_0\big(S^{(0)}\big) = p_0\big(t_d(0) = 0\big)p_0\big(S^{(0)}|t_d(0) = 0\big) + p_0\big(S^{(0)}|T_0^c\big)p_0\big(T_0^c\big) \geq \frac{f}{n} + \frac{1}{n-f}\Big(1 - \frac{f}{n}\Big)$$

In the end, $p_0(S^{(0)}) \geq \frac{f}{n} + \frac{1}{n}$. If node 1 initially has the message, we do the same split and obtain $p_1(S^{(0)}|t_d(0) = 0) = 0$ and so $p_1(S^{(0)}) = p_1(T_0^c)p_1(S^{(0)}|T_0^c) \leq \frac{1}{n}$.

The upper bound on prediction uncertainty is derived using the same quantities:

$$\frac{p(I_0 \neq 0|S^{(0)})}{p(I_0 = 0|S^{(0)})} = \sum_{i \notin \mathcal{C} \cup \{0\}} \frac{p_i(S^{(0)})}{p_0(S^{(0)})} \leq (n - f - 1)\frac{p_1(S^{(0)})}{p_0(S^{(0)})} \leq \frac{n - f - 1}{f + 1} = \frac{n}{f + 1} - 1.$$

Note that we have never assumed that curious nodes knew how many messages were sent at a given point in time. We have only bounded the probability that the source is the first node that sends a message to curious nodes.

(Matching guarantees for Algorithm 1 with $s = 0$) For this protocol, the only outputs S such that $p_0(S) \neq p_1(S)$ are those in which $t_d(0) = 0$ or $t_d(1) = 0$. We write:

$$p_0(S_0 = 0) = p_0(t_d(0) = 0)p_0(S_0 = 0|t_d(0) = 0) + p_0(T_0^c)p_0(S_0 = 0|T_0^c).$$

For any $i \notin \mathcal{C}$ where $\mathcal{C}$ is the set of curious nodes, we have that $p_0(S_0 = 0|T_0^c) = p_0(S_0 = i|T_0^c) = \frac{1}{n-f}$. Indeed, given that $t_d(0) \neq 0$, the node that receives the first message is selected uniformly at random among non-curious nodes, and has the same probability to disclose the gossip at future rounds. Plugging into the previous equation, we obtain:

$$p_0(S_0 = 0) = \frac{f}{n} + \Big(1 - \frac{f}{n}\Big)\frac{1}{n - f} = \frac{f + 1}{n}.$$

For any other node $i \notin \mathcal{C} \cup \{0\}$, $p_0(S_0 = i) = p_0(T_0^c)p_0(S_0 = i|T_0^c) = \frac{1}{n}$ because $p_0(S_0 = i|t_d(0) = 0) = 0$. Combining these results we get $p_0(S^{(0)}) \leq e^\epsilon p_1(S^{(0)}) + \delta$ for any $\epsilon > 0$ and $\delta = \frac{f}{n}(1 - \frac{e^\epsilon - 1}{f})$. By symmetry, we make a similar derivation for $S^{(1)}$.

To prove the prediction uncertainty result, we use the differential privacy result with $e^\epsilon = f + 1$ (and thus $\delta = 0$) and write that for any $S \in \mathcal{S}$:

$$\frac{p(I_0 \neq 0|S)}{p(I_0 = 0|S)} = \sum_{i \notin \mathcal{C} \cup \{0\}} \frac{p_i(S)}{p_0(S)} \geq (n - f - 1)e^{-\epsilon} = \frac{n}{f + 1} - 1. \qquad ◀$$

Theorem 5 establishes *matching upper and lower bounds* on the privacy guarantees of gossip protocols. More specifically, it shows that setting the muting parameter to $s = 0$ provides strong privacy guarantees that are in fact optimal. Note that in the regime where $\epsilon = 0$ (where DP corresponds to the total variation distance), δ cannot be smaller than the proportion of curious nodes. This is rather intuitive since the source node has probability at least f/n to send its first message to a curious node. However, one can also achieve differential privacy with δ much smaller than f/n by trading-off with $\epsilon > 0$. In particular, the *pure* version of differential privacy ($\delta = 0$) is attained for $\epsilon \approx \log f$, which provides good privacy guarantees when the number of curious nodes is not too large. Furthermore, even though the probability of disclosing *some* information is of order f/n, prediction uncertainty guarantee shows that an adversary with uniform prior always has a high probability of making a mistake when predicting the source. Crucially, these privacy guarantees are made possible by the *natural* randomness and partial observability of gossip protocols.

▶ Remark 6 (Special behavior of the source). A subtle but key property of Algorithm 1 is that the source follows the same behavior as other nodes. To illustrate how violating this property may give away the source, consider this natural protocol: the source node transmits the rumor to one random node and stops emitting, then standard push (Algorithm 1 with $s = 1$) starts from the node that received the information. While this *delayed start gossip protocol* achieves optimal differential privacy in some regimes, it is fundamentally flawed. In particular, it does not guarantee prediction uncertainty in the sense that $c \to 0$ as the graph grows. Indeed, the adversary can identify the source with high probability by detecting that it communicated only once and then stopped emitting for many rounds. We refer to the full version of this paper [6] for the formal proof.

5 Privacy vs. Speed Trade-offs

While choosing $s = 0$ achieves optimal privacy guarantees, an obvious drawback is that it leads to a very slow protocol since only one node can transmit the rumor at any given time. It is easy to see that the number of gossip operations needed to inform all nodes can be reduced to the time needed for the classical coupon collection problem: it takes $\mathcal{O}(n \log n)$ communications to inform all nodes with probability at least $1 - 1/n$ [19]. As this protocol performs exactly one communication at any given time, it needs time $\mathcal{O}(n \log n)$ to inform all nodes with high probability. This is in stark contrast to the standard push gossip protocol ($s = 1$) studied in Section 4.1 where all informed nodes can transmit the rumor in parallel, requiring only time $\mathcal{O}(\log n)$ [23].

These observations motivate the exploration of the privacy-speed trade-off (with parameter $0 < s < 1$). We first show below that nearly optimal privacy can be achieved for small values of s. Then, we study the spreading time and show that the $\mathcal{O}(\log n)$ time of the standard gossip protocol also holds for $s > 0$, leading to a sweet spot in the privacy-speed trade-off.

5.1 Privacy Guarantees

Theorem 7 conveys a $(0, \delta)$-differential privacy result, which means that apart from some unlikely outputs that may disclose the identity of the source node, most of these outputs actually have the same probability regardless of which node triggered the dissemination. We emphasize that the guarantee we obtain holds for any graph size with fixed proportion f/n of curious nodes.

▶ **Theorem 7** (Privacy guarantees for $s < 1$)**.** *For $0 < s < 1$ and any fixed $r \in \mathbb{N}^*$, Algorithm 1 with muting parameter s guarantees $(0, \delta)$-differential privacy with:*

$$\delta = 1 - (1-s) \sum_{k=0}^{\infty} s^k \left(1 - \frac{f}{n}\right)^{k+1} \leq 1 - (1 - s^r) \left(1 - \frac{f}{n}\right)^r .$$

For example, choosing $r = 1$ leads to $\delta \leq s + (1-s)\frac{f}{n}$, as reported in Table 1. Slightly tighter bounds can be obtained, but this is enough already to recover optimal guarantees as $s \to 0$.

Proof. We first consider that S is such that $t_d(0) \geq t_d(1)$. Then, $p_0(S) \leq p_1(S)$ since node 0 needs to receive the rumor before being able to communicate it to curious nodes, and Equation (2) is verified. Suppose now that S is such that $t_d(0) \leq t_d(1)$. In this case, we note t_m the first time at which the source stops to emit (which happens with probability $1 - s$ each time it sends a message). Then, we denote $F = \{t_d(0) \leq t_m\}$ (and F^c its complement). In this case, $p_0(S|F^c) \leq p_1(S|F^c)$. Indeed, conditioned on F^c, $t_d(0) \geq t_i(0)$ if node 0 is not the source and $t_d(0) \geq \max(t_m, t_i(0))$ if it is. Then, we can write:

$$p_0(S) = p_0(S, F^c) + p_0(S, F) \leq p_1(S, F^c) + p_0(F) \leq p_1(S) + p_0(F).$$

Denoting T_f the number of messages after which the source stops emitting, we write:

$$p_0(F) = \sum_{k=1}^{\infty} p_0(T_f = k) p_0(F|T_f = k) = \sum_{k=0}^{\infty} (1-s) s^k \Big(1 - \big(1 - \frac{f}{n}\big)^{k+1}\Big), \text{ for } s > 0.$$

Note that we can also write for $k \geq 1$ that $p_0(F) = p_0(F, T_f \leq k) + p_0(F, T_f > k)$, and so:

$$p_0(F) \leq (1 - s^k)\Big(1 - \big(1 - \frac{f}{n}\big)^k\Big) + s^k = 1 - (1 - s^k)\Big(1 - \frac{f}{n}\Big)^k . \quad ◀$$

The differential privacy guarantees given by Theorem 7 and the optimal guarantees of Theorem 5 are of the same order of magnitude when s is of order f/n. Indeed, consider $\epsilon = 0$. Then, setting $r = 1$ in Theorem 7 leads to an additive gap of $s(1 - f/n)$ between the privacy of Algorithm 1 and the optimal guarantee, showing that one can be as close as desired to the optimal privacy as long as s is chosen close enough to 0. In particular, the ratio between the privacy of Algorithm 1 and the lower bound is less than 2 for all $s \leq f/n$. This indicates that the privacy guarantees are very tight in this regime. We also recover exactly the optimal guarantee of Theorem 5 in the case $s = 0$ (without the ability to control the trade-off between ϵ and δ). Importantly, we also show that Algorithm 1 with $s < 1$ satisfies prediction uncertainty, unlike the case where $s = 1$.

▶ **Theorem 8.** *Algorithm 1 guarantees prediction uncertainty with $c = (1 - \frac{f+1}{n})(1 - s)$.*

This result is another evidence that picking $s < 1$ allows to derive meaningful privacy guarantees. The proof can be found in the full version of this paper [6].

5.2 Spreading time

We have shown that parameter s has a significant impact on privacy, from optimal ($s = 0$) to very weak ($s = 1$) guarantees. Yet, s also impacts the spreading time: the larger s, the more active nodes at each round. This is highlighted by the two extreme cases, for which the spreading time is already known and exhibits a large gap: $\mathcal{O}(\log n)$ for $s = 1$ and $\mathcal{O}(n \log n)$ for $s = 0$. To establish whether we can obtain a protocol that is both private and fast, we need to characterize the spreading time for the cases where $0 < s < 1$.

The key result of this section is to prove that the logarithmic speed of the standard push gossip protocol holds more generally for all $s > 0$. This result is derived from the fact that the ability to forget does not prevent an *exponential growth* phase. What changes is that the population of active nodes takes approximately $1/s$ rounds to double instead of 1 for standard gossip. For ease of presentation, we state below the result for the synchronous version of Algorithm 1, in which the notion of *round* corresponds to iterating over the full set A. A similar result (with an appropriate notion of rounds) can be obtained for the asynchronous version given in Algorithm 1.

▶ **Theorem 9.** *For a given $s > 0$ and for all $1 > \delta > 0$ and $C \geq 1$, there exists n large enough such that the synchronous version of Algorithm 1 with parameter s sends at least $Cn \log n$ messages in $6C \log(n)/s$ rounds with probability at least $1 - \delta$.*

Proof sketch. The key argument of the proof is that the gossip process very closely follows its mean dynamics. After a transition phase of a logarithmic number of rounds, a constant fraction of the nodes (depending on s) remains active despite the probability to stop emitting after each communication. This "determinism of gossip process" has been introduced in [40], but their analysis only applies to $s = 1$. Our proof accounts for the nontrivial impact of nodes deactivation in the exponential and linear growth phase. Besides, we need to show that in the last phase, with high probability, the population never drops below a critical threshold of active nodes. The full proof is in the full version of this paper [6]. ◀

Theorem 9 shows that generic gossip with $s > 0$ still achieves a logarithmic spreading time even though nodes can stop transmitting the message. The $1/s$ dependence is intuitive since $1/s$ rounds are needed in expectation to double the population of active nodes (without taking collisions into account). Therefore, the exponential growth phase which usually takes time $\mathcal{O}(\log n)$ now takes time $\mathcal{O}(\log(n)/s)$ for $s < 1$. To summarize, we have shown that one can achieve both fast spreading and near-optimal privacy, leading to the values presented in Table 1 of the introduction.

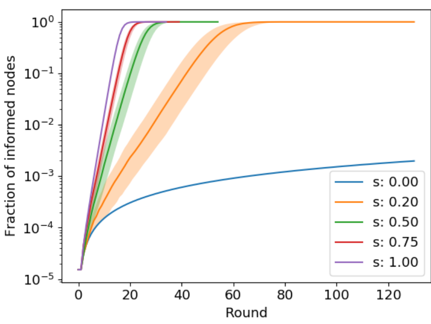
6 Empirical Evaluation

In this section, we evaluate the practical impact of s on the spreading time as well as on the robustness to source location attacks run by adversaries with background knowledge.

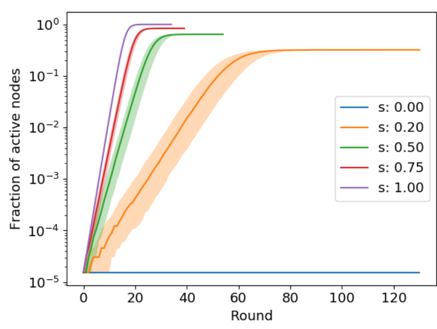
6.1 Spreading Time

To complement Theorem 9, which proves logarithmic spreading time (asymptotic in n), we run simulations on a network of size $n = 2^{16}$. The logarithmic spreading time for $s > 0$ is clearly visible in Figure 2a, where we see that the gossip spreads almost as fast for $s = 0.5$ that it does for $s = 1$. We also observe that even when s is small, the gossip remains much faster than for $s = 0$. The results in Figure 2b illustrate that the fraction of active nodes grows exponentially fast for all values of $s > 0$ and then reaches a plateau when the probability of creating a new active node is compensated by the probability of informing an already active node. Empirically, this happens when the fraction of active nodes is of order s.

We note incidentally that gossip protocols are often praised for their robustness to lost messages [3, 24]. While the protocol with $s = 0$ does not tolerate a single lost message, setting $s > 0$ improve the resilience thanks to the linear proportion of active nodes. The latter property makes it unlikely that the protocol stops because of lost messages as long as s is larger than the probability of losing messages. Of course, the protocol remains somewhat sensitive to messages lost during the first few steps.

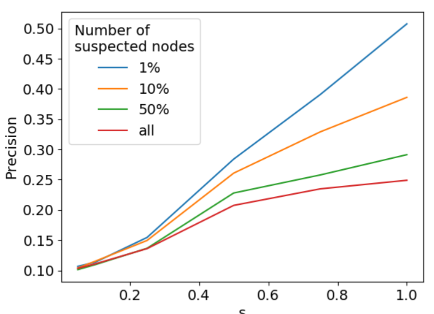


(a) Fraction of informed nodes.

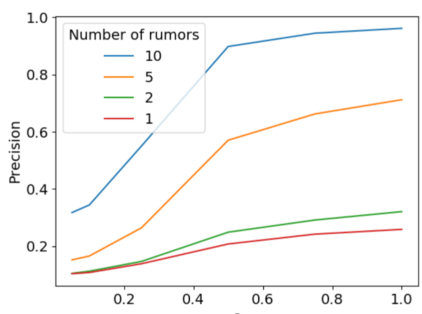


(b) Fraction of active nodes.

Figure 2 Effect of parameter s of Algorithm 1 on the spreading time for a network of $n = 2^{16}$ nodes. The curves represent median values and the shaded area represents the 10 and 90 percent confidence intervals over 100 runs. Each curve stops when all nodes are informed (and so the protocol terminates), except for $s = 0$ since the protocol is very slow in this case.



(a) Attack precision under prior information on the source.



(b) Attack precision when the source spreads multiple rumors.

Figure 3 Effect of parameter s of Algorithm 1 on the precision of source location attacks for a network of $n = 2^{16}$ node with 10% of curious nodes. Precision is estimated over 15,000 random runs.

6.2 Robustness Against Source Location Attacks

Getting an intuitive understanding of the privacy guarantees provided by Theorem 7 is not straightforward, as often the case with differential privacy. Therefore, we illustrate the effect of the muting parameter on the guarantees of our gossip protocol by simulating concrete source location attacks. We consider two challenging scenarios where the adversary has some background knowledge: either 1) prior knowledge that the source belongs to a subset of the nodes, or 2) side information indicating that the same source disseminates multiple rumors.

Prior knowledge on the source. We first consider the case where the adversary is able to narrow down the set of suspected nodes. In this case we can design a provably optimal attack, as shown by the following theorem (see [6]).

▶ **Theorem 10.** *If the adversary has a uniform prior over a subset P of nodes, i.e., $p(I_0 = i) = p(I_0 = j)$ for all $i, j \in P$ and $p(I_0 = i) = 0$ for $i \notin P$, and for some output sequence S, t_c is such that $S_{t_c} \in P$ and $S_t \notin P$ if $t < t_c$, then $p(I_0 = S_{t_c}|S) \geq p(I_0 = i|S)$ for all i.*

Theorem 10 means that under a uniform prior over nodes in P, the attack in which curious nodes predict the source to be the first node in P that communicates with them corresponds to the Maximum A Posteriori (MAP) estimator. The set P represents the prior knowledge of the adversary: he knows for sure that the source belongs to P.

Figure 3a shows the precision of this attack as a function of s for varying degrees of prior knowledge. We see that, when s is small, the prior knowledge does not improve the attack precision significantly, and that the precision remains very close to the probability that the source sends its first message to a curious node. This robustness to prior knowledge is consistent with the properties of differential privacy (see Section 2.3). On the contrary, when s is high (i.e., differential privacy guarantees are weak), the impact of the prior knowledge on the precision of the attack is much stronger.

Multiple dissemination. We investigate another scenario in which differential privacy guarantees can also provide robustness, namely when the adversary knows that the same source node disseminates multiple rumors. In this setting, analytically deriving an optimal attack is very difficult. Instead, we design an attack which leverages the fact that even though the source is not always the first node to communicate with curious nodes, with high probability it will be among the first to do so. More precisely, the curious nodes record the 10 first nodes that communicate with them in each instance (results are not very sensitive to this choice), and they predict the source to be the node that appears in the largest number of instances. In case of a tie, the curious nodes choose the node that first communicated with them, with ties broken at random. Figure 3b shows that the precision of this attack increases dramatically with the number of rumors when s is large, reaching almost sure detection for 10 rumors. Remarkably, for small values of s, the attack precision increases much more gracefully with the number of rumors, as expected from the composition property of differential privacy discussed in Section 2.3. Meaningful privacy guarantees can still be achieved as long as the source does not spread too many rumors.

7 Concluding Remarks

This paper initiates the formal study of privacy in gossip protocols to determine to which extent the source of a gossip can be traceable. Essentially: (1) We propose a formal model of anonymity in gossip protocols based on an adaptation of differential privacy; (2) We establish tight bounds on the privacy of gossip protocols, highlighting their natural privacy guarantees; (3) We precisely capture the trade-off between privacy and speed, showing in particular that it is possible to design both fast and near-optimally private gossip protocols; (4) We experimentally evaluate the speed of our protocols as well as their robustness to source location attacks, validating the relevance of our formal differential privacy guarantees in scenarios where the adversary has some background knowledge.

Our work opens several interesting perspectives. In particular, it paves the way to the study of differential privacy for gossip protocols in *general graphs*, which is challenging and requires relaxations of differential privacy in order to obtain nontrivial guarantees. We refer to the full version of this paper [6] for a discussion of these questions. Another avenue for future research is motivated by very recent work showing that hiding the source of a message can amplify differential privacy guarantees for the *content* of the message [20, 11, 5]. Unfortunately, classic primitives to hide the source of messages such as mixnets can be difficult and costly to deploy. Showing that gossip protocols can *naturally* amplify differential privacy for the message contents would make them very desirable for privacy-preserving distributed AI applications, such as those based on federated [32] and decentralized machine learning [7].

References

1 Martín Abadi, Andy Chu, Ian J. Goodfellow, H. Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. Deep learning with differential privacy. In *CCS*, 2016.

2 Huseyin Acan, Andrea Collevecchio, Abbas Mehrabian, and Nick Wormald. On the push&pull protocol for rumor spreading. *SIAM Journal on Discrete Mathematics*, 31(2):647–668, 2017.

3 Dan Alistarh, Seth Gilbert, Rachid Guerraoui, and Morteza Zadimoghaddam. How efficient can gossip be?(on the cost of resilient information exchange). In *ICALP*, 2010.

4 Dana Angluin, James Aspnes, and David Eisenstat. Fast computation by population protocols with a leader. *Distributed Computing*, 21(3):183–199, 2008.

5 Borja Balle, James Bell, Adria Gascon, and Kobbi Nissim. The Privacy Blanket of the Shuffle Model. Technical report, arXiv:1903.02837, 2019.

6 Aurélien Bellet, Rachid Guerraoui, and Hadrien Hendrikx. Who started this rumor? Quantifying the natural differential privacy guarantees of gossip protocols. *arXiv preprint arXiv:1902.07138*, 2019.

7 Aurélien Bellet, Rachid Guerraoui, Mahsa Taziki, and Marc Tommasi. Personalized and Private Peer-to-Peer Machine Learning. In *AISTATS*, 2018.

8 Petra Berenbrink, Jurek Czyzowicz, Robert Elsässer, and Leszek Gasieniec. Efficient information exchange in the random phone-call model. *Automata, Languages and Programming*, 2010.

9 Stephen Boyd, Arpita Ghosh, Balaji Prabhakar, and Devavrat Shah. Randomized gossip algorithms. *IEEE Transactions on Information Theory*, 52(6):2508–2530, 2006.

10 Kamalika Chaudhuri, Claire Monteleoni, and Anand D. Sarwate. Differentially Private Empirical Risk Minimization. *Journal of Machine Learning Research*, 12:1069–1109, 2011.

11 Albert Cheu, Adam D. Smith, Jonathan Ullman, David Zeber, and Maxim Zhilyaev. Distributed Differential Privacy via Shuffling. Technical report, arXiv:1808.01394, 2018.

12 Igor Colin, Aurélien Bellet, Joseph Salmon, and Stéphan Clémençon. Gossip dual averaging for decentralized optimization of pairwise functions. In *ICML*, 2016.

13 Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *PODC*, 1987.

14 Benjamin Doerr, Mahmoud Fouz, and Tobias Friedrich. Social networks spread rumors in sublogarithmic time. In *STOC*, 2011.

15 John C. Duchi, Alekh Agarwal, and Martin J. Wainwright. Dual Averaging for Distributed Optimization: Convergence Analysis and Network Scaling. *IEEE Transactions on Automatic Control*, 57(3):592–606, 2012.

16 Cynthia Dwork. Differential Privacy. In *ICALP*, 2006.

17 Cynthia Dwork, Krishnaram Kenthapadi, Frank McSherry, Ilya Mironov, and Moni Naor. Our data, ourselves: Privacy via distributed noise generation. In *EUROCRYPT*, 2006.

18 Cynthia Dwork and Aaron Roth. The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science*, 9(3–4):211–407, 2014.

19 Paul Erdős and Alfred Rényi. On a classical problem of probability theory. *Publ. Math. Inst. Hung. Acad. Sci.*, 1961.

20 Úlfar Erlingsson, Vitaly Feldman, Ilya Mironov, and Ananth Raghunathan abd Kunal Talwar. Amplification by Shuffling: From Local to Central Differential Privacy via Anonymity. Technical report, arXiv:1811.12469, 2018.

21 Patrick T Eugster, Rachid Guerraoui, Anne-Marie Kermarrec, and Laurent Massoulié. Epidemic information dissemination in distributed systems. *Computer*, 37(5):60–67, 2004.

22 Giulia Fanti, Peter Kairouz, Sewoong Oh, Kannan Ramchandran, and Pramod Viswanath. Hiding the rumor source. *IEEE Transactions on Information Theory*, 2017.

23 Alan M Frieze and Geoffrey R Grimmett. The shortest-path problem for graphs with random arc-lengths. *Discrete Applied Mathematics*, 10(1):57–77, 1985.

24 Chryssis Georgiou, Seth Gilbert, Rachid Guerraoui, and Dariusz R Kowalski. Asynchronous gossip. *Journal of the ACM*, 60(2):11, 2013.

25 Chryssis Georgiou, Seth Gilbert, and Dariusz R Kowalski. Confidential gossip. In *ICDCS*, 2011.

26 Mohsen Ghaffari and Calvin Newport. How to discreetly spread a rumor in a crowd. In *DISC*, 2016.

27 George Giakkoupis, Rachid Guerraoui, Arnaud Jégou, Anne-Marie Kermarrec, and Nupur Mittal. Privacy-conscious information diffusion in social networks. In *International Symposium on Distributed Computing*, pages 480–496. Springer, 2015.

28 Karol Gotfryd, Marek Klonowski, and Dominik Pajak. On location hiding in distributed systems. In *International Colloquium on Structural Information and Communication Complexity*, pages 174–192. Springer, 2017.

29 Herbert W. Hethcote. The mathematics of infectious diseases. *SIAM Review*, 42(4):599–653, 2000.

30 Zhiyi Huang and Jinyan Liu. Optimal differentially private algorithms for k-means clustering. In *PODS*, 2018.

31 Jiaojiao Jiang, Sheng Wen, Shui Yu, Yang Xiang, and Wanlei Zhou. Identifying propagation sources in networks: State-of-the-art and comparative studies. *IEEE Communications Surveys & Tutorials*, 19(1):465–481, 2017.

32 Peter Kairouz, H. Brendan McMahan, Brendan Avent, Aurélien Bellet, Mehdi Bennis, Arjun Nitin Bhagoji, Keith Bonawitz, Zachary Charles, Graham Cormode, Rachel Cummings, Rafael G. L. D'Oliveira, Salim El Rouayheb, David Evans, Josh Gardner, Zachary Garrett, Adrià Gascón, Badih Ghazi, Phillip B. Gibbons, Marco Gruteser, Zaid Harchaoui, Chaoyang He, Lie He, Zhouyuan Huo, Ben Hutchinson, Justin Hsu, Martin Jaggi, Tara Javidi, Gauri Joshi, Mikhail Khodak, Jakub Konečný, Aleksandra Korolova, Farinaz Koushanfar, Sanmi Koyejo, Tancrède Lepoint, Yang Liu, Prateek Mittal, Mehryar Mohri, Richard Nock, Ayfer Özgür, Rasmus Pagh, Mariana Raykova, Hang Qi, Daniel Ramage, Ramesh Raskar, Dawn Song, Weikang Song, Sebastian U. Stich, Ziteng Sun, Ananda Theertha Suresh, Florian Tramèr, Praneeth Vepakomma, Jianyu Wang, Li Xiong, Zheng Xu, Qiang Yang, Felix X. Yu, Han Yu, and Sen Zhao. Advances and Open Problems in Federated Learning. Technical report, arXiv:1912.04977, 2019.

33 Richard Karp, Christian Schindelhauer, Scott Shenker, and Berthold Vocking. Randomized rumor spreading. In *FOCS*, 2000.

34 David Kempe, Alin Dobra, and Johannes Gehrke. Gossip-Based Computation of Aggregate Information. In *FOCS*, 2003.

35 Anastasia Koloskova, Sebastian Stich, and Martin Jaggi. Decentralized Stochastic Optimization and Gossip Algorithms with Compressed Communication. In *ICML*, 2019.

36 Dariusz R Kowalski and Christopher Thraves Caro. Estimating time complexity of rumor spreading in ad-hoc networks. In *International Conference on Ad-Hoc Networks and Wireless*, pages 245–256. Springer, 2013.

37 Wentian Lu and Gerome Miklau. Exponential random graph estimation under differential privacy. In *KDD*, 2014.

38 Pedro C Pinto, Patrick Thiran, and Martin Vetterli. Locating the source of diffusion in large-scale networks. *Physical Review Letters*, 2012.

39 Boris Pittel. On spreading a rumor. *SIAM Journal on Applied Mathematics*, 47(1):213–223, 1987.

40 Sujay Sanghavi, Bruce Hajek, and Laurent Massoulié. Gossiping with multiple messages. *IEEE Transactions on Information Theory*, 53(12):4640–4654, 2007.

41 Devavrat Shah and Tauhid Zaman. Rumors in a network: Who's the culprit? *IEEE Transactions on Information Theory*, 2011.

42 Haipei Sun, Xiaokui Xiao, Issa Khalil, Yin Yang, Zhan Qin, Hui Wang, and Ting Yu. Analyzing subgraph statistics from extended local views with decentralized differential privacy. In *CCS*, 2019.

43 Paul Vanhaesebrouck, Aurélien Bellet, and Marc Tommasi. Decentralized collaborative learning of personalized models over networks. In *AISTATS*, 2017.

Spread of Information and Diseases via Random Walks in Sparse Graphs

George Giakkoupis
Inria, Université Rennes, CNRS, IRISA, Rennes, France
george.giakkoupis@inria.fr

Hayk Saribekyan
University of Cambridge, UK
hs586@cam.ac.uk

Thomas Sauerwald
University of Cambridge, UK
thomas.sauerwald@cl.cam.ac.uk

Abstract

We consider a natural network diffusion process, modeling the spread of information or infectious diseases. Multiple mobile agents perform independent simple random walks on an n-vertex connected graph G. The number of agents is linear in n and the walks start from the stationary distribution. Initially, a single vertex has a piece of information (or a virus). An agent becomes informed (or infected) the first time it visits some vertex with the information (or virus); thereafter, the agent informs (infects) all vertices it visits. Giakkoupis et al. [16] have shown that the spreading time, i.e., the time before all vertices are informed, is asymptotically and w.h.p. the same as in the well-studied randomized rumor spreading process, on any d-regular graph with $d = \Omega(\log n)$. The case of sub-logarithmic degree was left open, and is the main focus of this paper.

First, we observe that the equivalence shown in [16] does not hold for small d: We give an example of a 3-regular graph with logarithmic diameter for which the expected spreading time is $\Omega(\log^2 n/\log\log n)$, whereas randomized rumor spreading is completed in time $\Theta(\log n)$, w.h.p. Next, we show a general upper bound of $\tilde{O}(d \cdot \mathrm{diam}(G) + \log^3 n/d)$,[1] w.h.p., for the spreading time on any d-regular graph. We also provide a version of the bound based on the average degree, for non-regular graphs. Next, we give tight analyses for specific graph families. We show that the spreading time is $O(\log n)$, w.h.p., for constant-degree regular expanders. For the binary tree, we show an upper bound of $O(\log n \cdot \log\log n)$, w.h.p., and prove that this is tight, by giving a matching lower bound for the cover time of the tree by n random walks. Finally, we show a bound of $O(\mathrm{diam}(G))$, w.h.p., for k-dimensional grids ($k \geq 1$ is constant), by adapting a technique by Kesten and Sidoravicius [22, 23].

2012 ACM Subject Classification Mathematics of computing → Stochastic processes; Theory of computation → Random walks and Markov chains; Mathematics of computing → Graph algorithms

Keywords and phrases parallel random walks, information dissemination, infectious diseases

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.9

Related Version A full version of the paper is available at `https://hal.inria.fr/hal-02913942`.

Funding *George Giakkoupis*: Supported in part by ANR Project PAMELA (ANR16-CE23-0016-01).
Hayk Saribekyan: Gates Cambridge Scholarship programme.
Thomas Sauerwald: Supported by the ERC Grant “Dynamic March”.

[1] The tilde notation hides factors of order at most $(\log\log n)^2$.

© George Giakkoupis, Hayk Saribekyan, and Thomas Sauerwald;
licensed under Creative Commons License CC-BY

34th International Symposium on Distributed Computing (DISC 2020).
Editor: Hagit Attiya; Article No. 9; pp. 9:1–9:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

We consider the following natural diffusion process on a connected n-vertex graph G. A collection of mobile agents perform independent parallel (discrete-time) random walks on G, starting from the stationary distribution. Initially, there is a piece of information at some arbitrary source vertex. An agent learns the information the first time it visits some informed vertex (the vertex may have received the information in the same or a previous round). From that point on, the agent spreads the information to all vertices it visits. We study the time it takes before all vertices have been informed. We will refer to this process as VISIT-EXCHANGE, following the terminology of [16].

The above process suggests a simple message broadcasting algorithm for networks: Vertices correspond to processes, and agents are tokens circulated in the network. In each round, every process sends each of the tokens it received in the previous round to a random neighbor, and if the process knows the message, it transmits the message along with each token. As observed in [16], when the number of agents/tokens is linear in n, this algorithm has similar per round message complexity as standard randomized rumor spreading [14, 21], but in several graphs it outperforms the latter, due to a more fair bandwidth utilization: each edge is equally likely to be used in each round.

A second potential application of VISIT-EXCHANGE is as a basic model for the spread of diseases in populations. One can think of agents as the members of the population, where an infected member can transmit the infection to another either by direct contact, or indirectly. In the latter case a healthy individual contracts the virus by being in a place previously visited by an infected individual [25]. Alternatively, one can think of a larger population, residing on the vertices of the graphs (e.g., vertices are cities), and a few mobile individuals are responsible for transmitting the infection between different cities. Our basic model assumes perfect contagion and no recovery. It is an interesting future direction to analyze a refined model that allows probabilistic transmission and recovery.

Several works have studied the spread of information (or viruses) via mobile agents, performing random walks or more general jump processes, in discrete or continuous time, on various families of graphs [3, 8, 10, 16, 19, 20, 23, 24, 26, 28] (see Sect. 2 for an overview of this literature). In almost all of these works, the information is transmitted only between agents when they meet at a vertex, and vertices do not store information.

The work closest to ours is [16] (see also [17]). The authors consider VISIT-EXCHANGE with $\Theta(n)$ agents, starting from stationarity, and compare the spreading time to that of randomized rumor spreading [14, 21]. In the latter protocol, information is transmitted between adjacent vertices, without the use of agents, by having each vertex communicate with a random neighbor in each round. It was observed in [16] that there are graphs in which VISIT-EXCHANGE is significantly faster than randomized rumor spreading (logarithmic versus linear spreading time), and examples where the converse is true.

A main result of [16] is that on any d-regular graph with sufficiently large degree $d = \Omega(\log n)$, VISIT-EXCHANGE and randomized rumor spreading have the same asymptotic spreading time. The intuition for this result is the following. We have that: (i) the average number of agents per vertex is constant, since there are $\Theta(n)$ agents in total, (ii) all agents start from stationarity, and (iii) the graph is regular. It follows that, in every round, a constant expected number of agents depart from each vertex, to random neighbors. This should have a similar effect in the spread of information as randomized rumor spreading, where each vertex communicates with a random neighbor in each round.

The intuition above is not hard to formalize, and prove that VISIT-EXCHANGE is at least as fast as rumor spreading asymptotically:[2] If $d \geq c \log n$, for a large enough constant c, a Chernoff bound together with a union bound show that, w.h.p., for every vertex u and round $t \leq \mathrm{poly}(n)$, at least $\Omega(d)$ agents visit the neighbors of u in round $t-1$. Thus, at least one agent visits u in round t, with constant probability. This argument, however, does not extend to the case of $d = O(\log n)$. It was thus left as an open problem in [16], whether the same result holds for graphs of degree $d = O(\log n)$.

Our Contribution. First, we answer the above open question from [16] in the negative.

▶ **Observation 1.** *There is a 3-regular graph G with n vertices and diameter $\Theta(\log n)$, such that the expected spreading time of VISIT-EXCHANGE on G is $\Omega(\log^2 n/ \log\log n)$.*

The spreading time of randomized rumor spreading is $\Theta(\mathrm{diam}(G))$, w.h.p., on any constant degree graph G [14], thus it is logarithmic for the graph above. To simplify the exposition, here we only give an example of a constant-degree, *non-regular* graph G with diameter and spreading time as described in Observation 1. Consider a 3-regular graph R with n vertices and diameter $\Theta(\log n)$ (e.g., a 3-regular expander), and $\sqrt{n}$ path graphs, each of length $\log n/2$. We obtain G by connecting one of the two endpoints of each path graph, to a distinct vertex of R. The diameter of G is clearly logarithmic. The expected spreading time is $\Omega(\log^2 n/ \log\log n)$, because with constant probability, at least one path graph P contains no agents initially, and then it takes $\Omega(\log^2 n/ \log\log n)$ rounds before the endpoint of P not connected to R gets informed. We can replace the paths of length $\log n/2$ with "ladder" graphs, as detailed in [18], to construct a *regular* graph satisfying Observation 1.

A consequence of Observation 1 is that known bounds for rumor spreading do not readily apply to VISIT-EXCHANGE for low-degree regular graphs, thus new bounds are needed. In view of that, we first provide a general upper bound for VISIT-EXCHANGE for regular graphs of degree $d = O(\log n)$, in terms of the graph diameter. Then we provide tight bounds for several interesting graph families. All our results assume that the number of agents is αn, for some arbitrary constant $\alpha > 0$, and the walks start from the stationary distribution. We denote by $T(G)$ the spreading time on graph G. Since all our bounds hold for any source vertex, we do not explicitly specify a source in the notation. Moreover, we omit G and write just T, when the graph is clear from the context. We write *w.h.p. (with high probability)* to denote a probability that is at least $1 - n^{-c}$ for some constant $c > 0$.

▶ **Theorem 2.** *For any d-regular graph G with $d = O(\log n)$, $T = \tilde{O}(d \cdot \mathrm{diam}(G) + \log^3 n/d)$, w.h.p., where the tilde notation hides factors of order at most $(\log\log n)^2$.*

In the above bound, the dependence on the diameter is best possible (e.g., the spreading time along a cycle of d-cliques is proportional to the path length multiplied by d). An additive term is also needed when the diameter is sub-logarithmic, but it is not clear whether the term $\log^3 n/d$ is tight. Recall that the corresponding upper bound for randomized rumor spreading shown in [14] is $O(d \cdot (\mathrm{diam}(G) + \log n))$. Thus, it would be reasonable to guess that the right additive term is $d \cdot \log n$. However, the example in Observation 1 shows that the term must be at least $\tilde{\Omega}(\log^2 n)$. We conjecture that the tight bound is $\tilde{O}(d \cdot \mathrm{diam}(G) + \log^2 n)$.

The proof of Theorem 2 bounds the time that the information takes to spread along a given (shortest) path in the graph. We divide time into phases of length $\log^2 n$ rounds, and in each phase, we lower-bound the probability that the information spreads along a sub-path of

[2] The proof of the other direction, that rumor spreading is at least as fast as VISIT-EXCHANGE, is significantly more involved.

length $\tilde{\Omega}(\log^2 n/d)$. For $d = \omega(\log\log n)$, we show this probability to be $1 - e^{-\Omega(d)}$. Moreover, we ensure that this probability bound holds, essentially, *independently* of previous phases, by considering every other phase. We prove the bound by showing a concentration result on the number of agents at the neighborhood of each individual vertex in the sub-path, at each round of the phase, and then applying a union bound. To boost the above probability to $1 - e^{-\Omega(\log n)}$, we need $\log n/d$ phases, which yields the $\log^3 n/d$ term of the bound. For the case of $d = O(\log\log n)$, we use a similar approach, but argue instead about the number of agents that visit each vertex in the sub-path over an interval of multiple rounds (instead of looking at its neighborhood at each round). The main technical tool we use is an upper bound on the return probability from [27].

For non-regular graphs, a similar analysis as for Theorem 2 yields the following result.

▶ **Theorem 3.** *For any graph G with average degree d_{avg} and minimum degree $d_{\min} = \Omega(d_{\text{avg}})$, $T = O(d_{\text{avg}} \cdot \log^2 n \cdot (\text{diam}(G) + \log n))$, w.h.p.*

Even though this bound is likely not tight, it is interesting because there is no analogue of it for randomized rumor spreading. For example, in the graph consisting of two stars with their centers connected by an edge [16], for which $d_{\text{avg}} = O(1)$, randomized rumor spreading takes linear expected time, whereas VISIT-EXCHANGE takes logarithmic time w.h.p. (and Theorem 3 gives a $\text{poly}(\log n)$ bound).

Next we show that the spreading time on expanders is optimal, i.e., logarithmic.

▶ **Theorem 4.** *For any d-regular expander G with $d \geq 3$ constant, $T = O(\log n)$, w.h.p.*

Unlike the proof of Theorem 2, where we argue about individual vertices, to prove Theorem 4 we argue about the set of all informed vertices at time t, precisely, the subset S_t of informed vertices with at least one uninformed neighbor. By the expansion property, S_t contains at least a constant fraction of all informed vertices. We claim that a constant fraction of vertices in S_t are visited by some agent between rounds t and $t + r$, w.h.p., for any t and a large enough constant r. Since d is constant, this implies that the number of informed vertices increases by a constant factor every r rounds. To prove the above claim, we argue that the probability a given agent visits S_t between t and $t + r$ is proportional to $k = |S_t|$ and r. Thus, S_t is not visited by sufficiently many agents in these r rounds with probability decreasing exponentially in $r \cdot k$. Next, we consider all possible instantiations of S_t, and apply a union bound. Since the set of informed vertices at any time is connected, the number of different instantiations of S_t can be bounded by $d^{\Theta(k)}$. Since d is constant, the claim follows by choosing constant r large enough.

We currently do not know how to extend Theorem 4 to regular expanders of degree $\omega(1) \leq d \leq O(\log n)$ (for $d = \Omega(\log n)$, the result follows from [16]).

Next we study trees. Let $R_{b,h}$ denote a rooted b-ary tree where each vertex at distance less than h from the root has b children and all leaves are at distance h from the root. The total number of vertices is $n = (b^{h+1} - 1)/(b - 1)$.

▶ **Theorem 5.** *For any b-ary tree $R_{b,h}$ with $b \geq 2$, $T = O(h \log h + \log n)$, w.h.p. Furthermore, for the binary tree $R_{2,h}$, $T = \Omega(h \log h) = \Omega(\log n \cdot \log\log n)$, w.h.p.*

Note that the spreading time on $R_{b,h}$ of the push-only version of randomized rumor spreading is $\Theta(b \log n)$, w.h.p. Thus, VISIT-EXCHANGE is slower than push for small b, and faster for larger b. Another interesting implication of Theorem 5 is that the cover time of the tree by n random walks starting from stationarity has a super-linear speedup, compared to the cover time for a single random walk, which is $\Omega(n \log^2 n)$. Our analysis suggests a deeper connection between the cover time (or other quantities) of multiple random walks and the spreading time of VISIT-EXCHANGE, which might deserve further study.

We give now an overview of the proof of Theorem 5, for the binary tree case; the case of $b > 2$ is similar. To prove the upper bound, we fix a path between the root r and a vertex u at distance at most $h - \log h$. We show that information spreads between r and u in at most $O(\log n)$ rounds w.h.p., by showing that agents arrive at each vertex v of the path at roughly constant rate, independently of the other vertices in the path. To achieve this independence, for each v we identify a subset S_v of the descendants of v at distance $\log h$, and count only visits to v by agents that are in S_v a number of $\Theta(\log h)$ rounds ago, and in the meantime have not walked past v. To show a constant rate, instead of $1/\Theta(\log h)$, a careful pipeline argument is used. To bound the time to spread the information in the last $\log h$ levels of the tree, we bound the cover time of a tree of height $\log h$ by h walks starting from the root, which takes $O(h \log h)$ steps w.h.p. (in n). Finally, to show the lower bound of Theorem 5, we bound from below the cover time of the tree by n random walks starting from stationarity.

Last we show that the spreading time on grids is optimal, i.e., asymptotically equal to the diameter. Let $G_{k,n}$ denote the k-dimensional grid with side length $n^{1/k}$ and n vertices in total (for simplicity, we assume $n^{1/k}$ is an integer).

▶ **Theorem 6.** *For any grid graph $G_{k,n}$ that has a constant number of dimensions $k \geq 1$, $T = \Theta(\mathrm{diam}(G))$, w.h.p.*

A weaker version of this result, with additional $\log \log n$ factors, follows from Theorem 2. To get rid of these extra factors, we employ a much more fine-grained analysis.

Our proof of Theorem 6 uses a technique developed by Kesten and Sidoravicius [22, 23], who proved a similar bound for a continuous-time diffusion process, in which information spreads between agents when they meet (it is not stored on vertices). For our discussion here, we assume the 1-dimensional case, i.e., the n-path. We consider a sequence of $\Theta(\log \log n)$ tessellations of space-time (up to time linear in n), where each tessellation consists of square blocks; the length of the block side is constant in the first tessellation, and increases exponentially in each subsequent tessellation. Let Δ be the side length of a block, and let (v, t) be its bottom left corner; i.e., the block contains all points $(v + j, t + j')$, $0 \leq j, j' < \Delta$. Roughly speaking, the block is "good" if a sufficiently large neighborhood of the vertices in the block (namely, vertices $v - 3\Delta$ up to $v + 4\Delta$) is sufficiently densely populated by agents at time $t - \Delta$. This implies that any space-time point in the block has a good probability of containing some agent. Starting from the last tessellation, for which $\Delta = \Theta(\log n)$ and all blocks are good w.h.p., we recursively bound the number of bad blocks in each tessellation, concluding that at most a constant fraction of all blocks in the first tessellation are bad. Moreover, blocks that are far from each other by at least some constant distance (in space-time), satisfy the property of being good independently of one another. We can then use this result to show that any possible information path contains sufficiently many good blocks, which guarantees that information reaches from one end of the n-path to the other. We note that various aspects of our proof are simpler that in the original proof of Kesten and Sidoravicius, mainly because our process stores the information at vertices, resulting in information paths that are easier to analyse.

Road-map. In Sect. 2 we give an overview of the related work. In Sect. 3 we prove Theorem 2, and in Sect. 4 we prove the upper bound of Theorem 5. Due to space limitations, the proofs of the remaining results are only available in the full version of the paper [18].

2 Related Work

Independent parallel random walks have been studied since the late 70s [1], mainly as a way to speed-up cover and hitting times and related graph problems [2,4,7,8,12,13]. Similarly, randomized rumor spreading, where information exchange occurs between adjacent vertices (e.g., via push, pull, or push-pull), has been studied for the past 35 years [9,14,21], with the more recent results studying the spreading time in social networks [11], and bounds with graph expansion [5].

A closely related diffusion process to ours is the one where information is not stored on vertices, but is transmitted directly between agents when they meet, and initially a single agent is informed. Naturally, the spreading time in this setting is the time until all *agents* are informed. Several works have studied this process [8,10,16,26,28]. Dimitriou et al. [10], observed that on any graph the expected spreading time is $O(t^* \log m)$, where m is the number of agents (placed at arbitrary vertices, initially), and t^* is the maximum expected meeting time of two walks; this bound is tight for some graphs. Better bounds were also provided for the complete graph and expanders. Cooper et al. [8] showed (among other results) that the expected spreading time on a random d-regular graph converges to $\frac{2n \ln m}{m} \cdot \frac{d-1}{d-2}$, for most starting positions of the m agents. Pettarin et al. [28,29] considered the k-dimensional grid, $G_{k,n}$, for $k \in \{1,2\}$, and showed that the spreading time is $\tilde{\Theta}(n/\sqrt{m})$,[3] w.h.p., for m agents starting from stationarity. Lam et al. [26] studied the same problem for $k \geq 3$ dimensions, and showed a phase transition depending on m: for large m the spreading time is $\tilde{\Theta}(n^{1+1/k}/\sqrt{m})$, while for small m it is $\tilde{\Theta}(n/m)$. Giakkoupis et al. [16,17] considered the process on d-regular graphs, with $m = \Theta(n)$ agents starting from stationarity, and showed that, on any d-regular graph with $d = \Omega(\log n)$, the spreading time is asymptotically at least as large as for VISIT-EXCHANGE, and in some cases strictly larger.

Kesten and Sidoravicius [23,24] studied a continuous-time variant of the above process on the infinite grid, where the initial number of agents on each vertex is a poisson random variable with constant mean, and the information starts from the origin. They proved a theorem for the shape formed by the contour of the informed agents in the limiting case. In their analysis it is implicit that if the grid is finite, the spreading time is linear in the diameter (see also [19]). Our proof of Theorem 6 uses techniques from their analysis. A very similar process is the frog model, where only informed agents move, while uninformed ones stay at their initial position, until they are hit by an informed agent. At that point they get informed, and start their own walk. This process has been studied on infinite grids [3,30] and trees [20].

3 Upper Bound for Regular Graphs

In this section we prove Theorem 2.

3.1 Preliminaries

Let $G = (V, E)$ be any graph (not necessarily a regular one), and let A be the set of agents in VISIT-EXCHANGE, where $|A| = \alpha \cdot n$ for a constant $\alpha > 0$. The agents in A start their walks from the stationary distribution π. For a vertex u, let $N_u(t)$ be the number of agents

[3] The tilde asymptotic notation hides polylogarithmic factors.

that are at vertex u at round t. For an integer $r > 0$ and round t, let

$$\hat{N}_u(t,r) = \mathbb{E}\left[N_u(t+r) \mid N_v(t),\ \text{for all } v \in V\right] = \sum_{v \in V} p^r_{v,u} \cdot N_v(t),$$

where $p^r_{v,u}$ is the probability that a random walk starting from v is at u after exactly r rounds.

▶ **Lemma 7.** *For any vertex u, round t, and integer r,*

$$\mathbb{P}\left[\hat{N}_u(t,r) \le |A| \cdot \pi(u)/2\right] \le \exp\left(-\frac{|A| \cdot \pi(u)}{8 \cdot p^{2r}_{u,u}}\right).$$

Proof. Let $X^t_{v,g}$ be an indicator random variable, which is 1 when agent g is at vertex v at round t. Then, $N_v(t) = \sum_{g \in A} X^t_{v,g}$, which implies

$$\hat{N}_u(t,r) = \sum_{v \in V} p^r_{v,u} \sum_{g \in A} X^t_{v,g} = \sum_{g \in A} \sum_{v \in V} p^r_{v,u} \cdot X^t_{v,g} = \sum_{g \in A} Y_g,$$

where Y_g is the internal sum above for agent g. The random variables Y_g, $g \in A$, are independent, since the agents perform independent random walks. We compute the expectation and the second moment of Y_g to argue about the concentration of $\hat{N}_u(t,r)$.

$$\mathbb{E}\left[\hat{N}_u(t,r)\right] = \mathbb{E}\left[N_u(t+r)\right] = |A| \cdot \pi(u),$$

as the agents are initially distributed according to the stationary distribution π.

$$\begin{aligned}
\mathbb{E}\left[Y_g^2\right] &= \mathbb{E}\left[\sum_{v_1,v_2 \in V} p^r_{v_1,u} p^r_{v_2,u} \cdot X^t_{v_1,g} \cdot X^t_{v_2,g}\right] \\
&= \sum_{v \in V} \left(p^r_{v,u}\right)^2 \cdot \mathbb{E}\left[X^t_{v,g}\right], \quad \text{as } g \text{ cannot be in two vertices simultaneously}, \\
&= \sum_{v \in V} p^r_{v,u} \cdot \left(p^r_{v,u} \cdot \pi(v)\right), \quad \text{since } g \text{ is placed according to } \pi, \\
&= \sum_{v \in V} p^r_{u,v} \cdot \left(\pi(u) \cdot p^r_{u,v}\right), \quad \text{by reversibility}, \\
&= \pi(u) \cdot \sum_{v \in V} p^r_{u,v} \cdot p^r_{v,u} \\
&= \pi(u) \cdot p^{2r}_{u,u}.
\end{aligned}$$

We apply [6, Theorem 3.7], setting $\lambda = \mathbb{E}\left[\hat{N}_u(t,r)\right]/2$ and $M = 0$, to obtain

$$\begin{aligned}
\mathbb{P}\left[\hat{N}_u(t,r) \le |A| \cdot \pi(u)/2\right] &\le \exp\left(-\frac{\lambda^2}{2 \cdot \sum_{g \in A} \mathbb{E}\left[Y_g^2\right]}\right) \\
&\le \exp\left(-\frac{(|A| \cdot \pi(u))^2}{8 \cdot \sum_{g \in A} \pi(u) \cdot p^{2r}_{u,u}}\right) = \exp\left(-\frac{|A| \cdot \pi(u)}{8 \cdot p^{2r}_{u,u}}\right).
\end{aligned}$$ ◀

We will also need the following result, whose proof is in the full version of the paper [18].

▶ **Lemma 8.** *Let $X(t)$ be a simple random walk that starts at vertex u of a connected graph $G = (V,E)$. If $\deg(u)$ is the degree of u, and $d_{\min}$ is the smallest degree of G, then for any even $t \ge 0$, $\mathbb{P}\left[X(t) = u\right] \le \frac{\deg(u)}{|E|} + \frac{20 \cdot \deg(u)}{d_{\min} \cdot \sqrt{t+1}}$.*

3.2 Analysis

Suppose that $G = (V, E)$ is a d-regular graph with $d = O(\log n)$, thus $\pi(u) = 1/n$ for any $u \in V$. For a constant $\rho > 0$ define $r = r(\rho)$ as the smallest even integer such that

$$r \geq \max\{\rho \cdot \log^2 n,\ 256d \cdot \log n/\alpha\} = \Theta(\log^2 n). \tag{1}$$

We modify the VISIT-EXCHANGE process to create a new process called TWEAKED$_r$, as follows: At the end of each round $t \geq 0$, we add a minimal set of agents to the process to make sure that $\hat{N}_u(t, r) \geq |A| \cdot \pi(u)/2 = \alpha/2$, for every vertex u. Next we prove that, in the first polynomially many rounds TWEAKED$_r$ and VISIT-EXCHANGE are equivalent, w.h.p. Therefore, the results that we prove for TWEAKED$_r$, also hold for VISIT-EXCHANGE, w.h.p. This technique allows us to avoid dealing with dependencies of the random walks, which would arise if we directly analyzed VISIT-EXCHANGE conditioned on $\hat{N}_u(t, r) \geq \alpha/2$ for all u and t. (Similar tweaked processes are used in the proofs of Theorems 2 to 5 to circumvent some dependencies.)

▶ **Lemma 9.** *For any constant $c > 0$, there is a constant ρ such that VISIT-EXCHANGE and TWEAKED$_r$ are identical for the first T' rounds of their execution with probability at least $1 - T' \cdot n^{-(c+2)}$.*

Proof. By Lemma 8, $p^{2r}_{u,u} \leq \frac{2}{n} + \frac{20}{\sqrt{2r+1}} \leq \frac{20}{\sqrt{r}}$, since $r = O(\log^2 n)$. For $t < T'$, we substitute the above inequality into Lemma 7, and use the fact that $|A| \cdot \pi(u) = \alpha$, to get that

$$\mathbb{P}\left[\hat{N}_u(t, r) \leq \alpha/2\right] \leq \exp\left(-\frac{\alpha}{8 \cdot p^{2r}_{u,u}}\right) \leq \exp\left(-\frac{\alpha}{160} \cdot \sqrt{r}\right) \leq n^{-(c+3)},$$

for a sufficiently large constant ρ. By applying a union bound over all vertices u and rounds $t < T'$, we complete the proof. ◀

Consider two vertices u and v with distance $O(r/\max\{d, \log^2 \log n\})$, and assume u is informed at round t_0. The next key lemma provides a lower bound for the probability that v becomes informed $O(r)$ rounds after t_0. The lemma holds for any execution prefix of TWEAKED$_r$ up to round t_0, which means we can apply it repeatedly to prove Theorem 2. Let $\mathcal{K}_t$ be the σ-field that determines the execution of TWEAKED$_r$ until round t.

▶ **Lemma 10.** *Let $h = \max\{d, \log\log n\}$, and $k_{\max}(\gamma) = \frac{\gamma \cdot r}{\max\{d, (\log\log n)^2\}}$. There are constants $\gamma, \beta > 0$, such that for any round t_0 and any two vertices u, v with $\operatorname{dist}(u, v) \leq k_{max}(\gamma)$, given $\mathcal{K}_{t_0}$ and that u is informed at round t_0, vertex v is informed at round $t_0 + 2r$ with probability at least $1 - e^{-\beta \cdot h}$.*

Proof. Case $d = \omega(\log\log n)$. To simplify presentation, we assume $t_0 = 0$ and omit the conditional $\mathcal{K}_{t_0}$ throughout the proof. Fix the constant γ such that $k_{\max}(\gamma) \leq \frac{\alpha r}{256d}$. Consider two vertices u, v such that a shortest path between them is $u = u_0, \ldots, u_k = v$, where $k = \operatorname{dist}(u, v) \leq k_{\max}(\gamma)$. For a round $t \geq r$ and $i \in \{0, \ldots, k-1\}$, let $Z_{i,t}$ be the number of agents in the neighbourhood $\Gamma(u_i)$ of vertex u_i at round t. Then, by definition of TWEAKED$_r$,

$$\mathbb{E}\left[Z_{i,t}\right] = \sum_{w \in \Gamma(u_i)} \mathbb{E}\left[N_{u_i}(t)\right] = \sum_{w \in \Gamma(u_i)} \mathbb{E}\left[\hat{N}_{u_i}(t - r, r)\right] \geq \alpha \cdot d/2.$$

Since the agents make independent random walks, by a Chernoff bound we get that

$$\mathbb{P}\left[Z_{i,t} \geq \alpha \cdot d/4\right] \geq 1 - e^{-\alpha \cdot d/16}.$$

If $\mathcal{E}$ is the event that $Z_{i,t} \geq \alpha \cdot d/4$ for all $i \in \{0, \dots k-1\}$ and $t \in \{r, \dots 2r\}$ simultaneously, then, by a union bound,

$$\mathbb{P}[\mathcal{E}] \geq 1 - k \cdot r \cdot e^{-\alpha \cdot d/16} \geq 1 - e^{-\beta d}/2,$$

for a small enough constant β, because $kr = O(\text{poly}(\log n))$ and $d = \omega(\log \log n)$.

We modify TWEAKED$_r$ as follows: If $\mathcal{E}$ does not hold, then we add a minimum number of agents to the process so that $\mathcal{E}$ holds. We call the new process R-TWEAKED$_r$, and observe that TWEAKED$_r$ and R-TWEAKED$_r$ are identical with probability at least $1 - e^{-\beta d}/2$.

We divide the rounds $r, \dots, 2r-1$ of R-TWEAKED$_r$ into $r/2$ phases of 2 rounds each. For each $0 \leq i < r/2$, let $\mathcal{K}'_i$ be the σ-algebra which determines the execution prefix of R-TWEAKED$_r$ until round $r + 2i \leq 2r$. Let p_i be the largest integer, between 0 and k, such that vertex $w = u_{p_i}$ is informed at round $r + 2i$. If $p_i < k$, then each agent that is in the neighbourhood of w in round $r + 2i$, informs vertex u_{p_i+1} after two rounds, with probability $1/d^2$, by going through w. Define a Bernoulli random variable X_i, such that $X_i = 1$ if $p_i < k$ and u_{p_i+1} is informed in round $r + 2(i+1)$, i.e., the ith phase is successful. For technical convenience, we also define $X_i = 1$ if $p_i = k$, i.e., v is already informed in that phase. Then,

$$\mathbb{P}[X_i = 1 \mid \mathcal{K}'_i] \geq 1 - \left(1 - d^{-2}\right)^{\alpha \cdot d/4} \geq 1 - e^{-\alpha/(4d)} \geq \frac{\alpha}{8d}. \tag{2}$$

Define $Y = \sum_{i=0}^{r/2-1} Y_i$, where Y_i are independent Bernoulli random variables with success probability $\alpha/8d$. By our choice of γ and (1),

$$\mathbb{E}[Y] = \frac{\alpha r}{16d} \geq 8(k_{\max}(\gamma) + \log n) \geq 8(k + \log n),$$

and, by a Chernoff bound,

$$\mathbb{P}[Y \geq k] \geq \mathbb{P}[Y \geq \mathbb{E}[Y]/2] \geq 1 - e^{-\mathbb{E}[Y]/8} \geq 1 - 1/n \geq 1 - e^{-\beta d}/2,$$

since $d = O(\log n)$ and by choosing constant β smaller if necessary. On the other hand, for $X = \sum_{i=1}^{r/2-1} X_i$, (2) implies that X stochastically dominates Y, in particular,

$$\mathbb{P}[X \geq k] \geq \mathbb{P}[Y \geq k] \geq 1 - e^{-\beta d}/2.$$

Note, $X \geq k$ implies that v is informed in R-TWEAKED$_r$ at round $2r$. Since R-TWEAKED$_r$ and TWEAKED$_r$ are identical with probability $1 - e^{-\beta d}/2$, vertex v must be informed in TWEAKED$_r$ at round $2r$ with probability at least $1 - e^{-\beta d} = 1 - e^{-\beta h}$.

Case $d = O(\log \log n)$. As in the previous case, we assume $t_0 = 0$ and consider the spread of information along a shortest path from u to v, namely, $u = u_0, \dots, u_k = v$. Fix a round $t \geq r$ and some $i \in \{0, \dots, k-1\}$. Let $l = (\eta \log \log n)^2$ for some constant η that will be specified later. For an agent g define R_g as the number of times agent g visits u_i in rounds $t, \dots, t+l-1$. If $X_g(t')$ is the position of the agent g at round t', then $R_g = \sum_{t'=t}^{t+l-1} \mathbb{1}_{X_g(t')=u_i}$, so by Lemma 8,

$$\mathbb{E}[R_g \mid X_g > 0] = \sum_{t'=t}^{t+l-1} \mathbb{P}[X_g(t') = u_i \mid R_g > 0] \leq 1 + \sum_{t'=t}^{t+l-1} \left(\frac{1}{n} + \frac{20}{\sqrt{t - t' + 1}}\right) \leq 50 \cdot \sqrt{l}.$$

Let $Z_{i,t}$ be the number of unique agents that visit u_i in rounds $t, \dots, t+l-1$.

$$\mathbb{E}[Z_{i,t}] = \sum_{g \in A} \mathbb{P}[R_g > 0] = \sum_{g \in A} \frac{\mathbb{E}[R_g]}{\mathbb{E}[R_g \mid R_g > 0]}$$

$$\geq \frac{\sum_{g\in A}\mathbb{E}[R_g]}{50\cdot\sqrt{l}} = \frac{\sum_{t'=t}^{t+l-1}\mathbb{E}[N_{u_i}(t')]}{50\cdot\sqrt{l}} = \frac{\sum_{t'=t}^{t+l-1}\mathbb{E}\left[\hat{N}_{u_i}(t'-r,r)\right]}{50\cdot\sqrt{l}}$$
$$\geq \frac{l\cdot\alpha/2}{50\cdot\sqrt{l}} = \frac{\alpha\cdot\sqrt{l}}{100}.$$

Since the agents are performing independent random walks, then by a Chernoff bound,

$$\mathbb{P}\left[Z_{i,t}\geq\alpha\cdot\sqrt{l}/200\right]\geq 1-\exp\left(-\frac{\alpha\eta}{800}\cdot\log\log n\right)\geq 1-1/\log^5 n,$$

for a suitable choice of η. We now let $\mathcal{E}$ be the event $Z_{i,t}\geq\alpha\cdot\sqrt{l}/200$ for all $i\in\{0,\ldots,k-1\}$ and $t\in\{r,\ldots,2r\}$, simultaneously. As before, we create R-TWEAKED$_r$ by adding minimum number of agents to TWEAKED$_r$ to ensure that $\mathcal{E}$ holds. Since $rk=O(\log^4 n)$, by a union bound, there is a constant β such that $\mathbb{P}[\mathcal{E}]\geq 1-e^{-\beta h}/2$.

The rest of the proof follows the same line of logic as in the case of $d=\omega(\log\log n)$. The only difference is that instead of phases of 2 rounds, we consider phases of l rounds. $\mathcal{E}$ implies that after each phase R-TWEAKED$_r$ informs the next vertex on the path with a constant probability since $\sqrt{l}=\Omega(d)$. Therefore, as long as $k\leq\gamma\cdot r/l$ for a sufficiently small γ, vertex v becomes informed at round $2r$ of R-TWEAKED$_r$ w.h.p., which completes the proof. ◀

Proof of Theorem 2. First, we consider the TWEAKED$_r$ process for a constant ρ chosen by Lemma 9 such that TWEAKED$_r$ is identical to VISIT-EXCHANGE in the first n^2 rounds of its execution, with probability at least $1-n^{-2}$. Consider a shortest path $s=u_0,\ldots,u_m=u$ from source vertex s to vertex u. Let $k=k_{max}(\gamma)$ be the upper bound on the distance from Lemma 10, and as before $h=\max\{d,\log\log n\}$. We divide the execution of TWEAKED$_r$ into phases of $2r$ rounds each. If vertex u_i is informed at the end of a phase, then by Lemma 10, the vertex $u_{\min\{m,i+k\}}$ will be informed in the next phase of $2r$ rounds with probability at least $1-e^{-\beta h}$, independently from the past.

For some constant $\eta\in(0,1)$, let $l=\lceil m/k+\log n/h\rceil/(1-\eta)$. For $i\in\{1,\ldots,l\}$, let X_i be a Bernoulli random variable that is 0 if in the ith phase of TWEAKED$_r$ either k new vertices along the specified path become informed, or vertex u becomes informed, i.e., the phase is successful. For $X=\sum_{i=1}^{l}X_i$, if $X<l-\lceil m/k\rceil$ then vertex u is informed at the end of the lth phase, because at least $\lceil m/k\rceil$ phases were successful. By a stochastic dominance argument as in Lemma 10 we upper bound $\mathbb{P}[X<l-\lceil m/k\rceil]$.

Let $\{Y_i\}_{1\leq i\leq l}$ be a collection of independent Bernoulli random variables $\mathbb{P}[Y_i=1]=e^{-\beta h}$. By Lemma 10, $\mathbb{P}[X_i=1\mid X_1,\ldots,X_{i-1}]\leq\mathbb{P}[Y_i=1]$, and therefore, for $Y=\sum_{i=1}^{l}Y_i$,

$$\begin{aligned}\mathbb{P}[X>l-\lceil m/k\rceil]&\leq\mathbb{P}[Y>l-\lceil m/k\rceil]\leq\mathbb{P}[Y\geq l-\lceil m/k+\log n/h\rceil]\\&=\mathbb{P}[Y\geq\eta\cdot l]=\mathbb{P}\left[Y\geq\eta\cdot e^{\beta h}\cdot\mathbb{E}[Y]\right]\\&\leq\left(\eta\cdot e^{\beta h-1}\right)^{-\eta\cdot l}\leq n^{-3},\end{aligned}$$

by a Chernoff bound and by taking a value of η that is sufficiently close to 1. Thus, after $l\cdot 2r$ rounds of TWEAKED$_r$ vertex u is informed with probability $1-n^{-3}$. By a union bound over all vertices, and the fact that TWEAKED$_r$ and VISIT-EXCHANGE are identical in the first n^2 rounds we get that $T\leq l\cdot 2r$ w.h.p. Since $k=O(r/\max\{d,(\log\log n)^2\})$, and $m\leq\operatorname{diam}(G)$, and $h=\max\{d,\log\log n\}$, we finally get that, w.h.p.,

$$T=O\left(\max\{d,(\log\log n)^2\}\cdot\operatorname{diam}(G)+\frac{\log^3 n}{h}\right)=\tilde{O}\left(d\cdot\operatorname{diam}(G)+\frac{\log^3 n}{d}\right).$$ ◀

4 Upper Bound for Trees

In this section we prove the upper bound part of Theorem 5. Recall, $R_{b,h}$ is a rooted b-ary tree, where each vertex at distance less than h from the root has b children, and all leaves are at distance h from the root; thus h is the *height* of the tree. The total number of vertices is $n = (b^{h+1} - 1)/(b - 1)$. The set of children of vertex u is denoted C_u. The set of descendants of u is denoted D_u; precisely, D_u contains the vertices in the subtree rooted u, including u itself. The height of that subtree is denoted h_u. We define the set $B_{u,l} = \{v \in D_u \mid h_v = h_u - l\}$, which contains all descendants of v at distance l from u. Finally, $Z_u(t)$ denotes the set of agents at vertex u at round t, and $Z_S(t) = \bigcup_{u \in S} Z(t)$ is the set of agents in the set $S \subseteq V$ at that round.

4.1 The Lucky-Gambler Process

We define an auxiliary process, called LUCKY-GAMBLER, which will be used in the analysis. The process has three parameters: two integers $m, k > 0$, and a probability $p < 1/2$. Consider a path graph P_m of length m, with vertices 0 up to m. For every integer $s \geq 0$, at round s exactly k *gamblers* appear on vertex 1 and make a biased random walk: for $0 < i < m$, the probability of moving from vertex i to $(i+1)$ and $(i-1)$ is $p_{i,i+1} = p$ and $p_{i,i-1} = 1 - p = q$, respectively. When the gambler reaches vertex 0 or m, it stops, i.e., $p_{0,0} = p_{m,m} = 1$ (states $0, m$ are absorbing). We will write LUCKY-GAMBLER(m, p, k) to explicitly state the parameters of the process.

For a vertex v of $R_{b,h}$, where $h_v \geq m$, we are going to couple the movement of the agents in part of the subtree of v, with the gamblers in LUCKY-GAMBLER. Using the coupling and the next lemmas, we argue that v receives agents at a constant rate. By carefully selecting the agents that are coupled, we can claim that agents arrive at constant rate to every vertex v on a given path to the root, independently for each vertex.

▶ **Lemma 11.** *If $p = 1/(b+1)$ and $k \geq \epsilon \cdot b^{m-1}$, for some constant $\epsilon > 0$, then there is a constant $\beta < 1$ such that for any round $t \geq 4m$ and positive integer Δ the probability that no gambler reaches vertex m during any round in $\gamma_0 = \{t, \ldots, t + \Delta - 1\}$ is at most $(1 - \beta)^{\Delta}$.*

▶ **Lemma 12.** *If $p = 1/(b+1)$ and $k \geq \kappa \cdot b^{m-1}$, for some integer κ, then there is a constant γ, such that for any integer $\tau \geq 8m$, at least $\gamma\kappa\tau$ gamblers reach vertex m in the first τ rounds, with probability at least $1 - e^{-\gamma\kappa\tau/4}$.*

We will use the next two results for a single gambler g making a biased random walk on P_m starting at round 0. Let $X_g(t)$ be the position of gambler g at round t and let $\tau_g(i) = \min\{t \mid X_g(t) = i\}$ be the hitting time of vertex i of g. We denote the event that $\tau_g(m) < \tau_g(0)$ as $\mathcal{L}_g$, and we will say that g is *lucky* if it occurs.

▶ **Lemma 13** ([15, Chapter 14]). *If $p \neq q$, then for $0 < i < m$, $\mathbb{P}[\mathcal{L}_g \mid X_g(0) = i] = \frac{(q/p)^i - 1}{(q/p)^m - 1}$.*

▶ **Lemma 14.** *If $p < q$, then for $0 < i < m$, $\mathbb{E}[\tau_g(m) \mid \mathcal{L}_g, X_g(0) = i] \leq \frac{m-i}{q-p}$.*

Below we prove Lemma 11; the proofs of Lemmas 12 and 14 can be found in the full version of the paper [18].

Proof of Lemma 11. For $s \geq 0$ and $1 \leq i \leq k$, let $g_{s,i}$ be the ith gambler that starts its walk at round s at vertex 1. Let $\tau_{s,i} = \tau_{g_{s,i}}$ be defined as for the single gambler g above. Clearly, $\tau_{s,i}(j) - s$ and $\tau_g(j)$ are identically distributed, if $X_g(0) = 1$. We also extend the definition of γ_0, letting $\gamma_s = \{t - s, \ldots, t + \Delta - s - 1\}$.

We would like to study the number of lucky gamblers that reach m at rounds in γ_0. Consider first a "toy" example, which assumes that for each s, exactly one gambler is lucky among the k gamblers that start their walk at round s. Suppose that g'_s is that lucky gambler. We study the expected number of these agents that reach m during the rounds in γ_0:

$$\mathbb{E}\left[\sum_{s\geq 0}\mathbf{1}_{\{\tau_{g'_s}(m)\in\gamma_0\}} \mid \mathcal{L}_{g'_s} \text{ for } s\geq 0\right] = \sum_{s=0}^{t+\Delta}\mathbb{P}\left[\tau_{g'_s}(m)\in\gamma_0 \mid \mathcal{L}_{g'_s}\right] = \sum_{s=0}^{t+\Delta}\mathbb{P}\left[\tau_g(m)\in\gamma_s \mid \mathcal{L}_g\right].$$

The setup in the "toy" example is unlikely to occur, however, we use it as a motivation to lower bound the last quantity, which will be used in the main part of the proof.

$$\begin{aligned}
\sum_{s=0}^{t+\Delta}\mathbb{P}\left[\tau_g(m)\in\gamma_s \mid \mathcal{L}_g\right] &= \sum_{l=0}^{\Delta-1}\sum_{\substack{0\leq s\leq t+\Delta\\ s\equiv l \pmod{\Delta}}}\mathbb{P}\left[\tau_g(m)\in\gamma_s \mid \mathcal{L}_g\right], \\
&\qquad \text{the inner sum is over every } \Delta\text{th summand,} \\
&\geq \sum_{l=0}^{\Delta-1}\mathbb{P}\left[\tau_g(m)<t \mid \mathcal{L}_g\right], \quad \text{by union of disjoint events,} \\
&= \Delta\cdot\mathbb{P}\left[\tau_g(m)<t \mid \mathcal{L}_g\right] \\
&\geq \Delta\cdot\left(1-\frac{\mathbb{E}\left[\tau_g(m) \mid \mathcal{L}_g\right]}{t}\right), \quad \text{by Markov's inequality,} \\
&\geq \Delta\cdot\left(1-\frac{m\cdot(b+1)}{t\cdot(b-1)}\right), \quad \text{by Lemma 14 as } q-p=\frac{b-1}{b+1}, \\
&\geq \Delta\cdot\left(1-\frac{b+1}{4(b-1)}\right), \quad \text{since } t\geq 4m, \\
&\geq \Delta/4.
\end{aligned}$$

We can now bound the probability that no agent visits vertex m between rounds t and $t+\Delta$:

$$\begin{aligned}
\mathbb{P}\left[\bigcap_{\substack{0\leq s\leq t+\Delta\\ 1\leq i\leq k}}\{\tau_{s,i}(m)\notin\gamma_0\}\right] &= \prod_{s=0}^{t+\Delta}\left(\mathbb{P}\left[\tau_{s,i}(m)\notin\gamma_0\right]\right)^k, \quad \text{by independence of the walks,} \\
&= \prod_{s=0}^{t+\Delta}\left(\mathbb{P}\left[\tau_g(m)\notin\gamma_s\right]\right)^k \\
&= \prod_{s=0}^{t+\Delta}\left(1-\mathbb{P}\left[\mathcal{L}_g\right]\cdot\mathbb{P}\left[\tau_g(m)\in\gamma_s \mid \mathcal{L}_g\right]\right)^k \\
&= \prod_{s=0}^{t+\Delta}\left(1-\frac{b-1}{b^m-1}\cdot\mathbb{P}\left[\tau_g(m)\in\gamma_s \mid \mathcal{L}_g\right]\right)^k, \text{ by Lemma 13,} \\
&\leq \prod_{s=0}^{t+\Delta}\exp\left(-\frac{k\cdot(b-1)}{b^m-1}\cdot\mathbb{P}\left[\tau_g(m)\in\gamma_s \mid \mathcal{L}_g\right]\right) \\
&\leq \exp\left(-\epsilon\cdot\frac{b^{m-1}(b-1)}{b^m-1}\cdot\sum_{s=0}^{t+\Delta}\mathbb{P}\left[\tau_g(m)\in\gamma_s \mid \mathcal{L}_g\right]\right) \\
&\leq \exp\left(-\frac{\epsilon\Delta}{8}\right), \quad \text{by the analysis of the toy example.} \quad \blacktriangleleft
\end{aligned}$$

4.2 Analysis

We define another auxiliary process, called TWEAKED, which is a slight modification of the original VISIT-EXCHANGE process. Let m be the *smallest* integer such that $b^m \geq \mu \cdot \ln n$ for a constant μ to be defined later, and let $k = \lceil \alpha \cdot b^m/8 \rceil$. Consider a vertex u of the tree, such that $h_u \geq m$, and recall that $B_{u,m}$ is the set of descendants of u at distance m. Let v be one of the children of u and define $Z'_{u,v}(t)$ be the set of agents that are in $B_{u,m-1}$ at round t and were in $B_{u,m} \setminus B_{v,m-1}$ the round before, i.e., $Z'_{u,v}(t) = Z_{B_{u,m-1}}(t) \cap Z_{B_{u,m} \setminus B_{v,m-1}}(t-1)$. For a round $t \geq 0$ let $q_{u,v}(t)$ be the smallest non-negative integer for which

$$|Z'_{u,v}(t)| + q_{u,v}(t) \geq \left\lceil \frac{\alpha}{8} \cdot |B_{u,m}| \right\rceil = \left\lceil \frac{\alpha}{8} \cdot b^m \right\rceil = k.$$

To construct TWEAKED we add exactly $q_{u,v}(t)$ agents in $B_{u,m-1}$ at round t (it is not important to which vertices in $B_{u,m-1}$ these agents are added).

To motivate the construction of TWEAKED, consider a vertex u and its child v, such that $m \leq h_u < h$. In round t of TWEAKED, there are at least k agents at vertices in $B_{u,m} \setminus B_{v,m-1}$ (of height $h_u - m$) that move closer to u in the next round. This allows us to couple these agents to that of gamblers in a LUCKY-GAMBLER$(m+1, 1/(b+1), k)$ process, and use our results from Sect. 4.1 to show that agents arrive at the parent of u at a constant rate. A key insight is that by not considering agents that are in descendants of v, the same argument can be made for vertex v, independently of u, if $h_v \geq m$ too. By repeating this argument, we show that in $O(\log n)$ rounds all vertices of height at least m are informed once one such vertex is informed. TWEAKED and LUCKY-GAMBLER are also used to analyse the spread of the message in the vertices of height at most m.

Using a Chernoff bound we can show that TWEAKED and VISIT-EXCHANGE are equivalent in the first polynomially many rounds, w.h.p.

▶ **Lemma 15.** *The probability that no agent is added in the* TWEAKED *process in the first r rounds is at least $1 - r \cdot n^{-\frac{\alpha \cdot \mu}{32}+1}$.*

We will use the same notation for TWEAKED and VISIT-EXCHANGE processes.

▶ **Lemma 16.** *Let u be any vertex of the tree $R_{b,h}$ such that $h_u \geq m$. For any constant $c > 0$, if u is informed, then after $O(\log n)$ rounds of* TWEAKED *the root ρ of $R_{b,h}$ gets informed, with probability at least $1 - n^{-c}$.*

Proof. Consider the path $u = u_1, \ldots, u_l = \rho$ from u to the root of the tree. Due to the symmetry of the tree, we can assume that the path is the "leftmost" path of the tree, i.e., for any $i \geq 1$, u_{i-1} is the leftmost child of u_i (for consistency, we let u_0 be the leftmost child of u_1). Roughly speaking, we show that for any i, the number of rounds between two consecutive visits to u_i (by a certain subset of agent) follows a geometric distribution, independently of the other $u_{i'}$. To that end, we couple the movement of agents of TWEAKED to $l-1$ independent instances of process LUCKY-GAMBLER$(m+1, 1/(b+1), k)$, one corresponding to each of the vertices u_i for $1 \leq i < l$.

Next we give some definitions and describe the coupling for a fixed i. For simplicity, define $B_i = B_{u_i,m}$ and $B'_i = B_i \setminus B_{u_{i-1},m-1} = \bigcup_{v \in C_{u_i} \setminus \{u_{i-1}\}} B_{v,m-1}$. I.e., B_i is the set of descendants of u_i at distance m from it, and to get B'_i we remove the descendants of u_{i-1} from B_i. Let $g_1, \ldots, g_{z_{i,t}}$ be the agents in TWEAKED that were at B'_i in round $t-1$ and moved closer to the root in the next round. By definition of TWEAKED, there are at least $k = \lceil \alpha \cdot b^m/8 \rceil$ such agents.

In the LUCKY-GAMBLER$(m+1, 1/(b+1), k)$ process that corresponds to vertex u_i, we start k gamblers in round t, denoted $g'_1, \ldots, g'_k$. For each $1 \le j \le k$, and for each round $t' \ge t$ until g_j reaches u_{i+1} or any vertex in B_i, the walks g_j and g'_j are coupled: if g_j moves closer to the root then g'_j moves to the right on the path, and if g_j moves away from the root, g'_j moves left. If g_j is at u_{i+1} or in $B_{u_i,m}$, then by the coupling, g'_j has finished its walk at one of the endpoints of the path. Before this happens we say that g_j is *i-coupled*.

Let $t_1 = 4\cdot(m+1)$, and let t_{i+1} be the first round after t_i when u_{i+1} receives an i-coupled agent from u_i. Now, notice that by construction no agent can be i-coupled and i'-coupled at the same time for $i' \ne i$. It implies that the rounds when u_{i+1} receives i-coupled agents are independent from the walks of i'-coupled agents. On the other hand the walks of i-coupled agents are coupled with an independent LUCKY-GAMBLER process thus, Lemma 11 implies

$$\mathbb{P}\left[t_{i+1} - t_i \le s \mid t_1, \ldots, t_i\right] = (1-\beta)^s = \mathbb{P}\left[F_i \ge s\right],$$

where $F_i \sim Geom(\beta)$, $1 \le i < l$, are a collection of independent geometric random variables with success probability β. If τ_ρ is the round when the root is informed then $\tau_\rho \le t_l = t_1 + \sum_{i=1}^{l-1}(t_{i+1} - t_i)$. It follows that $(\tau_\rho - t_1)$ is stochastically dominated by $F = \sum_{i=1}^{l-1} F_i$, and from a Chernoff bound for the sum of independent geometric random variables,

$$\mathbb{P}\left[\tau_\rho \ge f + t_1\right] \le \mathbb{P}\left[F \ge f\right] \le e^{-f\cdot\beta/8},$$

for any $f \ge 2h/\beta$. Since $t_1 = O(h)$, we can take a large enough $f = O(\log n)$, completing the proof. ◀

Next we prove that if vertex u of height $h_u = m$ is informed, then after at most $O(m \ln n)$ rounds a given leaf v in u's subtree becomes informed, w.h.p. For that, we first show that there are at least $\Theta(m \ln n)$ visits to u in those rounds (possibly multiple times by the same agent). Using a lower bound on the probability that an agent that is at u visits v before returning to u, we can show that one of these agents will visit v in $O(m \ln n)$ rounds, w.h.p.

▶ **Lemma 17.** *Let u be such that $h_u = m$. For any constant $c > 0$, there is a round $\tau = O(m \ln n)$ such that in the first τ rounds of* TWEAKED*, u is visited at least $c \cdot mb \cdot \ln n$ times, with probability at least $1 - n^{-cmb}$.*

Proof. For a round t, let $g_1, \ldots, g_{z_{u,t}}$ be the agents that are in $B_{u,m-1}$ at round t, and have also been at the leaf vertices $B_{u,m}$ in the previous round. By the definition of TWEAKED, $z_{u,t} \ge k$, where $k = \lceil \alpha b^m/8 \rceil$. We construct an instance of LUCKY-GAMBLER$(m, 1/(b+1), k)$ as follows. If $g'_1, \ldots, g'_k$ are the gamblers that started their walk at round t, then for each $1 \le j \le k$, the walk of agent g_j is coupled with the walk of the gambler g'_k: If g_j moves closer to the root of the tree, then g'_j moves right on the path and left otherwise. The coupling ends when g'_j arrives at either vertex 0 or m of its path. That corresponds to g_j either visiting a leaf vertex in $B_{u,m}$ or visiting vertex u.

Consider the first τ rounds of TWEAKED. Since $k \ge \alpha b^m/8$, we can apply Lemma 12 with parameter $\kappa = \alpha b/8$ to the coupled LUCKY-GAMBLER process. Let γ be the constant guaranteed by the lemma and let $\tau = \frac{8c}{\alpha\gamma} \cdot m \ln n$. Lemma 12 implies that in the first τ rounds of LUCKY-GAMBLER there are at least $\gamma\kappa\tau = c \cdot mb \cdot \ln n$ lucky gamblers, with probability at least $1 - e^{-\gamma\kappa\tau/4} = 1 - e^{-cmb \ln n} = 1 - n^{-cmb}$. Since each lucky gambler corresponds to a single visit to u by some agent, we complete the proof. ◀

▶ **Lemma 18.** *Let u be such that $h_u = m$ and let v be a leaf in the subtree of u. For any constant $c_l > 0$, if vertex u is informed then after at most $O(m \ln n)$ rounds of* TWEAKED*, vertex v is informed with probability at least $1 - n^{-c_l}$.*

Proof. Let τ be the round guaranteed by Lemma 17 for a constant $c > 0$. If after the first τ rounds of TWEAKED, there have been fewer than $cmb\ln n$ visits to u, then we add a minimal number of agents to u at round τ to have at least $cmb\ln n$ agents there. We call the resulting process TWEAKED$_u$. By Lemma 17 and an application of union bound over the first $\log^2 n = \omega(m\ln n)$ rounds, TWEAKED$_u$ and TWEAKED are identical in the first $\Theta(m\ln n)$ rounds of execution with probability at least $1-n^{-cmb}\log^2 n$. We therefore analyse TWEAKED$_u$.

For a round $t \le \tau$, consider an agent g that visits u at round t. Let $\mathcal{D}_{g,t}$ be the event that g moves to one of u's children at round $t+1$. Let also $\mathcal{E}_{g,t}$ be the event that g visits v before returning to u, and before round $\tau' = \tau + 8mb^{m-1}$. Clearly, $\mathcal{E}_{g,t}$ implies $\mathcal{D}_{g,t}$, and $\mathbb{P}[\mathcal{D}_{g,t}] = \frac{b}{b+1}$. Also, we can show that $\mathbb{P}[\mathcal{E}_{g,t} \mid \mathcal{D}_{g,t}] \ge 1/(12mb)$, by analysing a single random walk in $R_{b,m}$ that starts in the root of the tree [18]. Therefore,

$$\mathbb{P}[\mathcal{E}_{g,t}] = \mathbb{P}[\mathcal{E}_{g,t} \cap \mathcal{D}_{g,t}] = \mathbb{P}[\mathcal{D}_{g,t}] \cdot \mathbb{P}[\mathcal{E}_{g,t} \mid \mathcal{D}_{g,t}] \ge \frac{b}{b+1} \cdot \frac{1}{12mb} \ge \frac{1}{18mb}.$$

The probability that v is not visited by any informed agent before round τ' is at most

$$\mathbb{P}\left[\bigcap_{t\le\tau,\, g\in Z_u(t)} \neg\mathcal{E}_{g,t}\right] \le \left(1-\frac{1}{18mb}\right)^{cmb\ln n} \le e^{-c\ln n/18} \le n^{-c/18} \le n^{-c_l-1},$$

for a large enough constant c. Notice that $\tau' = \tau + 8mb^{m-1} = O(m\ln n)$ by the definition of m. Since TWEAKED and TWEAKED$_u$ are identical in the first $\log^2 n$ rounds with probability at least $1-n^{-cmb}\log^2 n$, v will be informed in $O(m\ln n)$ rounds in TWEAKED, with probability at least $1-n^{-c_l-1}-n^{-cmb}\log^2 n \ge 1-n^{-c_l}$. ◀

Proof of the Upper Bound of Theorem 5. We will use the following simple symmetry lemma, which holds for any graph: If $T_{u,v}$ is the number of rounds of VISIT-EXCHANGE until vertex v is informed when the information originates at u, then the random variables $T_{u,v}$ and $T_{v,u}$ have the same distribution [18].

Consider the TWEAKED process, and suppose that the source of the information is vertex u with $h_u = m$, for m as defined at the beginning of Sect. 4.2. By Lemma 16, for an arbitrary constant c, there is $T_1 = O(\log n)$ such that the root ρ is informed by time T_1, with probability at least $1-n^{-c}$. Lemma 15 then implies that the same bound T_1 holds for the VISIT-EXCHANGE process, with probability $p \ge 1-n^{-c}-n^{-\alpha\mu/32}$, for an arbitrary large μ. From the symmetry lemma above, it follows that if ρ is the initial source of the information instead, then u becomes informed within T_1 rounds of VISIT-EXCHANGE with the same probability $p \ge 1-n^{-c}-n^{-\alpha\mu/32}$.

Suppose again that information originates at some u with $h_u = m$, and let v be any leaf that is a descendant of u. From Lemma 18 and Lemma 15, for an arbitrary constant c, there is some $T_2 = O(m\log n)$, such that v gets informed after at most T_2 rounds of VISIT-EXCHANGE, with probability at least $1-n^{-c}-n^{-\alpha\mu/32}$.

Combining the above we obtain that if ρ is the source of the information, then any given leaf v is informed after at most T_1+T_2 rounds of VISIT-EXCHANGE, with probability at least $1-2n^{-c}-2n^{-\alpha\mu/32}$. And by a union bound, all leaves (and thus all vertices) are informed within T_1+T_2 rounds with probability at least $1-2n^{-c+1}-2n^{-\alpha\mu/32+1}$.

Finally, by employing the symmetry argument above again, we obtain that for any source vertex (not just ρ), all vertices are informed within $2(T_1+T_2)$ rounds with probability at least $1-4n^{-c+1}-4n^{-\alpha\mu/32+1}$. Since $T_1+T_2 = O(\log n + m\log n) = O(\log n + \log_b \log n \cdot \log n) = O(\log n + h\log h)$, the theorem follows. ◀

References

1. Romas Aleliunas, Richard M. Karp, Richard J. Lipton, László Lovász, and Charles Rackoff. Random walks, universal traversal sequences, and the complexity of maze problems. In *Proc. 20th IEEE Symposium on Foundations of Computer Science, FOCS*, pages 218–223, 1979. doi:10.1109/SFCS.1979.34.
2. Noga Alon, Chen Avin, Michal Koucký, Gady Kozma, Zvi Lotker, and Mark R. Tuttle. Many random walks are faster than one. *Combinatorics, Probability & Computing*, 20(4):481–502, 2011. doi:10.1017/S0963548311000125.
3. O. S. M. Alves, F. P. Machado, and S. Yu. Popov. The shape theorem for the frog model. *The Annals of Applied Probability*, 12(2):533–546, May 2002. doi:10.1214/aoap/1026915614.
4. Andrei Z. Broder, Anna R. Karlin, Prabhakar Raghavan, and Eli Upfal. Trading space for time in undirected s-t connectivity. *SIAM J. Comput.*, 23(2):324–334, 1994. doi:10.1137/S0097539790190144.
5. Flavio Chierichetti, George Giakkoupis, Silvio Lattanzi, and Alessandro Panconesi. Rumor spreading and conductance. *J. ACM*, 65(4):17:1–17:21, 2018. doi:10.1145/3173043.
6. Fan R. K. Chung and Lincoln Lu. Concentration inequalities and martingale inequalities: A survey. *Internet Mathematics*, 3(1):79–127, 2006. doi:10.1080/15427951.2006.10129115.
7. Colin Cooper. Random walks, interacting particles, dynamic networks: Randomness can be helpful. In *Proc. 18th International Colloquium on Structural Information and Communication Complexity, SIROCCO*, pages 1–14, 2011. doi:10.1007/978-3-642-22212-2_1.
8. Colin Cooper, Alan M. Frieze, and Tomasz Radzik. Multiple random walks in random regular graphs. *SIAM J. Discrete Math.*, 23(4):1738–1761, 2009. doi:10.1137/080729542.
9. Alan J. Demers, Daniel H. Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard E. Sturgis, Daniel C. Swinehart, and Douglas B. Terry. Epidemic algorithms for replicated database maintenance. *Operating Systems Review*, 22(1):8–32, 1988. doi:10.1145/43921.43922.
10. Tassos Dimitriou, Sotiris E. Nikoletseas, and Paul G. Spirakis. The infection time of graphs. *Discrete Applied Mathematics*, 154(18):2577–2589, 2006. doi:10.1016/j.dam.2006.04.026.
11. Benjamin Doerr, Mahmoud Fouz, and Tobias Friedrich. Social networks spread rumors in sublogarithmic time. In *Proc. 43rd ACM Symposium on Theory of Computing, STOC*, pages 21–30, 2011. doi:10.1145/1993636.1993640.
12. Klim Efremenko and Omer Reingold. How well do random walks parallelize? In *Proc. Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM*, pages 476–489, 2009. doi:10.1007/978-3-642-03685-9_36.
13. Robert Elsässer and Thomas Sauerwald. Tight bounds for the cover time of multiple random walks. *Theor. Comput. Sci.*, 412(24):2623–2641, 2011. doi:10.1016/j.tcs.2010.08.010.
14. Uriel Feige, David Peleg, Prabhakar Raghavan, and Eli Upfal. Randomized broadcast in networks. *Random Struct. Algorithms*, 1(4):447–460, 1990. doi:10.1002/rsa.3240010406.
15. William Feller. *An Introduction to Probability Theory and its Applications. Vol. 1.* Wiley series in probability and mathematical statistics. Wiley, 1968.
16. George Giakkoupis, Frederik Mallmann-Trenn, and Hayk Saribekyan. How to spread a rumor: Call your neighbors or take a walk? In *Proc. 38th ACM Symposium on Principles of Distributed Computing, PODC*, pages 24–33, 2019. doi:10.1145/3293611.3331622.
17. George Giakkoupis, Frederik Mallmann-Trenn, and Hayk Saribekyan. How to spread a rumor: Call your neighbors or take a walk? *CoRR*, abs/2006.02368, 2020. arXiv:2006.02368.
18. George Giakkoupis, Hayk Saribekyan, and Thomas Sauerwald. Spread of information and diseases via random walks in sparse graphs. Research report, Inria, August 2020. URL: https://hal.inria.fr/hal-02913942.
19. Peter Gracar and Alexandre Stauffer. Percolation of lipschitz surface and tight bounds on the spread of information among mobile agents. In *Proc. Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM*, volume 116 of *LIPIcs*, pages 39:1–39:17, 2018. doi:10.4230/LIPIcs.APPROX-RANDOM.2018.39.

20 Jonathan Hermon. Frogs on trees? *Electron. J. Probab.*, 23:40 pp., 2018. doi:10.1214/18-EJP144.

21 Richard M. Karp, Christian Schindelhauer, Scott Shenker, and Berthold Vöcking. Randomized rumor spreading. In *Proc. 41st IEEE Symposium on Foundations of Computer Science, FOCS*, pages 565–574, 2000. doi:10.1109/SFCS.2000.892324.

22 Harry Kesten and Vladas Sidoravicius. Branching random walk with catalysts. *Electronic Journal of Probability*, 8(0), 2003. doi:10.1214/EJP.v8-127.

23 Harry Kesten and Vladas Sidoravicius. The spread of a rumor or infection in a moving population. *The Annals of Probability*, 33(6):2402–2462, November 2005. Zbl: 1111.60074. doi:10.1214/009117905000000413.

24 Harry Kesten and Vladas Sidoravicius. A shape theorem for the spread of an infection. *Annals of Mathematics*, 167(3):701–766, May 2008. doi:10.4007/annals.2008.167.701.

25 Axel Kramer, Ingeborg Schwebke, and Günter Kampf. How long do nosocomial pathogens persist on inanimate surfaces? A systematic review. *BMC Infectious Diseases*, 6(1):130, December 2006. doi:10.1186/1471-2334-6-130.

26 Henry Lam, Zhenming Liu, Michael Mitzenmacher, Xiaorui Sun, and Yajun Wang. Information dissemination via random walks in d-dimensional space. In *Proc. 23rd ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 1612–1622, 2012. doi:10.1137/1.9781611973099.128.

27 Russell Lyons and Shayan Oveis Gharan. Sharp bounds on random walk eigenvalues via spectral embedding. *International Mathematics Research Notices*, 2018(24):7555–7605, May 2017. doi:10.1093/imrn/rnx082.

28 Alberto Pettarin, Andrea Pietracaprina, Geppino Pucci, and Eli Upfal. Infectious random walks. *CoRR*, abs/1007.1604:21 pp., 2010. arXiv:1007.1604.

29 Alberto Pettarin, Andrea Pietracaprina, Geppino Pucci, and Eli Upfal. Tight bounds on information dissemination in sparse mobile networks. In *Proc. 30th ACM Symposium on Principles of Distributed Computing, PODC*, pages 355–362, 2011. doi:10.1145/1993806.1993882.

30 Serguei Yu. Popov. Frogs and some other interacting random walks models. In *Proc. Discrete Random Walks, DRW*, pages 277–288, 2003. URL: http://dmtcs.episciences.org/3328.

Spiking Neural Networks Through the Lens of Streaming Algorithms

Yael Hitron
Weizmann Institute of Science, Rehovot, Israel
yael.hitron@weizmann.ac.il

Cameron Musco
University of Massachusetts, Amherst, MA, USA
cmusco@cs.umass.edu

Merav Parter
Weizmann Institute of Science, Rehovot, Israel
merav.parter@weizmann.ac.il

Abstract

We initiate the study of biologically-inspired *spiking neural networks* from the perspective of streaming algorithms. Like computers, human brains face memory limitations, which pose a significant obstacle when processing large scale and dynamically changing data. In computer science, these challenges are captured by the well-known streaming model, which can be traced back to Munro and Paterson '78 and has had significant impact in theory and beyond. In the classical streaming setting, one must compute a function f of a stream of updates $\mathcal{S} = \{u_1, \ldots, u_m\}$, given restricted single-pass access to the stream. The primary complexity measure is the space used by the algorithm.

In contrast to the large body of work on streaming algorithms, relatively little is known about the computational aspects of data processing in spiking neural networks. In this work, we seek to connect these two models, leveraging techniques developed for streaming algorithms to better understand neural computation. Our primary goal is to design networks for various computational tasks using as few auxiliary (non-input or output) neurons as possible. The number of auxiliary neurons can be thought of as the "space" required by the network.

Previous algorithmic work in spiking neural networks has many similarities with streaming algorithms. However, the connection between these two space-limited models has not been formally addressed. We take the first steps towards understanding this connection. On the upper bound side, we design neural algorithms based on known streaming algorithms for fundamental tasks, including distinct elements, approximate median, and heavy hitters. The number of neurons in our solutions almost match the space bounds of the corresponding streaming algorithms. As a general algorithmic primitive, we show how to implement the important streaming technique of linear sketching efficiently in spiking neural networks. On the lower bound side, we give a generic reduction, showing that any space-efficient spiking neural network can be simulated by a space-efficient streaming algorithm. This reduction lets us translate streaming-space lower bounds into nearly matching neural-space lower bounds, establishing a close connection between the two models.

2012 ACM Subject Classification Networks → Network algorithms

Keywords and phrases Biological distributed algorithms, Spiking neural networks, Streaming algorithms

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.10

Related Version https://arxiv.org/abs/2010.01423

Funding Supported by the BSF-NSF-Computer Science grant no. 713043.

Acknowledgements We are very grateful to Eylon Yogev for various discussions on pseudorandom generators, and for pointing out to us Lemma 27.

© Yael Hitron, Cameron Musco, and Merav Parter;
licensed under Creative Commons License CC-BY
34th International Symposium on Distributed Computing (DISC 2020).
Editor: Hagit Attiya; Article No. 10; pp. 10:1–10:18
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

In this work, we seek to understand the role of *memory constraints* in neural data processing. We consider data-stream tasks, in which a long stream of inputs is presented over time and a neural network must evaluate some function f of this stream. Examples include identifying frequent input patterns (items) or estimating summary statistics, such as the number of distinct items presented. The network cannot store the full stream and so must maintain some form of compressed representation in its working memory, which allows the eventual computation of f. The primary objective is to compute f with as few auxiliary (non-input or output) neurons as possible. The number of auxiliary neurons can be thought of as the "space" required by the network.

In computer science, data processing under space limitations is extensively studied in the area of streaming algorithms [37, 38]. We leverage this body of work to further our understanding of space-efficient neural networks. We start by designing neural networks for a large class of data-stream tasks, building off fundamental streaming algorithms and techniques, such as linear sketching. We also establish general connections between these models, showing that streaming-space lower bounds can be translated to neural-space lower bounds. We hope that these connections are a first step in extending work on streaming computation to better understand neural processing of massive and dynamically changing data under memory constraints.

The spiking neural network (SNN) model [32, 33]. A spiking network is represented by a directed weighted graph over n input neurons, r output neurons, and s auxiliary neurons. The edges of the graph represent synapses of different strengths connecting the neurons. The network evolves in discrete, synchronous rounds as a Markov chain where each neuron u acts as a (possibly probabilistic) threshold gate that either fires (spikes) or is silent in each round. In round t, the firing status of u depends on the firing status of its incoming neighbors in the preceding round $t-1$, and the strength of the connections from these neighbors. In *randomized* SNNs, there are two sources of randomness: the spiking behavior of the neurons and the selection of random edge weights in the network. In *deterministic* SNNs, the neurons are deterministic threshold gates and the edge weights are deterministically chosen. Aside from their relevance in modeling biological computation, SNNs have received significant attention as more energy efficient alternatives to traditional artificial neural networks [25, 42].

A recent series of works in the emerging area of *algorithmic SNNs* [33, 34, 11, 29, 28, 43, 8, 26, 41, 35, 40, 16] focuses on network design tasks. In this framework, given a target function $f : \{0,1\}^n \to \{0,1\}^r$, one seeks to design a space-efficient SNN (with few auxiliary neurons) that converges rapidly to an output spiking pattern matching $f(x)$ when the input spiking pattern matches x. Space-efficient SNNs have been devised for the winner-takes-all problem [28, 41], similarity testing and compression [31, 40], clustering [14, 26], approximate counting, and time estimation [30, 15]. Interestingly, many of these works borrow ideas from related streaming algorithms. However, despite the flow of ideas from streaming to neural algorithms, the connection between these models has not been studied formally.

The streaming model [37, 38]. A data-stream is a sequence of updates $\mathcal{S} = \{u_1, \ldots, u_m\}$. A streaming algorithm $\mathcal{A}$ computes some function of $\mathcal{S}$, given restricted access to the stream. In the standard single-pass model, the algorithm can only read the updates in $\mathcal{S}$ once, in the order they are presented.

Most commonly, and throughout this work, each update u_i represents the insertion or deletion of an item x_i belonging to a universe U with $|U| = n$. Without loss of generality, we will always consider U to be the set of integers $[n] = 1, \ldots, n$, and f is a function of the frequency vector $\bar{z} \in \mathbb{R}^n$, which tracks the total frequency of each item in the stream (the number of insertions minus the number of deletions). In the *insertion-only* setting, only insertions are allowed – i.e., each update increments some entry of $\bar{z}$. In the general *turnstile* (dynamic) setting, there are both insertions and deletions – i.e., increments and decrements to entries in $\bar{z}$. The primary complexity measure of a streaming algorithm is the *space* (measured in number of bits) required to maintain the evaluation of f on the data-stream.

Neural networks from a streaming perspective. Our primary goal is to devise space-efficient spiking neural networks that solve natural data-stream tasks, which mirror data processing tasks solved in real biological networks. In light of the large collection of space-efficient streaming algorithms that have been designed for various problems, we start by asking:

▶ **Question 1.** Is it possible to translate a space-efficient streaming algorithm for a given task into a space-efficient SNN algorithm for that task? Do generic reductions from SNNs to streaming exist?

The streaming literature is also rich with space lower bounds. For many classical data-stream problems, these lower bounds are nearly tight. To obtain space lower bounds for SNNs, we ask if reductions in the reverse direction exist:

▶ **Question 2.** Is it possible to translate a space-efficient SNN for a given task into a space-efficient streaming algorithm for that task?

An affirmative answer to both of these questions would imply that the streaming and SNN models are, roughly speaking, computationally equivalent. A priori, it is unclear if this is the case. On the one hand, streaming algorithms have the potential to be more space-efficient than SNNs. For example, a space-efficient algorithm may still have a lengthy description, which is not taken into account in its space complexity. In the SNN setting, where the algorithm description and memory are both encoded by the auxiliary neurons in the network and their connections, a lengthy description may lead to a large, and hence not space-efficient network.

On the other hand, SNNs have the potential to be more space-efficient than streaming algorithms. For example, a randomized SNN with a large number of input neurons but a small number of auxiliary neurons may have a large number of random bits encoded in random connections between its inputs and auxiliary neurons. These bits are not counted as part of its space complexity. In contrast, a streaming algorithm that requires persistent access to many random bits must store these bits, possibly leading to large space complexity.

1.1 Our Results

We take the first steps towards formally understanding the connections between streaming algorithms and spiking neural networks. The first part of the paper is devoted to studying upper bounds for SNNs, addressing Question 1. We design space-efficient neural networks for a wide class of streaming problems by simulating their respective streaming algorithms. These simulations must overcome several challenges in implementing traditional algorithms in neural networks. Most notably, in an SNN, the spiking status of the auxiliary neurons encodes the working memory of the algorithm, and their connections encode the algorithm

itself. A space-efficient network with few auxiliary neurons thus inherently has limited ability to express complex algorithms. In many data-stream algorithms, the target space complexity is only polylogarithmic in the input size, making this challenge significant. Additionally, unlike traditional algorithms, a neural network evolves continuously in response to its inputs. This leads to synchronization issues – for example, if an input is not presented for a sufficient number of rounds, the firing status of the network may not converge to a proper state before the next input is presented.

The second part of the paper focuses on lower-bound aspects, addressing Question 2. We show that any space-efficient neural network can be translated into a space-efficient streaming algorithm, while paying a small additive term (logarithmic in the stream length/universe size). For deterministic SNNs, such a reduction is not difficult. For randomized SNNs, the reduction is more involved, as it must account for the large number of random bits that may be implicitly stored in the random edge weights of the network. Throughout, we use the $\widetilde{O}()$ notation to hides factors that are poly-logarithmic in n, m and $1/\delta$, where n is the size of the domain, m is a bound on the stream length and δ is the error parameter.

1.1.1 Efficient Streaming Algorithms Yield Efficient SNNs

We consider data-stream tasks in which each update is an insertion or deletion of an integer item $x \in [n]$, and f is a function of the frequency vector $\bar{z} \in \mathbb{R}^n$ of these items. In the streaming setting, each update can be thought as an n-length vector with a single ± 1 entry, corresponding to an item insertion or deletion. In the SNN setting, each update may be encoded as the firing of one of n input neurons along with a sign neuron indicating if the update is an increment or a decrement. Or, the update may be encoded via $O(\log n)$ input neurons, indicating the item to be inserted or deleted. These different encodings correspond to different natural settings – the first corresponds to a network that collects firing statistics from a large set of inputs and the second to a network that records statistics on a large number of possible input patterns, encoded in the spiking patterns of a smaller number of input neurons.

In either case, each input is presented for some *persistence time*, a certain number of rounds in which the input is fixed to allow the network state to converge before the next input is presented.

Linear sketching. A linear sketching algorithm is a streaming algorithm in which the state of the algorithm is a linear function of the updates seen so far. In particular, the state can be represented as the multiplication of a sketching matrix $A \in \mathbb{R}^{r \times n}$ with the frequency vector $\bar{z} \in \mathbb{R}^n$. Such algorithms have many useful properties applicable in both the turnstile setting and in distributed settings. For example, the additive nature of these algorithms allows one to split the data-stream across multiple sites, which can process the data in an independent manner. Additionally, the obliviousness of linear sketching algorithms to the ordering of the stream yields an efficient generic derandomization scheme using the Nisan's PRG for space bounded computation [18]. Linear sketching algorithms constitute the state-of-the-art algorithms for essentially all problems in the turnstile model, including heavy-hitters, coresets for clustering problems [19], and ℓ_p estimation [9]. In fact, Li, Nguyen and Woodruff [27] present a general reduction from the streaming turnstile model to linear sketching. This reduction, and its caveats have been further studied in a recent work by Kallaugher and Price [21]. Given their ubiquity in turnstile streaming algorithms, an important step in designing space-efficient SNNs for data-stream problems is an efficient implementation of linear sketching in the neural setting. We give such an implementation:

▶ **Theorem 3** (Linear Sketch). *Let $\mathcal{A}$ be an algorithm approximating a function $f(\bar{x})$ in the turnstile model using a linear sketch with an integer matrix A of size $r \times n$. Let ℓ be a bound on the maximum entry in $|A\bar{x}|$ for every item $\bar{x}$. There exists a network $\mathcal{N}$ with $n+1$ input neurons, $r \cdot (\lceil \log \ell \rceil + 1)$ output neurons, $O(r \cdot \log \ell)$ auxiliary neurons which implements $\mathcal{A}$ in the following sense. The first n input neurons $x = (x_1, \ldots, x_n)$ represent the inserted item $[1, n]$, and the additional input neuron s indicates the sign of the update. Each input update has a persistence time of $O(\log \ell)$ rounds. The output neurons are divided into r vectors $\bar{y}_1, \ldots, \bar{y}_r$ each of length $\log \ell$, and r neurons $s_1, \ldots, s_r$. For every $i \in \{1, \ldots, r\}$, the decimal value of the binary vector $\bar{y}_i$ is equal to the absolute value of the ith entry of $A \cdot \bar{z}$, and the sign neuron s_i indicates the sign, where $\bar{z}$ is the summations of all input items presented in the current stream.*

Theorem 3 applies to linear sketches using integer matrices, which are commonly used, see [27]. Via scaling, the construction can be extended to rational matrices as well. We note that the network of Theorem 3 does not implement the "decoding" step which estimates $f(\bar{z})$ from $A \cdot \bar{z}$. This step depends on the problem being solved, however it is often very simple and thus implementable via a space-efficient SNN. E.g., in ℓ_p norm estimation one might just have to compute the ℓ_p norm of $A \cdot \bar{z}$ [18]. In frequency estimation, one might have to compute an average of a subset of entries in $A \cdot \bar{z}$ [5].

Beyond our generic linear sketching reduction, we give neural solutions for two challenging problems in the insertion-only model, namely, distinct elements and median estimation. These simulation results are less general and provide several tools for bypassing critical obstacles that arise in streaming to SNN reductions.

Distinct elements. In the *distinct elements problem* one must approximate the number of distinct items appearing in a data-stream with repeated items. It is well known that an exact solution by a single-pass streaming algorithm requires linear space. In fact, as we discuss later on, one can also show that the exact computation requires linear space in the SNN setting. Therefore, we restrict our attention to $(1+\epsilon)$ approximation for the number of distinct elements for any $\epsilon \in (0,1)$. This problem has been studied thoroughly in the streaming literature [6, 2, 12, 13, 22, 3, 20, 44, 1].

In this work, we provide an efficient neural implementation for the well-known LogLog streaming algorithm by [12, 13]. The LogLog and its improved variant the hyper-Loglog algorithms provide sub-optimal space bounds, but due to their simplicity they are commonly used in practice. As we will see, they are efficiently implementable in the neural setting. In addition, we provide a nearly matching space lower bound.

▶ **Theorem 4** (Neural Computation of Distinct Elements). *For every $n \in \mathbb{N}$, $\epsilon, \delta \in (0,1)$, given n input neurons $\bar{x}$ representing the elements in $[n]$ there exists a network $\mathcal{N}$ with $\log n$ output neurons and $\widetilde{O}(1/\epsilon^2)$ auxiliary neurons that encode the logarithm of an $(1 \pm \epsilon)$ approximation of the number of distinct elements in the current stream, with probability $1-\delta$. In addition, any SNN requires $\Omega(\log n + 1/\epsilon^2)$ neurons to compute an $(1 \pm \epsilon)$ approximation for the problem, with constant probability.*

The lower bound is obtained via a communication complexity reduction that mimics the corresponding streaming reduction. We note that this reduction works perfectly, i.e., without any asymptomatic loss in the space-bound (compared to the streaming bound).

Count-Min sketch. A common tool used in many of the streaming algorithms is the Count-Min sketch data structure, which maintains frequency estimates for all items in a stream. Count-Min sketch is in fact a linear sketch, and thus can be implemented via

Theorem 3. However, it is not immediately clear how to implement certain important operations, like approximate frequency (count) queries via this reduction. We thus provide a direct implementation. Our implementation applies in the setting where there are $O(\log n)$ input neurons representing each insertion/deletion of an item $x \in [n]$. However, it can easily be extended to the setting in which there are n input neurons, one for each item.

▶ **Definition 5** (Count-Min Sketch [10]). *Given parameters $\epsilon, \delta > 0$, the Count-Min sketch is a probabilistic data structure that serves as a frequency table of items in a stream. It supports two operations: (i)* $\mathrm{inc}(x)$ *increases the frequency of x by one; (ii)* $\mathrm{count}(x)$ *returns an $(1+\epsilon)$ approximation of the frequency of x with probability $1-\delta$.*

For given parameters $\epsilon, \delta > 0$, the Count-Min sketch data structure contains $\ell = O(\log 1/\delta)$ hash tables $T_1, \ldots T_\ell$ each with $b = O(1/\epsilon)$ bins, and each table T_i is indexed using a different pairwise-independent hash function h_i. The $\mathrm{inc}(x)$ operation applies $T_i[h_i(x)] \leftarrow T_i[h_i(x)]+1$ for every $i \in [\ell]$. The $\mathrm{count}(x)$ operation returns $\min_{i\in[\ell]} T_i[h_i(x)]$, which is shown to provide a good approximation for the frequency of x. The Count-Min data structure is used in many streaming algorithms including heavy-hitters, range queries, quantile estimation, and more. We provide an efficient implementation of a Count-Min sketch data structure, and show:

▶ **Theorem 6** (Neural Implementation of Count-Min Sketch). *For every $n, m \in \mathbb{N}$ and $\epsilon, \delta \in (0,1)$ there exists a network $\mathcal{N}$ with $\log n$ input neurons, $O(1/\epsilon \cdot \mathrm{poly}(\log m, \log 1/\delta)))$ auxiliary neurons, and $\widetilde{O}(1)$ persistence time that implements a Count-Min sketch with approximation ration $(1+\epsilon)$ and success probability $1-\delta$, for an input stream of length at most m.*

Our neural implementation of the Count-Min sketch can immediately be used to give, e.g., a simple neural approximate heavy-hitters algorithm, which returns TRUE if a presented item has frequency $\geq m/k$ in a data-stream for some integer k, and FALSE if it has frequency $\leq (1-\epsilon)m/k$. Setting $\epsilon' = O(\epsilon/k)$, a $\mathrm{count}(x)$ query will return a frequency estimate $\geq m/k$ for any true heavy-hitter x and $\leq m/k$ for any x with frequency $\leq (1-\epsilon)m/k$. By keeping a counter for m using $O(\log m)$ neurons and performing a comparison operation with the output of $\mathrm{count}(x)$, we can thus solve the heavy hitters problem. Other applications of Count-Min sketch require more complex processing of the data structure's output. To illustrate how this processing can be implemented efficiently in an SNN, we detail one such application, to median approximation.

Approximate median. One of the most fundamental statistical measures of a data-stream is its quantile. The 1/2-quantile known as the median, attracts most attention in the streaming literature [37, 36, 5, 7]. Its non-linearity nature makes it considerably harder to maintain compared to its linear cousin, the mean. As in many other streaming problems, the exact computation of the median requires linear space both in the streaming and in the SNN setting (as will be discussed later on). This motivates the study of the relaxed $(1+\epsilon)$ approximation task. In the latter, the algorithm is allowed to output an item j provided that the total number of items with value at most j is in $[m/2 - \epsilon m, m/2 + \epsilon m]$.

Cormode and Muthukrishnan [10] presented an elegant streaming algorithm for this problem using a space of $\widetilde{O}(1/\epsilon)$ bits. The algorithm is based on the Count-Min sketch data structure, combined with a dyadic decomposition technique that is used in a number of other streaming algorithms. One of our key technical algorithmic contributions is in providing an efficient neural implementation of this algorithm.

▶ **Theorem 7** (Approximate Median). *For every $n, m \in \mathbb{N}$ and $\epsilon, \delta \in (0,1)$, there exists a neural network $\mathcal{N}_{n,m}$ solving the ϵ-approximate median problem using $O(1/\epsilon \cdot \mathrm{poly}(\log m, \log n, \log 1/\delta))$ auxiliary neurons and persistence time $\widetilde{O}(1)$ with probability $1-\delta$.*

1.1.2 Streaming Lower Bounds Yield SNN Lower Bounds

Our second contribution focuses on Question 2, showing that space-efficient SNNs can be translated into space-efficient streaming algorithms, and thus that lower bounds in the streaming model imply lower bounds in the neural setting. The underlying intuition for this transformation is based on the following observation.

▶ **Observation 8.** *A spiking neural network with deterministic edge weights and n input neurons and S non-input neurons can be simulated by a streaming algorithm using S bits of space.*

In the SNN model, the spiking behavior of neurons in a given round depends only on the firing states of their incoming neighbors in the previous round. Thus, to simulate the behavior of the network as one pass over the data-stream, it is sufficient to maintain the firing states of all non-input neurons in the network, thus storing S bits of information. When the edge weights of the network are randomly sampled such a small-space simulation becomes more involved. The explicit storage of all the edge weights might be too costly since there can be $\Omega(nS + S^2)$ edges in a network with n input neurons and S non-inputs. Nevertheless, we show that a small-space simulation is still possible using a pseudorandom number generator, if we pay an additive logarithmic overhead in the length of the stream and universe size.

▶ **Theorem 9.** *Any SNN $\mathcal{N}$ with n input neurons, S non-input neurons for $S = \text{poly}(n)$, and $\text{poly}(n)$ persistence time can be simulated over a data-stream of length m using a total space of $O(S + \log(nm))$. The success guarantee of the simulation is $1 - 1/\text{poly}(n, m)$.*

Theorem 9 is a powerful tool, since it lets us apply any streaming space lower bound (of which there are many) to give an SNN lower bound, with a loss of an $O(\log(nm))$ factor. In some cases, we can avoid this loss by more directly considering the lower-bound technique. This is obtained when the streaming lower bounds are derived via a reduction to communication complexity with shared randomness that can be applied in the SNN setting with no loss. For example, using this tighter approach we show that our neural network for the distinct elements problem is nearly space-optimal (see the full version for details).

1.2 Preliminaries

Spiking neural networks. A deterministic neuron u is modeled by a *deterministic* threshold gate. Letting $b(u)$ to be the threshold value of u, then u outputs 1 if the weighted sum of its incoming neighbors exceeds $b(u)$. A *spiking neuron* is modeled by a probabilistic threshold gate, which fires with a sigmoidal probability that depends on the difference between its weighted incoming sum and $b(u)$.

A *Neural Network* (NN) $\mathcal{N} = \langle X, Z, Y, w, b\rangle$ consists of n input neurons $X = \{x_1, \ldots, x_n\}$, m output neurons $Y = \{y_1, \ldots, y_m\}$, and k auxiliary neurons $Z = \{z_1, ..., z_k\}$. In spiking neural networks (SNN), the neurons can be either deterministic threshold gates or probabilistic threshold gates. The directed weighted synaptic connections between $V = X \cup Z \cup Y$ are described by the weight function $w : V \times V \rightarrow \mathbb{R}$. A weight $w(u, v) = 0$ indicates that a connection is not present between neurons u and v. Finally, for any neuron v, the value $b(v) \in \mathbb{R}$ is the bias value (activation threshold). Additionally, each neuron is either inhibitory or excitatory: if v is inhibitory, then $w(v, u) \leq 0$ and if v is excitatory, then $w(v, u) \geq 0$ for every u. This restriction arises from the biological structure of the neurons.

Network dynamics. The network evolves in discrete, synchronous rounds as a Markov chain. The firing status of every neuron u in round τ denoted as $\sigma_\tau(u)$, depends on the firing status of its neighbors in round $\tau - 1$, via a standard sigmoid function. For each neuron u, and each round $\tau \geq 0$, let $\sigma_\tau(u) = 1$ if u fires (i.e., generates a spike) in round τ. For every neuron u and every round $\tau \geq 1$, let $\mathrm{pot}(u, \tau) = \sum_{v \in V} w(v, u) \cdot \sigma_{\tau-1}(v) - b(u)$ denote the membrane potential at round τ. A deterministic threshold gate u fires in round τ iff $\mathrm{pot}(u, \tau) \geq 0$. A probabilistic threshold gate fires with a probability that depends on $\mathrm{pot}(u, \tau)$. All our network constructions in this work use deterministic threshold-gates, and the randomness of the network comes from the randomized selection of the edge weights.

Neural networks for data-stream problems. A data-stream problem is defined by a relation $P_n \subset \mathbb{Z}^n \times \mathbb{Z}$. The length of the stream is upper bounded by some integer m. Each data-item is represented by a binary vector of length n. A value $i \in [1, n]$ is represented by having the ith input neuron fire while all other input neurons are idle. Each input is presented for some persistence time, at the end of which the output neurons of the network encode (in binary) the evaluation of a given relation over the current stream. To avoid cumbersome notation, we may assume that m and n are powers of 2.

1.3 Basic Tools

Our constructions are based on the following neural network modules.

Neural timers and counters. For a given time parameter t, a neural timer $\mathcal{NT}_t$ is an SNN network that consists of an input neuron x, an output neuron y, and additional auxiliary neurons. The network satisfies that in every round τ, y fires in round τ iff x fires at some round τ' for $\tau' \in [\tau - t, \tau]$. It is fairly trivial to design a neural timer network with $O(t)$ auxiliary neurons. [15] presented a construction of a considerably more succinct network $\mathcal{NT}_t$ with only $O(\log t)$ neurons. In the related setting of neural counting, the network is required to encode the *number* of firing events of its input neuron within a given time window. Specifically, given time parameter t, a *neural counter network* $\mathcal{NC}_t$ has a single input neuron x, and $\lceil \log t \rceil$ output neurons that encode the number of firing events of x within a span of t rounds.

► **Fact 10.** *[30, 15] For every integer parameter t, there exist (i) a neural timer network $\mathcal{NT}_t$ with $O(\log t)$ neurons, and (ii) a neural counter network $\mathcal{NC}_t$ with $O(\log t)$ auxiliary neurons, such that for every round i, the output neurons encode f_i by round $i + O(\log t)$ where f_i is the number of firing events up to round i. Both networks $\mathcal{NT}_t$ and $\mathcal{NC}_t$ are deterministic.*

We next describe *new* tools introduced in this work which will be heavily used in our constructions. Missing proofs are deferred to the full version of the paper.

Potential encoding. Our SNN constructions are based on a module that encodes the potential p of a given neuron x by its binary representation using $\log p$ neurons. We will use this modules in the constructions of Theorem 3 and Lemma 14.

► **Lemma 11.** *Let x be a deterministic neuron such that $\mathrm{pot}(x, t') \leq 2^\ell$ for every $t' \in [t, t + O(\ell)]$ for some integer $\ell \in \mathbb{N}_{>0}$. There exists a deterministic network $\mathrm{POT}_\ell(x)$ which uses ℓ identical copies of x (the same input and bias), 2ℓ auxiliary neurons, and ℓ output neurons $y_0 \ldots y_{\ell-1}$ that encodes $\mathrm{pot}(x, t)$ in a binary form within $O(\ell)$ rounds.*

Implementing pairwise-independent hash functions. Many streaming algorithms in the insertion only model are based on the notion of pairwise independent hash functions.

▶ **Definition 12** (Pairwise Independence Hash Functions). *A family of functions* $\mathcal{H} : [a] \to [b]$ *is* pairwise independent *if for every* $x_1 \neq x_2 \in [a]$ *and* $y_1, y_2 \in [b]$*, we have:* $\Pr[h(x_1) = y_1$ *and* $h(x_2) = y_2] = 1/b^2$*.*

For ease of notation, assume that a, b are powers of 2.

▶ **Definition 13** (Pairwise Independence Hash SNN). *Given two integers* a, b*, a* pairwise independent hash network $\mathcal{N}_{a,b}$ *is an SNN with an input layer of* $\log a$ *neurons, an output layer of* $\log b$ *neurons, and a set of* s *auxiliary spiking neurons. For every input value* x *presented at round* t*, let* $\mathcal{N}(x)$ *be the value of the output layer after* $\tau_{a,b}$ *rounds. Then, for every* $x \neq x' \in [a]$*, it holds that* $\Pr[\mathcal{N}(x) = \mathcal{N}(x')] = 1/b$*.*

We show a neural network implementation of a pairwise independent hash function using the construction of pairwise hash function by [4].

▶ **Lemma 14** (Neural Implementation of Pairwise Indep. Hash Function). *There exists a pairwise independent hash network* $\mathcal{N}_{a,b}$ *with* $s = O(\log b \cdot \log\log a)$ *auxiliary neurons that computes the output value of each input within* $O(\log\log a)$ *rounds (persistence of the input neurons).*

2 Linear Sketching

A linear sketching algorithm is a streaming algorithm in which the state of the algorithm at time t is a linear function of the updates seen up to time t. We start with a formal definition.

▶ **Definition 15** (Linear Sketching Algorithm, [23]). *A linear sketching algorithm* $\mathcal{L}$ *gives a method for processing a vector* $\bar{x} \in \mathbb{R}^n$*. The algorithm is characterized by a (typically randomized) sketch matrix* $A \in \mathbb{R}^{r \times n}$*, and by a possibly randomized decoding function* $f : \mathbb{R}^r \to O$ *where* O *is some output domain. Algorithm* $\mathcal{L}$ *is executed by first computing* $A \cdot \bar{x}$ *and then outputting* $f(A \cdot \bar{x})$*. Note that* f *only takes* $A \cdot \bar{x}$ *as input,* f *cannot depend on* A *in any other way, e.g. it cannot share randomness with* A*.*

Linear sketching algorithms provide the state-of-the-art space bounds for a large collection of problems in the turnstile model.

The challenge and our approach. Throughout we assume the sketching matrix is integral, i.e., $A \in \mathbb{Z}^{r \times n}$, which captures most of the classic implementations in the turnstile model. We start by describing a straw man approach for computing the value $A\bar{x}$ in the neural setting: Take a single-layer neural network with an input layer of length $n+1$ and an output layer of length r. Specifically, the input layer contains n neurons $x_1, \ldots, x_n$ that represent the absolute value of the update, and an additional *sign* neuron that indicates the sign of the update. For example, an update vector $[0, 0, -1, 0]$ is represented by letting $x_3 = 1$, $s = 1$ and $x_1, x_2, x_4 = 0$. The output layer is defined by r output neurons $y_1, \ldots, y_r$. The edge weights are specified by the matrix A where $w(x_j, y_i) = A_{i,j}$. It is then easy to verify that the weighted sum of the incoming neighbors of each neuron y_j (i.e., its potential) is the value of the jth bit in $A\bar{x}$.

This naive description fails for various reasons. First, from a biological perspective, each input neuron can be either inhibitory or excitatory. This implies that the sign of the outgoing edge weights of a given neuron must be either a plus (excitatory) or a minus (inhibitory).

Mathematically, this requires the sketch matrix A to be sign-consistent (i.e., the sign of all entries in a given raw are either a plus or a minus). However, in general, the given sketch matrix might not be sign-consistent. The second technicality is that the neurons $y_1, \ldots, y_n$ have a *binary* output (either firing or not) rather then an *integer* value. The third aspect to take into account is concerned with the update mechanism. Specifically, given a stream of data items, one should make sure that each data item would be processed exactly *once* by the network. This requires a more delicate update mechanism.

In the high-level, we handle the sign-consistency challenge by dividing the sketch matrix A into a non-negative matrix A^+ and a non-positive matrix A^- where $A = A^+ - A^-$. Then, given a new update $(\bar{x}, s)$, the network computes $A\bar{x}$ and $-A\bar{x}$ using $A^+\bar{x}$ and $A^-\bar{x}$. The final output $A\bar{x}$ is computed by using these values combined with the sign neuron s. To handle the second challenge, we use the module of Lemma 11 to translate the *potential* of each output neuron y_j (corresponding to the j'th bit in the sketch) into its binary representation. The output layer consists of $O(r \log n)$ output neurons that encode the value of the current r-length sketch. The complete implementation details are given to the full version.

Due the space consideration, the neural implementation for the *distinct-element problem* (Proof of Theorem 4) is deferred to the full version of the paper. In this proof we also demonstrate how to translate the streaming lower bound into a matching space lower bound for the neural setting.

3 Median Approximation

Before presenting the neural computation of the approximate median, we describe the neural implementation of the Count-Min Sketch and prove Theorem 6.

3.1 A Neural Implementation of Count-Min Sketch

We follow the streaming implementation of Count-Min by [10] described as follows. The algorithm maintains a data structure that consists of $\ell = O(\log 1/\delta)$ hash tables $T_1, \ldots T_\ell$, each with $b = O(1/\epsilon)$ bins, and each table T_i is indexed using a different pairwise-independent hash function h_i (i.e., the output domain of h_i is $\{0,1\}^{\log b}$). The operation $\text{inc}(x)$ increases the value in each bin $T_i[h_i(x)]$ for every $i \in [\ell]$. The $\text{count}(x)$ operation returns the value $\min_{i\in[\ell]} T_i[h_i(x)]$.

▶ **Fact 16** **([10]).** $\Pr[\text{count}(x) \notin (f(x), f(x) + O(m/b))] \leq 1/2^{\Omega(\ell)}$ *where* $f(x)$ *is actual frequency of* x *in the stream of length* m.

▶ **Definition 17** (Neural Count-Min Sketch)**.** *Given parameters* $\epsilon, \delta > 0$*, a neural Count-Min sketch network* $\mathcal{N}_{\epsilon,\delta}$ *has an input layer of* $\log n + 1$ *neurons denoted as* $a, x_1, \ldots x_{\log n}$*, an output layer of* $\log m$ *neurons* $y_1, \ldots y_{\log m}$*, and a set of* s *auxiliary neurons. The neurons* $x_1, \ldots x_{\log n}$ *encode the binary representation of an element* $x \in [n]$ *and the neuron* a *indicates whether this is an* inc *or* count *operation, where* $a = 1$ *indicates an* inc *operation. For every fixed input value* $x = (x_1, \ldots x_{\log n})$ *presented at round* t *and* $a = 0$ *(i.e., a* count *operation), let* $\mathcal{N}_{\delta,\epsilon}(x)$ *be the value encoded in binary by the output layer* $y_1, \ldots y_{\log m}$ *in round* $t + \tau_{n,m}$*. It holds that* $\Pr[\mathcal{N}_{\delta,\epsilon}(x) \notin (f(x), f(x) + O(\epsilon m'))] \leq \delta$*, where* $m' \leq m$ *is the stream length by round* t *and* $f(x)$ *is the current frequency of* x*.*

We first describe the network construction to support the $\text{inc}(x)$ operation. Then we explain the remaining network details for implementing a $\text{count}(x)$ operation.

Supporting inc(x) operation. The network contains $\ell = O(\log 1/\delta)$ sub-networks $\mathcal{H}^1_{n,b}, \ldots, \mathcal{H}^\ell_{n,b}$ each implements a pairwise independent hash function $h_i : \{0,1\}^{\log n} \rightarrow \{0,1\}^{\log b}$ using Lemma 14. The output vector of each network $\mathcal{H}^i_{\log n,b}$ is denoted by $\bar{h}_i$ for every $i \in \{1, \ldots, \ell\}$. Every $\bar{h}_i$ has an inhibitory copy $\bar{h}'_i$.

For each sub-networks $\mathcal{H}^i_{\log n,b}$, and for every value $j \in \{1, \ldots, b\}$, the network contains a counter sub-network that counts the number of data-items x in the stream that satisfies $h_i(x) = j$. Every counter network is implemented by a neural-counter network from Fact 10 with time parameter $t = m$. Let $\mathcal{C}_{i,1}, \ldots, \mathcal{C}_{i,b}$ be the neural counter networks corresponding to the ith hash network $\mathcal{H}^i_{\log n,b}$. The counter $\mathcal{C}_{i,j}$ is updated based on the values of the output neurons $\bar{h}_i$ as follows. For every counter $\mathcal{C}_{i,j}$ the network contains an index neuron $c_{i,j}$ with input from $\bar{h}_i$ and $\overline{h'}_i$ which fires only if[1] $\text{dec}(\bar{h}_i) = j$. The input to the counter $\mathcal{C}_{i,j}$ denoted as $e_{i,j}$ is an AND gate between the input neuron a and the index neuron $c_{i,j}$, firing in inc(x) operations where $h_i(x) = j$. To make sure the counter is incremented once per inc(x) operation, the network contains an inhibitory neuron denoted as $e'_{i,j}$ which has the same incoming edges and weights as $e_{i,j}$, that inhibits the neurons $e_{i,j}$, $c_{i,j}$ and a. This guarantees that $e_{i,j}$ would be active for exactly *one* round per inc(x) operation.

Supporting count(x) operation. To support a count(x) operation, for each counter $\mathcal{C}_{i,j}$, the network includes $\log m$ neurons $\bar{s}_{i,j} = s^1_{i,j}, \ldots s^{\log m}_{i,j}$ which hold the value stored in the counter $\mathcal{C}_{i,j}$ such that $h_i(x) = j$. Each neuron $s^k_{i,j}$ is an AND gate of the index neuron $c_{i,j}$ and the jth output neuron of $\mathcal{C}_{i,j}$. In addition, for every $i \in \{1, \ldots, \ell\}$ there are $\log m$ neurons $\bar{g}_i = g_{i,1}, \ldots g_{i,\log m}$ where the jth neuron $g_{i,j}$ is an OR gate of all the jth neurons of the vectors $\bar{s}_{i,1}, \ldots, \bar{s}_{i,b}$. As a result, $\bar{g}_i$ encodes the value stored in $h_i(x)$. Finally, the output value is set to be the *minimum value* of $\text{dec}(\bar{g}_1), \ldots, \text{dec}(\bar{g}_\ell)$ using the minimum computation network of [34]. The correctness analysis is deferred to the full version of the paper.

3.2 Neural Computation of the Approximate Median

In this section, we present our main technically involved algorithmic result for computing an estimate for the median of the data-stream.

▶ **Definition 18** (Approximate Median). *Given $\epsilon, \delta \in (0,1)$ and a stream $\mathcal{S} = \{x_1, x_2, \ldots x_m\}$ with each $x_i \in [n]$, in the* approximate median *problem, it is required to output an element $x_j \in \mathcal{S}$ whose rank is $m/2 \pm \epsilon m$ with probability at least $1 - \delta$.*

For ease of notation, assume that n is power of 2. Our neural solution is based on the streaming algorithm of [10], that uses $\widetilde{O}(1/\epsilon)$ space. Up to the logarithmic terms, this space-bound is known to be optimal [24].

▶ **Fact 19** (Theorem 5 [10]). *For every $\epsilon, \delta \in (0,1)$, there exists a randomized streaming algorithm for computing the ϵ-approximate median with probability $1 - \delta$ and $\widetilde{O}(1/\epsilon)$ space.*

We start by providing a high-level exposition of this streaming algorithm, and then explain its implementation in the neural setting. The latter turns out to be quite involved, yet demonstrating the expressive power of SNN networks.

[1] For implementation reasons, verifying that $\text{dec}(\bar{h}_i) = j$ requires input from both $\bar{h}_i$ and $\overline{h'}_i$.

A high-level description of the streaming algorithm. The algorithm is based on applying a binary search over *range queries* which, roughly speaking, compute the frequency of the elements in a given range.

▶ **Definition 20** (Range Queries). *Given a data-stream of numbers $\mathcal{S} = \{x_1, \ldots, x_m\}$ with each $x_i \in [n]$, a range query receives a range of number $[a, b] \subseteq [1, n]$ and returns the frequency of the items $\{a, a+1, \ldots, b\}$ in the stream $\mathcal{S}$.*

To support range queries with small space, the algorithm maintains $\log n$ data structures of Count-Min sketch, for each of the $\log n$ *dyadic intervals* of $[n]$.

▶ **Definition 21** (Dyadic Intervals). *The* dyadic intervals *of the set $[n]$ are a collection of $\log n$ partitions of n, $\mathcal{I}_1 \ldots, \mathcal{I}_{\log n}$ such that*

$$\begin{aligned}
\mathcal{I}_0 &= \{\{1\}, \{2\}, \{3\}, \ldots, \{n\}\} \\
\mathcal{I}_1 &= \{\{1,2\}, \{3,4\}, \{5,6\}, \ldots, \{n-1, n\}\} \\
\mathcal{I}_2 &= \{\{1,2,3,4\}, \{5,6,7,8\}, \ldots, \{n-3, n-2, n-1, n\}\} \\
&\ldots \\
\mathcal{I}_{\log n} &= \{\{1, 2 \ldots n\}\}
\end{aligned}$$

Note that every range $[i, j] \subseteq [n]$ can be written as a union of at most $\log n$ sets from the dyadic intervals. Hence, by introducing $\log n$ Count-Min data structures with parameters $\delta' = \log(\log n/\delta)$ and $\epsilon' = \epsilon/\log n$ for dyadic-intervals of $[n]$, we can answer range queries within an additive error of $m \cdot \epsilon$ with probability $1 - \delta$. The approximated median is obtained by employing a Binary search over the range queries [2].

▶ **Definition 22** (SNN for the Approximate Median Problem). *Given two integers n, m and additional parameters $\epsilon, \delta \in (0,1)$, an approximate-median network $\mathcal{N}_{n,m}$ has an input layer of $n+1$ neurons, an output layer of $\log n$ neurons and a set of s auxiliary neurons. The input neurons are denoted as $(a, x_1, \ldots, x_n)$ where the neuron a indicates whether this is a median query or an insertion operation. When the input layer represents a median query, the neuron a fires and the neurons $x_1, \ldots x_n$ are idle. For every round t, let $\mathcal{S}_t = \{a_1, a_2, \ldots a_t\}$ be the data-stream presented as input to the network by round t. For any median-query presented in round t, by round $t + \tau_{n,m}$ the output layer encodes an element $y \in \mathcal{S}_t$ whose rank in $\mathcal{S}_t$ is $t/2 \pm \epsilon t$ with probability at least $1 - \delta$.*

The challenge: The crux of the streaming algorithm is based on a binary search over range queries. A-priori, it is unclear how to implement such a search using a poly-logarithmic number of neurons. Specifically, the (implicit) decision tree that governs the binary search has a linear size. Since the neural network (unlike the streaming algorithm) has to hard-wire the algorithm description, the explicit encoding of the search tree leads to a linear space solution. Our key contribution is in showing a succinct network construction that simulates the binary search of the streaming algorithm using a nearly matching space bound.

We next provide a high-level description of the network. Recall that the type of the operation is represented by the input neuron a, where $a = 1$ represents a median query.

[2] The same algorithm can be applied for any quantile estimation.

Supporting an insertion operation. In the high level, the network contains 3 parts (1) a set of $\log n$ neurons that encode the inserted element in its binary form, (2) a neural counter that counts the length of the current stream, and (3) $\log n$ Count-Min sketch sub-networks that maintain the frequencies of the $\log n$ dyadic intervals of $[n]$.

1. The n-length input vector $\bar{x}$ is connected to $\log n$ neurons $\bar{x}' = (x'_1, \ldots, x'_{\log n})$ such that $\bar{x}'$ encodes the binary representation[3] of the element presented in the input neurons $\bar{x}$.
2. The network contains a counter sub-networks $\mathcal{NC}_m$ for counting the number of data-items inserted so far (i.e., the current length of the stream). The counter is implemented by a neural-counter network from Fact 10 with time parameter $t = m$. The input neuron to the $\mathcal{NC}_m$ sub-network denoted as a' is an OR gate of the input neurons $\bar{x}$. To make sure the counter is incremented once per insertion operation, the network contains an inhibitory copy of a' denoted as r', which inhibits a' and the neurons $\bar{x}$. As a result, the input neuron a' will be active for exactly *one* round per insertion operation.
3. The network contains $\log n$ sub-networks $\mathcal{C}_1, \ldots \mathcal{C}_{\log n}$ each implements a Count-Min sketch with parameters n, m and $\epsilon' = O(\epsilon/\log n)$, $\delta' = O(\delta/\log n)$ using Theorem 6. For each Count-Min sketch sub-networks $\mathcal{C}_i$, let $\bar{z}_i = (z_{i,1}, \ldots z_{i,\log n})$ and b_i be its input layer, where the neuron b_i indicates whether the operation is inc or count. The neuron b_i is an OR gate of the neurons in $\bar{x}$.
   The input neurons $\bar{z}_i$ are connected to the binary representation of the input $\bar{x}'$ in the following manner. For every $i \in \{1, \ldots, \log n\}$ and every $j \geq i$, the neuron x'_j is connected to the neuron $z_{i,j}$. In addition, for every $j < i$ the neuron $z_{i,j}$ serves as an OR gate between the neurons of $\bar{x}'$. The neurons b_i, $\bar{z}_i$ are equipped with self-loops. The Count-Min sketch sub-networks are then modified such that these neurons will be inhibited once the computation is complete (by the inhibitory neurons $e'_{i,j}$ of each sub-networks respectively).

Supporting a median query. Given a median query, the network computes the approximate median by employing at most $\log n$ steps of binary search. In every step[4] $i \in \{\log n, \ldots, 1\}$, the network obtains a current candidate for the median denoted by χ_i. Initially, $\chi_{\log n} = n/2$. Each χ_i would be provided as input for the ith Count-Min sketch $\mathcal{C}_i$. The output neurons of $\mathcal{C}_i$ would then define the next candidate χ_{i-1}. Specifically, depending on the rank estimation of χ_i, the network defines the new search range. The width of the search range would be cut by a factor 2 in every step i. Consequently, the algorithm will be using the Count-Min sketch $\mathcal{C}_{i-1}$ which is defined over a partitioning $\mathcal{I}_{i-1}$ in which each set is smaller by factor 2 compared to $\mathcal{I}_i$. We now describe these steps in more details.

1. For every $i \in \{\log n, \ldots, 1\}$ the network contains an additional Count-Min sub-networks $\mathcal{C}'_i$ which counts the frequencies of the data-elements (similar to $\mathcal{C}_1$). This additional Count-Min sub-networks will be useful in a scenario where the median item j^* has a very large frequency. In such a case, the frequency of the range $[1, j^*]$ is too large and the frequency of $[1, j^* - 1]$ is too small. This special case would be handled using the $\mathcal{C}'_i$ sub-networks.
2. For every $i \in \{\log n, \ldots, 1\}$ the network contains three *comparison* neurons s_i, g_i, e_i (corresponding to *smaller, greater* or *equal*). These neurons receive their input from the output neurons of the counters $\mathcal{C}_{\log n}, \ldots, \mathcal{C}_i$, and from the the output of the neural

[3] As discussed in the introduction our solution supports both types of input formats: $\log n$-bits of the binary representation or an n-length vector with one active entry.

[4] It is convenient to count the steps in a backward manner, as in the ith step the network will access the ith Min-Sketch module $\mathcal{C}_i$.

counter $\mathcal{NC}_m$. Let $\chi_i = \text{dec}(\bar{z}_i)$, this value would correspond to the median candidate at phase i of the binary-search. The firing states of the comparison neurons would be determined as follows. The neuron g_i would fire if the frequency estimation of $[1, \chi_i]$ is greater than $m'/2 + \epsilon/2m'$. The neuron s_i would fire if frequency estimation of $[1, \chi_i]$ is smaller than $m'/2 - \epsilon/2m'$. Finally, e_i would fire if the frequency estimation of $[1, \chi_i]$ is in the range $(m'/2 - \epsilon/2m', m'/2 + \epsilon/2)$.

3. For $i \in \{\log n, \ldots, 1\}$, every two consecutive sub-networks $\mathcal{C}_{i+1}$ and $\mathcal{C}_i$ are connected in a way that guarantees the following. Let χ_{i+1} be median candidate at phase $i+1$ of the binary search (i.e., that was fed as input to $\mathcal{C}_{i+1}$). Let $\text{freq}([x, y])$ be the estimated frequency of the range $[x, y]$ obtained by the Count-Min sketch networks $\mathcal{C}_{\log n}, \ldots, \mathcal{C}_{i+1}$. Then candidate χ_i is defined as:

$$\chi_i = \begin{cases} \chi_{i+1} - 2^{i-1}, & \text{if } \text{freq}([1, \chi_{i+1}]) > m'/2 + \epsilon/2m' \\ \chi_{i+1} + 2^{i-1}, & \text{if } \text{freq}([1, \chi_{i+1}]) < m'/2 - \epsilon/2m' \end{cases}.$$

In the remaining case where $\text{freq}([1, \chi_{i+1}]) \in [m'/2 \pm \epsilon/2m']$, the candidate χ_{i+1} is returned as the output result. Its value will be encoded by the output neurons of the network. The complete description and its analysis is deferred to the full version of the paper.

4 Streaming Lower Bounds Yield SNN Lower Bounds

We conclude by addressing Question 2, giving a generic reduction that lets us simulate a space-efficient SNN with a space-efficient neural network. This establishes a tight connection between the two models – any streaming space lower bound yields a near-matching neural-space lower bound. Missing proofs of this section appear in the full version of the paper.

Complexity classes in the SNN model. For integer parameters n, m, S, let $\mathcal{SNN}_{\text{det}}(n, m, S)$ be the set of all data-stream problems $P_{n,m}$ defined over universe $[n]$ and stream length at most m that are solvable by a deterministic SNN with (i) at most $O(S)$ non-input neurons (i.e., auxiliary and output neurons) and (ii) polynomially bounded edge weights (by n and m). Let $\mathcal{SNN}^{\text{poly}}_{\text{det}}(n, m, S)$ be the class of all data-stream problems $P_{n,m}$ in $\mathcal{SNN}_{\text{det}}(n, m, S)$ whose network solution also have in addition a polynomial persistence time (in n and m). That is, the problems in $\mathcal{SNN}^{\text{poly}}_{\text{det}}(n, m, S)$ are solvable in polynomial-time by a deterministic SNN that has properties (i,ii).

We also consider the class of data-stream problems that are solvable by a randomized SNN. Let $\mathcal{SNN}_{\text{rand}}(n, m, S, \delta)$ be the set of all data-stream problems $P_{n,m}$ that are solvable by a randomized SNN with: (i) at most $O(S)$ non-input neurons, (ii) polynomially bounded edge weights, and (iii) $\leq \delta$ failure probability on any input. The class $\mathcal{SNN}^{\text{poly}}_{\text{rand}}(n, m, S, \delta)$ is a sub-class of $\mathcal{SNN}_{\text{rand}}(n, m, S, \delta)$ that requires also a polynomial persistence time.

Complexity classes in the streaming model. Let $\mathcal{ST}_{\text{det}}(n, m, S)$ be the class of all data-stream problems for which there exists a single-pass deterministic streaming algorithm for the problem using space $O(S)$ (potentially with exponentially large update time). Also, let $\mathcal{ST}_{\text{rand}}(n, m, S, \delta)$ be the class of all data-stream problems for which there exists a single-pass randomized streaming algorithm that solves the problem with failure probability $\leq \delta$ using space $O(S)$. One can also define the classes $\mathcal{ST}^{\text{poly}}_{\text{det}}(n, m, S)$ and $\mathcal{ST}^{\text{poly}}_{\text{rand}}(n, m, S, \delta)$ which require polynomial update time.

We start by showing that any deterministic SNN with space S for a given data-stream problem $P_{n,m}$ yields an S-space deterministic streaming algorithm for the problem.

▶ **Lemma 23.** *For every n, m, S, we have:*
$\mathcal{SNN}_{\text{det}}(n,m,S) \subseteq \mathcal{ST}_{\text{det}}(n,m,S)$ *and* $\mathcal{SNN}^{\text{poly}}_{\text{det}}(n,m,S) \subseteq \mathcal{ST}^{\text{poly}}_{\text{det}}(n,m,S)$.

Proof. Fix the parameters n, m, S, and consider a problem $\Pi \in \mathcal{SNN}_{\text{det}}(n,m,S)$. Let $\mathcal{N}$ be the SNN for the problem Π. Thus $\mathcal{N}$ has S auxiliary and output neurons. We now describe a streaming algorithm for Π that uses space S. The algorithm traverses the stream and feeds each item as an input to the network $\mathcal{N}$ (with sufficient large persistence time). Importantly, when considering the subsequent input item, the streaming algorithm only keeps the current firing states of the S auxiliary and output neurons. The correctness follows immediately by the correctness of the network $\mathcal{N}$. The space complexity is S bits corresponding to the firing states of the (non-input) neurons in $\mathcal{N}$. The proof that $\mathcal{SNN}^{\text{poly}}_{\text{det}}(n,m,S) \subseteq \mathcal{ST}^{\text{poly}}_{\text{det}}(n,m,S)$ is analogous since the update time of the streaming algorithm is polynomial in the network size and the persistence time of the network. ◀

Pseudorandomness for neural networks. Our next goal is to simulate space-efficient randomized SNNs for data-stream problems with small-efficient streaming algorithms. The main barrier arises in the case where the edge weights of the network $\mathcal{N}$ are chosen randomly according to some distribution. Since an S-space network with n input neurons might have $\Omega(Sn + S^2)$ edges, the explicit specification of the edge weights is too costly for our purposes.

To overcome this barrier, we will use Nisan-Wigderson type pseudorandom generators [39], which fool circuits of a given size, defined as follows:

▶ **Definition 24** (NW-type PRGs.). *A function* $\text{NWPRG}\colon \{0,1\}^{d(n)} \to \{0,1\}^n$ *is an* NW-type PRG against circuits of size $t(n)$ *if it is (i) computable in time* $2^{O(d(n))}$ *and (ii) any circuit C of size at most $t(n)$ distinguishes $U \leftarrow \{0,1\}^n$ from* $\text{NWPRG}(s)$, *where* $s \leftarrow \{0,1\}^{d(n)}$, *with advantage at most* $1/t(n)$.

▶ **Theorem 25.** *[17] Assume there exists a function in E =* $\text{DTIME}(2^{O(n)})$ *with circuit complexity* $2^{\Omega(n)}$. *Then, for any polynomial $t(\cdot)$, there exists a NW-type generator* $\text{NWPRG}\colon \{0,1\}^{d(n)} \to \{0,1\}^n$ *against circuits of size $t(n)$, where $d(n) = O(\log n)$.*

▶ **Lemma 26.** *If one does not restrict the running time of the PRG (and allows it to be computable by a non-uniform circuit), then a version of Theorem 25 holds* unconditionally*.*

Since an SNN with n input neurons, S non-input neurons for $S = \text{poly}(n)$, and polynomial persistence time can be computed in polynomial time, we have the following:

▶ **Lemma 27.** *Any SNN $\mathcal{N}$ with n input neurons, S non-input neurons for $S = \text{poly}(n)$, and persistence time* $\text{poly}(n)$ *in an m-length stream can be simulated using a total space of* $O(S + \log(nm))$. *The success guarantee of the simulation is* $1 - 1/\text{poly}(n,m)$.

Proof. Consider a (centralized, offline) algorithm that given an ordered stream of length $m' \leq m$ of elements in $[1,n]$ evaluates the output of the network $\mathcal{N}$ on that stream. This algorithm can be implemented in time $\text{poly}(n,m)$ and thus there exists a circuit of size $M = \text{poly}(n,m)$ that implements this algorithm. Our goal is to simulate this circuit using a random seed of length $O(\log(nm))$ while reducing the success guarantee by an additive term of $1/\text{poly}(n,m)$. To do that, we will use the PRG construction of Lemma 26 that given a random seed of size $O(\log(nm))$ fools the family of all circuits of size at most M with probability $1 - 1/\text{poly}(M)$.

We now describe how to simulate $\mathcal{N}$ using $O(\log(nm) + S)$ space. We store a seed of $O(\log(nm))$ random bits R and the current firing states of all non-input neurons in $\mathcal{N}$. Then, as we traverse the stream, for every data-item in the stream, the algorithm first applies

the NWPRG function on the random seed R and outputs $\mathrm{poly}(n,m)$ coins that are used to define the edge weights of $\mathcal{N}$, as well as to simulate the spiking decisions of the S neurons in the ith step. The evaluation time of this NWPRG function (i.e., outputting each coin) might be exponential. However, we will only store one pseudorandom coin at a time, when its value is required as part of an update. By knowing the states of the S non-input neurons from the previous step, and using the coins to determine the edge weights one at a time (requiring $O(\log n)$ space as the edge weights are polynomially bounded) one can simulate the firing pattern of the neurons in the given step. When the neurons are probabilistic threshold gates, the pseudorandom coins are also used to simulate the firing decisions of these neurons. While each firing probability is real-valued, it can be rounded to $1/\mathrm{poly}(n,m)$ accuracy (and hence determined by $O(\log(nm))$ random coins) without changing the probability of any output configuration by more than $1/\mathrm{poly}(n,m)$. The step ends with the computation of the firing states of all non-input neurons, which are stored for the next step. The success guarantee of using these coins rather than fresh random coins is decreased by an additive term of $1/\mathrm{poly}(n,m)$. ◀

Lemma 27 implies that any randomized SNN with space S that solves a streaming problem $P_{n,m}$ with probability $1-\delta$ in polynomial time translates into a randomized streaming algorithm for $P_{n,m}$ using space of $S+O(\log(nm))$. We therefore have:

▶ **Theorem 28.** $\mathcal{SNN}^{\mathrm{poly}}_{\mathrm{rand}}(n,m,S,\delta) \subseteq \mathcal{ST}_{\mathrm{rand}}(n,m,S+O(\log(nm)),\delta+1/\mathrm{poly}(n,m))$.

A useful implication of Theorem 28 is that any space lower-bound in the streaming model immediately translates into space lower-bound for networks that have a polynomial persistence time on the input stream. See the full version for the precise formulation.

References

1. Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. *Journal of Computer and system sciences*, 58(1):137–147, 1999.
2. Ziv Bar-Yossef, T. S. Jayram, Ravi Kumar, D. Sivakumar, and Luca Trevisan. Counting distinct elements in a data stream. In *Randomization and Approximation Techniques, 6th International Workshop, RANDOM 2002, Cambridge, MA, USA, September 13-15, 2002, Proceedings*, pages 1–10, 2002.
3. Jaroslaw Blasiok. Optimal streaming and tracking distinct elements with high probability. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 2432–2448, 2018. `doi:10.1137/1.9781611975031.156`.
4. J Lawrence Carter and Mark N Wegman. Universal classes of hash functions. *Journal of computer and system sciences*, 18(2):143–154, 1979.
5. Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. In *International Colloquium on Automata, Languages, and Programming*, pages 693–703. Springer, 2002.
6. Philippe Chassaing and Lucas Gerin. Efficient estimation of the cardinality of large data sets. *arXiv preprint math/0701347*, 2007.
7. Zhiwei Chen and Aoqian Zhang. A survey of approximate quantile computation on large-scale data. *IEEE Access*, 8:34585–34597, 2020.
8. Chi-Ning Chou, Kai-Min Chung, and Chi-Jen Lu. On the algorithmic power of spiking neural networks. In *10th Innovations in Theoretical Computer Science Conference (ITCS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

9 Graham Cormode, Mayur Datar, Piotr Indyk, and Shanmugavelayutham Muthukrishnan. Comparing data streams using hamming norms (how to zero in). *IEEE Transactions on Knowledge and Data Engineering*, 15(3):529–540, 2003.

10 Graham Cormode and Shan Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

11 Sanjoy Dasgupta, Charles F Stevens, and Saket Navlakha. A neural algorithm for a fundamental computing problem. *Science*, 358(6364):793–796, 2017.

12 Marianne Durand and Philippe Flajolet. Loglog counting of large cardinalities (extended abstract). In *Algorithms - ESA 2003, 11th Annual European Symposium, Budapest, Hungary, September 16-19, 2003, Proceedings*, pages 605–617, 2003.

13 Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In *DMTCS Proceedings vol. AH, 2007 Conference on Analysis of Algorithms (AofA 07)*, 2007. URL: `https://dmtcs.episciences.org/3545`.

14 Yael Hitron, Nancy A. Lynch, Cameron Musco, and Merav Parter. Random sketching, clustering, and short-term memory in spiking neural networks. In *11th Innovations in Theoretical Computer Science Conference, ITCS 2020, January 12-14, 2020, Seattle, Washington, USA*, pages 23:1–23:31, 2020.

15 Yael Hitron and Merav Parter. Counting to ten with two fingers: Compressed counting with spiking neurons. In *27th Annual European Symposium on Algorithms, ESA 2019, September 9-11, 2019, Munich/Garching, Germany*, pages 57:1–57:17, 2019.

16 Yael Hitron, Merav Parter, and Gur Perri. The computational cost of asynchronous neural communication. In *11th Innovations in Theoretical Computer Science Conference, ITCS 2020, January 12-14, 2020, Seattle, Washington, USA*, pages 48:1–48:47, 2020.

17 Russell Impagliazzo and Avi Wigderson. $P = BPP$ if E requires exponential circuits: Derandomizing the XOR lemma. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on the Theory of Computing, El Paso, Texas, USA, May 4-6, 1997*, pages 220–229, 1997.

18 Piotr Indyk. Stable distributions, pseudorandom generators, embeddings, and data stream computation. *Journal of the ACM (JACM)*, 53(3):307–323, 2006.

19 Piotr Indyk and Eric Price. K-median clustering, model-based compressive sensing, and sparse recovery for earth mover distance. In *Proceedings of the forty-third annual ACM symposium on Theory of computing*, pages 627–636, 2011.

20 Piotr Indyk and David P. Woodruff. Tight lower bounds for the distinct elements problem. In *44th Symposium on Foundations of Computer Science (FOCS 2003), 11-14 October 2003, Cambridge, MA, USA, Proceedings*, pages 283–288, 2003.

21 John Kallaugher and Eric Price. Separations and equivalences between turnstile streaming and linear sketching. In *Symposium on Theory of Computing, STOC 2020*, 2020.

22 Daniel M. Kane, Jelani Nelson, and David P. Woodruff. An optimal algorithm for the distinct elements problem. In *Proceedings of the Twenty-Ninth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2010, June 6-11, 2010, Indianapolis, Indiana, USA*, pages 41–52, 2010. `doi:10.1145/1807085.1807094`.

23 Michael Kapralov, Aida Mousavifar, Cameron Musco, Christopher Musco, Navid Nouri, Aaron Sidford, and Jakab Tardos. Fast and space efficient spectral sparsification in dynamic streams. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1814–1833. SIAM, 2020.

24 Zohar Karnin, Kevin Lang, and Edo Liberty. Optimal quantile approximation in streams. In *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 71–78. IEEE, 2016.

25 Jun Haeng Lee, Tobi Delbruck, and Michael Pfeiffer. Training deep spiking neural networks using backpropagation. *Frontiers in Neuroscience*, 10:508, 2016.

26 Robert A. Legenstein, Wolfgang Maass, Christos H. Papadimitriou, and Santosh S. Vempala. Long term memory and the densest k-subgraph problem. In *9th Innovations in Theoretical*

Computer Science Conference, ITCS 2018, January 11-14, 2018, Cambridge, MA, USA, pages 57:1–57:15, 2018.

27 Yi Li, Huy L. Nguyen, and David P. Woodruff. Turnstile streaming algorithms might as well be linear sketches. In *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014*, pages 174–183, 2014. doi:10.1145/2591796.2591812.

28 Nancy Lynch, Cameron Musco, and Merav Parter. Computational tradeoffs in biological neural networks: Self-stabilizing winner-take-all networks. *Innovations in Theoretical Computer Science*, 2017.

29 Nancy Lynch, Cameron Musco, and Merav Parter. Spiking neural networks: An algorithmic perspective. In *5th Workshop on Biological Distributed Algorithms (BDA 2017)*, 2017.

30 Nancy Lynch and Mien Brabeeba Wang. Integrating temporal information to spatial information in a neural circuit. *arXiv preprint arXiv:1903.01217*, 2019.

31 Nancy A. Lynch, Cameron Musco, and Merav Parter. Neuro-ram unit with applications to similarity testing and compression in spiking neural networks. In *31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria*, pages 33:1–33:16, 2017.

32 Wolfgang Maass. On the computational power of noisy spiking neurons. In *Advances in neural information processing systems*, pages 211–217, 1996.

33 Wolfgang Maass. Networks of spiking neurons: the third generation of neural network models. *Neural networks*, 10(9):1659–1671, 1997.

34 Wolfgang Maass. On the computational power of winner-take-all. *Neural computation*, 12(11):2519–2535, 2000.

35 Wolfgang Maass, Christos H. Papadimitriou, Santosh S. Vempala, and Robert A. Legenstein. Brain computation: A computer science perspective. In *Computing and Software Science - State of the Art and Perspectives*, pages 184–199. Springer, 2019. doi:10.1007/978-3-319-91908-9_11.

36 Gurmeet Singh Manku, Sridhar Rajagopalan, and Bruce G Lindsay. Approximate medians and other quantiles in one pass and with limited memory. *ACM SIGMOD Record*, 27(2):426–435, 1998.

37 J Ian Munro and Mike S Paterson. Selection and sorting with limited storage. *Theoretical computer science*, 12(3):315–323, 1980.

38 Shanmugavelayutham Muthukrishnan. *Data streams: Algorithms and applications*. Now Publishers Inc, 2005.

39 Noam Nisan and Avi Wigderson. Hardness vs randomness. *J. Comput. Syst. Sci.*, 49(2):149–167, 1994.

40 Christos H. Papadimitriou and Santosh S. Vempala. Random projection in the brain and computation with assemblies of neurons. In *10th Innovations in Theoretical Computer Science Conference, ITCS 2019, January 10-12, 2019, San Diego, California, USA*, pages 57:1–57:19, 2019.

41 Lili Su, Chia-Jung Chang, and Nancy Lynch. Spike-based winner-take-all computation: Fundamental limits and order-optimal circuits. *Neural Computation*, 31(12):2523–2561, 2019.

42 Amirhossein Tavanaei, Masoud Ghodrati, Saeed Reza Kheradpisheh, Timothée Masquelier, and Anthony Maida. Deep learning in spiking neural networks. *Neural Networks*, 111:47–63, 2019.

43 Leslie G. Valiant. Capacity of neural networks for lifelong learning of composable tasks. In *58th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2017, Berkeley, CA, USA, October 15-17, 2017*, pages 367–378, 2017.

44 David P. Woodruff. Optimal space lower bounds for all frequency moments. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2004, New Orleans, Louisiana, USA, January 11-14, 2004*, pages 167–175, 2004.

Communication Efficient Self-Stabilizing Leader Election

Xavier Défago
Tokyo Institute of Technology, Japan
defago@c.titech.ac.jp

Yuval Emek
Technion – Israel Institute of Technology, Haifa, Israel
yemek@technion.ac.il

Shay Kutten
Technion – Israel Institute of Technology, Haifa, Israel
kutten@ie.technion.ac.il

Toshimitsu Masuzawa
Osaka University, Japan
masuzawa@ist.osaka-u.ac.jp

Yasumasa Tamura
Tokyo Institute of Technology, Japan
tamura@c.titech.ac.jp

Abstract

This paper presents a randomized self-stabilizing algorithm that elects a leader r in a general n-node undirected graph and constructs a spanning tree T rooted at r. The algorithm works under the synchronous message passing network model, assuming that the nodes know a linear upper bound on n and that each edge has a unique ID known to both its endpoints (or, alternatively, assuming the KT_1 model). The highlight of this algorithm is its superior communication efficiency: It is guaranteed to send a total of $\tilde{O}(n)$ messages, each of constant size, till stabilization, while stabilizing in $\tilde{O}(n)$ rounds, in expectation and with high probability. After stabilization, the algorithm sends at most one constant size message per round while communicating only over the $(n-1)$ edges of T. In all these aspects, the communication overhead of the new algorithm is far smaller than that of the existing (mostly deterministic) self-stabilizing leader election algorithms.

The algorithm is relatively simple and relies mostly on known modules that are common in the fault free leader election literature; these modules are enhanced in various subtle ways in order to assemble them into a communication efficient self-stabilizing algorithm.

2012 ACM Subject Classification Computer systems organization → Fault-tolerant network topologies

Keywords and phrases self-stabilization, leader election, communication overhead

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.11

Related Version A full version that contains all missing proofs as well as additional discussions is available at `http://arxiv.org/abs/2008.04252` [44].

Funding The work of X. Défago, T. Masuzawa, and Y. Tamura was supported by JST SICORP Grant Number JPMJSC1606. The work of Y. Emek and S. Kutten was supported by the Israeli Ministry of Science and Technology (MOST) grant number 3-13565 and by the Technion Hiroshi Fujiwara Cyber Security Research Center and the Israel National Cyber Directorate.
Toshimitsu Masuzawa: The work of T. Masuzawa was supported by JSPS KAKENHI Grant Number 19H04085.

Acknowledgements We are grateful to Valerie King for helpful discussions.

© Xavier Défago, Yuval Emek, Shay Kutten, Toshimitsu Masuzawa, and Yasumasa Tamura;
licensed under Creative Commons License CC-BY

1 Introduction

The *leader election* problem has been recognized early as canonical in capturing the unique characteristics of distributed systems [67, 47, 70, 10]. Together with related problems, such as *spanning tree* construction and *broadcast*, it has been extensively studied through the years in various contexts including mobile networks [71, 81], key distribution [32], routing coordination [78], sensor control [52], general control [51], Paxos and its practical applications [65, 66, 24, 69], peer-to-peer networks [60] and more. The study of efficient algorithms for leader election and its related problems still draws plenty of attention in the present, see, e.g., [59, 76, 72, 48, 45].

A central efficiency measure in this regard is the *communication overhead* that the algorithm adds to the system, in terms of both the number of messages and their size. This has been a topic of interest from the early days, e.g., in [37, 47, 61, 46, 11, 3], to the more recent literature [59, 48, 45], in works of theoretical nature as well as in practically motivated ones [66, 81, 60].

In the realm of *self-stabilizing* algorithms [36] (see [38, 8] for textbooks), communication efficiency is a little bit trickier. Starting from an adversarially chosen initial configuration, the algorithm sends a certain number of messages until it stabilizes; after stabilization, the algorithm is required to keep sending messages indefinitely for the purpose of fault detection (see, e.e., [17]). Consequently, complexity measures related to the algorithm's communication overhead in the self-stabilization literature are divided into two "schools", depending on whether they focus on the messages sent post-stabilization or pre-stabilization.

The motivation behind bounding the communication overhead after the algorithm stabilizes comes from the assumption that faults are relatively rare and most of the time, the system is in a correct configuration. Here, a natural measure is the algorithm's *stabilization bandwidth*, defined in the influential paper of Awerbuch and Varghese [17] as the worst case number of messages sent during any time window of length τ after the algorithm has stabilized, where τ is the algorithm's stabilization time.[1] This measure is justified by the observation that if faults occurred after the algorithm was supposed to have stabilized, then the algorithm must recover from them also in τ time, hence a time window of length τ inherently encapsulates a "full cycle" of the operations executed by the algorithm post-stabilization.

The pre-stabilization approach is motivated by the realization that faulty configurations may lead to bursts of heavy communication, risking an overload of the system's communication components. The natural measure in this regard counts the number of messages sent until the algorithm stabilizes, starting from a worst case initial configuration (see, e.g., [63]).

In this paper, we develop a randomized self-stabilizing algorithm that elects a leader r and constructs a spanning tree T rooted at r in a general n-node (undirected) communication graph, assuming the *synchronous* KT_1 model of [13] (or a slightly weaker version thereof, see Sec. 1.1). Using constant size messages, the algorithm is guaranteed to stabilize in $\tilde{O}(n)$ rounds, while sending $\tilde{O}(n)$ messages, in expectation and whp.[2 3] The algorithm's stabilization bandwidth is also $\tilde{O}(n)$ as it is guaranteed to send at most one (constant size) message per round after stabilization. Another appealing feature is that after stabilization, the algorithm's communication is restricted to the edges of T.

[1] In [17], this notion is measured per edge, dividing the expression defined in the current paper by the number of edges.

[2] The asymptotic notation $\tilde{O}(x)$ hides polylog(x) factors. Refer to Thm. 2 and 3 for more accurate (asymptotic) bounds.

[3] An event A occurs whp (with high probability) if $\mathbb{P}(A) \geq 1 - n^{-c}$ for an arbitrarily large constant c.

The communication overhead of the new algorithm significantly improves upon the state-of-the-art for self-stabilizing leader election algorithms in general graphs with respect to the aforementioned complexity measures: To the best of our knowledge, no algorithm in the existing literature gets below the $\Omega(m)$ bound for neither the number of messages sent before stabilization, nor the stabilization bandwidth, where m is the number of edges in the graph. In fact, the algorithm's communication efficiency matches (up to logarithmic factors) the state-of-the-art for leader election also in the fault free setting in terms of both the number of messages and the number of bits sent before stabilization.

1.1 Model and Problem

Consider a communication network represented as a simple undirected graph $G = (V, E)$ whose nodes are identified with processing units that may exchange messages of constant size with their neighbors. The execution progresses in synchronous *rounds*, where round $t \in \mathbb{Z}_{\geq 0}$ starts at time t. In each round t, node $v \in V$ (1) receives the messages sent to it over its incident edges in round $t - 1$ (if any); (2) performs local computation; and (3) sends messages over a subset of its incident edges.

Let $n = |V|$. Each edge $e \in E$ admits a unique ID, represented as a bit string of size $O(\log n)$, that is known to both endpoints of e (a slightly weaker assumption than that of the KT_1 model [13]). The nodes also know a linear upper bound N on n that is assumed to be a sufficiently large power of 2.

The goal in the *leader election* problem is to reach a configuration where exactly one node $r \in V$ is marked as a leader (each node $u \in V - \{r\}$ knows that it is not the leader). In this paper, we also require that the nodes maintain a spanning tree of G rooted at the leader r.

We wish to develop a *self-stabilizing* leader election algorithm, where the initial configuration, at time 0, is determined by a malicious adversary that knows the algorithm's code but is oblivious to its random coin tosses (if any). When determining the initial configuration, the adversary may set all variables maintained by node $v \in V$, including its local clock (if such a variable is maintained by v) and incoming messages, with the exception of the variables that store the IDs of v's incident edges and the upper bound N on n.

1.2 Additional Related Work and Discussion

The shortage of work on communication overhead in the context of self-stabilization may have resulted from the fact that any non-trivial self-stabilizing system must send messages infinitely often. This makes its "message complexity" (as defined for non-self-stabilizing systems) infinite [17], giving rise to multiple competing communication measures. Luckily, the current algorithm is efficient in all the suggested measures.

Another possible explanation is that the known self-stabilizing leader election algorithms were not developed from the efficient non-self-stabilizing leader election techniques that preceded them, e.g., they are not derived from the seminal work of Gallager et al. [47]. This is probably because turning a sophisticated algorithm, designed for a fault free environment, into a self-stabilizing one is often a complex task which is prone to mistakes. Hence, less involved, though communication wasteful, methods were the common choice for the design of self-stabilizing algorithms.[4] This contrasts the algorithm developed in the current paper, assembled mainly from modules that are common in the fault free leader election literature.

[4] In explaining how to design an efficient fault tolerant system such as an efficient implementation of Paxos, Lampson writes that "More efficiency means more complicated invariants" and quotes Dijkstra as saying that "An efficient program is an exercise in logical brinkmanship" [66].

Early research on self-stabilization concentrated initially on whether a self-stabilizing algorithm is at all possible for a given task [36, 2, 42, 58, 5] and the required memory size (see, e.g., [36, 5, 39]). The main emphasis later has been on reducing the time complexity in various forms, such as the number of rounds, the asynchronous time, or the number of steps.

In addition to the aforementioned measures for the communication overhead of a self-stabilizing algorithm, it was suggested to measure self-stabilization (and also the related weak detectors) communication efficiency by the number of links over which the local checking is performed indefinitely after stabilization, or the maximum number of such links per node [6, 33, 35, 75, 74, 30]. For leader election, the optimum is $n-1$ links [68, 33]. We note that the algorithm presented in the current paper exactly matches this optimal bound.

Reducing communication overhead is also mentioned as a motivation for reducing the number of steps before stabilization, e.g., in [28], where it is also argued that the number of steps in the shared memory model is closely related to the amount of communication needed to implement that model. The step complexity has been addressed in other papers as well, e.g., [36, 34, 26].

Reducing the communication overhead is also stated as a primary advantage of silent self stabilizing algorithms [39]. In algorithms where neighbors exchange the description of their states periodically (e.g., when using local checking), reducing the state size translates to a reduced message size. Approaches for reducing the message size for detection (though not necessarily the number of messages) by sending only some compressed versions of the state were studied in [41, 77]. The number of messages in certain algorithms with a single starter (irrelevant to the leader election task) is reduced in [43], but is still $\Omega(n^2)$ until stabilization and $\Omega(n)$ per time unit after stabilization, when applied in synchronous networks.

The tasks of leader election and tree constructions are common building blocks in numerous algorithms and practical systems such as mutual exclusion [67], handling various race conditions and topology dependent tasks [47, 24], ensuring that the components in distributed applications remain consistent [64], implementing databases and data centers [55], locks [22], file servers [49, 25], broadcast and multicast [79, 23], and topology update and virtually every global task [12]. In the context of self-stabilization, given a spanning tree, it is possible to apply a self-stabilizing reset, and to use the reset as a module of a transformer that can transform non-self-stabilizing algorithms to be self-stabilizing [9, 17, 15, 16, 5]. The task of converting leader election algorithms themselves to be self-stabilizing has not enjoyed the help of such transformers, since most transformers use a leader and/or a spanning tree.

Numerous self-stabilizing leader election algorithms appeared in the literature; we mention here only a few [4, 9, 17, 42, 14, 40, 1, 53, 21, 19, 31, 7, 62, 20]. Non constant space is needed for leader election [18] by a deterministic protocol even under a centralized daemon if the nodes identities (IDs) are not bounded. Logarithmic space is needed if the algorithm is silent [39].

The algorithm presented in the current paper utilizes a token circulation and timeouts. Self-stabilizing token circulation algorithms in general networks have been treated in numerous papers, e.g., [54, 57, 56, 29, 80]. Multiple papers deal with the related problem of a self-stabilizing construction of a depth first search tree, starting with [27]. Some self-stabilizing tree construction algorithms are implemented on top of tokens (or similar messages that are logically sent to nodes that are not immediate neighbors), e.g., [53, 19]. Timeouts are commonly used in distributed computing, including in leader election algorithms, e.g., [46, 53, 19].

The KT_1 model is advocated in [13] as being more realistic than KT_0, where a node only knows its ports, but not the node connected to each port. In recent years, several non-self-stabilizing algorithms were proposed to explore the properties of KT_1 in order to reduce the message complexity [59, 48, 50, 72, 73].

1.3 Paper's Organization

The rest of the paper is organized as follows. Following some notation and terminology defined in Sec. 2, the algorithm is described in Sec. 3, starting with an informal overview (Sec. 3.1) that includes a discussion of the main technical ideas. Sec. 4 presents a sketch of the algorithm's analysis, including its fault recovery guarantees, stabilization run-time, and message complexity; refer to [44] for the complete analysis.

2 Preliminaries

Throughout this paper, it is assumed that (directed and undirected) graphs may include self loops, but not parallel edges. We denote the node set and edge set of a (directed or undirected) graph G by $\mathcal{V}(G)$ and $\mathcal{E}(G)$, respectively.

Consider an undirected graph G. A subgraph T of G that admits a tree topology is called a *subtree* of G. Two node disjoint subtrees T and T' of G are said to be *adjacent* if there exists an edge $e = \{v, v'\} \in \mathcal{E}(G)$ such that $v \in \mathcal{V}(T)$ and $v' \in \mathcal{V}(T')$; such an edge e is referred to as a *crossing* edge. A *merger* of the subtrees T and T' (over the crossing edge e) forms a new subtree of G whose node set is $\mathcal{V}(T) \cup \mathcal{V}(T')$ and whose edge set is $\mathcal{E}(T) \cup \mathcal{E}(T') \cup \{e\}$.

Consider a directed graph D. The *undirected version* of D is the undirected graph obtained from D by ignoring the edge directions. We say that D is *weakly connected* if the undirected version of D is connected (note the distinction from the notion of a strongly connected directed graph).

The directed graph D is said to be a *pseudoforest* if the outdegree of every node $v \in \mathcal{V}(D)$ is at most 1. A *pseudotree* is a weakly connected pseudoforest. The undirected versions of a pseudoforest and a pseudotree are referred to as an *undirected pseudoforest* and an *undirected pseudotree*, respectively.

We subsequently reserve the notation G for the n-node undirected communication graph and denote $V = \mathcal{V}(G)$ and $E = \mathcal{E}(G)$. For a node $v \in V$, its neighbor set in G is denoted by $N(v) = \{u \in V \mid \{u, v\} \in E\}$.

3 The Algorithm

3.1 An Informal Overview

In this section, we provide an overview of our algorithm, composed of various modules that the informed reader will recognize from the vast literature on leader election and spanning tree construction in fault free environments. The algorithm maintains a partition of the nodes into rooted trees, defined by means of variables $v.\texttt{prnt}$ and $v.\texttt{chld}$ in which each node v stores its tree parent and children, respectively. The actions of each tree T are coordinated by its root, namely, the unique node $r \in \mathcal{V}(T)$ satisfying $r.\texttt{prnt} = \perp$. The goal is to repeatedly merge the trees over crossing edges until a single tree that spans the whole graph remains.

The algorithm's main module, denoted by `Main`, divides the execution into *phases*, where in each phase the root r of tree T decides at random between two modules, denoted by `Propose` and `Accept`, to be invoked in the current phase. The role of `Propose` is to find a crossing edge over which a merger proposal is sent. To this end, r invokes a randomized module, denoted by `Cross`, that implements procedure *FindAny* of [59]. If `Cross` fails to find a crossing edge, then the phase ends.

Otherwise, an edge $\{x, y\}$, that crosses between $\mathcal{V}(T) \ni x$ and $V - \mathcal{V}(T) \ni y$, is returned. Then, r invokes a module, denoted by `Trnsfer`, whose role is to update the `prnt` and `chld` variables along the unique simple (r, x)-path in T so that T is re-rooted at x (cf. [47]). Following that, x sends a merger proposal to y and waits silently for a reply. If such a reply does not arrive within the next $\Theta(N)$ rounds, then the phase ends and x, now being the root, initiates a new phase. Otherwise, x becomes the child of y, merging T into y's tree.

The role of `Accept` is to accept incoming merger proposals. In particular, a merger proposal sent from a node x to a node y in tree T' is accepted by y, sending a reply to x, if and only if the current phase of T' ("as far as y knows") is an `Accept` phase. This ensures that the resulting structure is cycle free.

The aforementioned process is implemented on top of a module, denoted by `Traverse`, that implements a depth first search traversal of the tree by a conceptual *token.* The executions of both `Cross` and `Accept` are divided into $\Theta(\log N)$ *epochs* so that `Traverse` is invoked by the root r at the beginning of each epoch. The epochs lasts for a fixed $\Theta(N)$ number of rounds, chosen to guarantee that the traversal can be safely completed, returning the token to r where it is stored until the next epoch begins. Apart from `Traverse` that controls the token's mobility, `Trnsfer` also shifts the token down the root transfer path, thus ensuring that outside the scope of the traversals handled by `Traverse`, the token is always stored at the root.

A key property of the algorithm, that facilitates its communication efficiency, is that a node may send a message only when it holds the token (and then, it may send at most one message per round). In particular, `Cross` implements procedure *FindAny* [59] on top of the tree traversals, so that each one of its $\Theta(\log N)$ epochs is responsible for one broadcast-echo process in the tree (carrying $O(1)$ bits of information). Moreover, a merger proposal sent from node x to node y is recorded at y (assuming that y's tree is in an `Accept` phase) until y holds the token as part of a traversal of its tree; the proposal is then processed and y sends a reply to x. The token held by x prior to the merger is dissolved when x's tree is merged into y's, striving for the invariant that each tree holds a single token.

The fact that each tree is repeatedly traversed by its token is exploited by the algorithm's fault detection mechanism: Each node v maintains a v.`tmr` variable that is reset when v receives the token from its parent as part of a traversal and is incremented in every round otherwise; if v.`tmr` exceeds a predetermined $\Theta(N)$ threshold, then v invokes a procedure named `Restart` that resets all its variables, so that v forms a new singleton tree, and generates a fresh token at v. A malformed tree is (implicitly) detected by the token being lost when it is sent to v from an adjacent node u while v does not "expect" to receive a token from u. We emphasize that in both events (v experiencing a restart and v ignoring u's message), v does not inform any of its neighbors of the detected fault.

This fault detection mechanism essentially replaces the local checking module, common to many self-stabilizing algorithms: rather than checking "all neighbors all the time", node v checks only the tokens it receives, which is clearly more efficient communication-wise. Our algorithm's self-stabilization guarantees follow from this mechanism with some additional fine points explained in Sec. 3.2.

3.2 A Detailed Description

Our algorithm performs leader election by constructing a rooted spanning tree of G. To this end, each node $v \in V$ maintains a variable v.`prnt` $\in N(v) \cup \{\perp\}$ that points to its tree parent and a variable v.`chld` $\in (N(v))^*$ that stores a list of pointers to its tree children; by a slight abuse of notation, we may address v.`chld` as a subset of $N(v)$ when the order of its elements is not important.

We say that v is a *root* if v.prnt $= \bot$. Let v.T_nbrs $= v$.chld if v is a root; and v.T_nbrs $= v$.chld $\cup \{v$.prnt$\}$ if v is not a root. We refer to the nodes in v.T_nbrs as the *tree neighbors* of v (from the perspective of v).

For the sake of simplifying the algorithm's description, we augment the communication graph with a virtual *shadow* node $\tilde{v}$ for each node v of the original graph so that $\tilde{v}$ has a single incident edge connecting it to v. The algorithm is designed so that v is always the parent of $\tilde{v}$ which guarantees, in particular, that the set of tree neighbors of a node is never empty. To distinguish the original graph nodes from their virtual shadows we refer to the former as *physical* nodes. The operation of the virtual shadow node $\tilde{v}$ is simulated by its corresponding physical node v; this simulation is straightforward as v is the only neighbor of $\tilde{v}$. We assume hereafter that the node set V of the communication graph $G = (V, E)$ contains both the physical and shadow nodes.

The aforementioned modules Traverse, Main Propose, Accept, Cross, and Trnsfer are presented in Sec. 3.2.1–3.2.6, respectively, while Restart is presented in Sec. 3.2.7. To help the reader put things into context, some parts of the algorithm's description are written as if no "recent faults" has occurred; we emphasize that the correctness of the algorithm does not rely on any such assumption. Moreover, to clarify the algorithm's presentation, we restrict our attention to the less trivial components, leaving out some details that the reader can easily complete by themselves; for ease of reference, Table 1 summarizes the variables maintained by these components.

Table 1 The variables of node $v \in V$.

Variable	Range	Semantics
prnt	$N(v) \cup \{\bot\}$	v's parent
chld	$(N(v))^*$	v's children
T_nbrs	$2^{N(v)}$	v's tree neighbors
tkn	$\{0, 1\}$	indicates that v holds a token
tkn_d	T_nbrs	the direction of the token
recent	$\{0, 1\}$	indicates that v has passed a token in the previous round
tmr	$\mathbb{Z}_{\geq 0}$	#rounds since receiving a (hot) token
out_prop	$N(v) \cup \{\bot\}$	the direction of the (recently found) crossing edge
in_prop(u)	$\{0, 1\}$	indicates that a proposal from $u \in N(v)$ has been registered

3.2.1 Module Traverse

Consider some root node r and let T be r's tree. The Traverse module implements a distributed process in which a conceptual *token* is passed from node to node, using designated pass_tkn messages, to form a depth first search traversal of T.

To monitor the token distribution, each node $v \in V$ maintains three variables: (a) v.tkn $\in \{0, 1\}$ that indicates whether v holds a token; (b) v.tkn_d $\in v$.T_nbrs that points to the last tree neighbor to which v has passed a token; and (c) v.recent $\in \{0, 1\}$ that indicates whether v has passed a token in the previous round.

Denoting $d = |v$.T_nbrs$|$, node v also employs a permutation (i.e., a bijection) π_v : $\{0, 1, \ldots, d-1\} \to v$.T_nbrs defined so that $\pi_v(i-1)$ points to the i-th element in v.chld for $1 \leq i \leq |v$.chld$|$ and $\pi_v(d-1)$ points to v.prnt if v.prnt $\neq \bot$. This permutation naturally induces a periodic sequence over the nodes in v.T_nbrs so that the i-th element of this sequence is $\pi_v(i \bmod d)$ for every $i \in \mathbb{Z}_{>0}$, referring to $\pi_v(i + 1 \bmod d)$ as its π_v-*successor*. If v is a physical node, which ensures that v.chld $\neq \emptyset$, then we refer to $\pi_v(0)$ and to $\pi_v(|v$.chld$| - 1)$ as v's *first child* and *last child*, respectively.

Consider some non-root node v. Upon receiving a pass_tkn message M from node $u \in N(v)$, node v verifies that (i) $v.\texttt{tkn} = 0$; (ii) $v.\texttt{recent} = 0$; (iii) $v.\texttt{tkn_d} = u$; and (iv) its parent-child relations with u are consistent between the two nodes, that is, either $u = v.\texttt{prnt}$ and M indicates that u is the parent of v or $u \in v.\texttt{chld}$ and M indicates that u is a child of v. If any of these four conditions is not satisfied, then v ignores M altogether, thus causing the token it carries to (conceptually) disappear.

Assuming that the four conditions are satisfied, node v sets $v.\texttt{tkn} \leftarrow 1$, thus indicating that it now holds the token that conceptually arrived with message M. Following that, v holds the token for 1–4 full rounds, where the exact number is determined by the modules implemented on top of Traverse. Then, v passes the token to the π_v-successor u' of $u = v.\texttt{tkn_d}$ by sending to it a pass_tkn message and sets (1) $v.\texttt{tkn} \leftarrow 0$; (2) $v.\texttt{tkn_d} \leftarrow u'$; and (3) $v.\texttt{recent} \leftarrow 1$. The sole purpose of the flag $v.\texttt{recent}$ is to ensure that v holds no token for (at least) one full round before it can accept a token again. This flag is always turned off one round after it has been turned on; that is, v resets $v.\texttt{recent} \leftarrow 0$ in each round, after the value of $v.\texttt{recent}$ has been read as part of processing the incoming messages.

The operation of the root r (which is always a physical node) in Traverse is identical to that of any non-root node except that r is also responsible for initiating the traversal, by passing the token to its first child, and terminating the traversal, after receiving the token from its last child, holding the token in between traversals. To distinguish the situation where the token is held by a (root or non-root) node in the midst of a traversal from the situation in which the token is held by a (root) node between traversals, we refer to the token in the former situation as *hot* and in the latter situation as *cold*. The traversal process is initiated at r upon receiving a designated signal from the modules implemented on top of Traverse. If r does not hold a cold token when this signal is received, then Restart is invoked; otherwise, the token becomes hot and a traversal of T is initiated.

Consider some traversal of T. We refer to the pass_tkn message received (resp., sent) by a non-root node v from (resp., to) its parent as v's *discovery message* (resp., *retraction message*) and to the round in which this message is received (resp., sent) as v's *discovery round* (resp., *retraction round*). The *discovery round* (resp., *retraction round*) of T's root r is defined to be the round in which the traversal is initiated (resp., terminated), i.e., the round in which the token held by r turns from cold to hot (resp., from hot to cold). A key observation is that the traversal allows r to simulate a broadcast-echo process over its tree T by piggybacking the content of the broadcast (resp., echo) messages over the discovery (resp., retraction) messages.

Traverse is responsible also for a timer mechanism monitored by variable $v.\texttt{tmr} \in \mathbb{Z}_{\geq 0}$ that each node $v \in V$ maintains. This variable is incremented by v in every round. During a discovery round, v verifies that $v.\texttt{tmr} \geq C_{\text{tr}} N$, where C_{tr} is a positive constant whose value is determined in Sec. 4. If this condition is not satisfied, an event referred to as a *premature discovery*, then v invokes Restart; otherwise, v resets $v.\texttt{tmr} \leftarrow 0$. If, at any stage of the execution, variable $v.\texttt{tmr}$ exceeds the $8C_{\text{tr}}N$ threshold, then v also invokes Restart; this event is referred to as an *expiration* of $v.\texttt{tmr}$.

3.2.2 Module Main

The execution of Main is partitioned into *phases* that are further partitioned into *epochs*; these partitions are orchestrated by the root nodes r. Each phase is dedicated to an invocation of either Propose or Accept; the root r decides between the two at the beginning of the phase by tossing a fair coin.

A Propose phase consists of $C_{\text{ph}} \log N$ *search* epochs, where $C_{\text{ph}} \in \mathbb{Z}_{>0}$ is a constant to be determined in the sequel, followed by a *root transfer* epoch, followed by a *proposing* epoch. The search epochs are dedicated to an invocation of Cross and based on its outcome, the root transfer epoch is either empty, in which case, the proposing epoch is also empty, or dedicated to an invocation of Trnsfer. An Accept phase consists of $C_{\text{ph}} \log N + 2$ *accepting* epochs.

Consider a tree T rooted at r. Each search and accepting epoch is dedicated to an invocation of Traverse, initiated by r, during which the (hot) token completes one traversal of T and returns to r, where it is held (cold) until the epoch ends. These epochs have a fixed length of $2C_{\text{tr}}N$ rounds. The length of the root transfer and proposing epochs may vary, however, it is guaranteed that their combined length is at most $4C_{\text{tr}}N$ rounds. Therefore, the Accept phases last for exactly $(C_{\text{ph}} \log N + 2) \cdot 2C_{\text{tr}}N$ rounds, whereas the length of each Propose phase is at least $C_{\text{ph}} \log N \cdot 2C_{\text{tr}}N$ and at most $(C_{\text{ph}} \log N + 2) \cdot 2C_{\text{tr}}N$ rounds.

A variable that plays an important role in the distinction between Accept and Propose phases is v.accpt $\in \{0, 1\}$, maintained by each node $v \in \mathcal{V}(T)$, that indicates weather v is available for merger proposals (more on that soon). Node v turns this variable on (if it is not on already) in the retraction round of any traversal associated with an accepting epoch (i.e., during Accept phases); it turns this variable off (if it is not off already) in the discovery round of any traversal associated with a search epoch (i.e., during Propose phases). This means that the accpt variables are turned on in a bottom up order and turned off in a top down order. Notice that once the variable v.accpt is on (resp., off), it stays on (resp., off) at least until the end of the current Accept (resp., Propose) phase.

3.2.3 Module Propose

Consider some tree T rooted at node r. Propose controls the algorithm's operation during Propose phases orchestrated by r. Specifically, during the $C_{\text{ph}} \log N$ search epochs of a Propose phase, r orchestrates the search for an edge that crosses from T to an (arbitrary) adjacent tree. This is done by invoking Cross that is guaranteed to return a crossing edge with a positive constant probability if such an edge exists.

The output of Cross is returned by means of the variables v.out_prop $\in N(v) \cup \{\bot\}$ that each node $v \in \mathcal{V}(T)$ maintains. Assuming that no faults have occurred during the execution of Cross, it is guaranteed that these variables satisfy the following three properties: (OP1) If r.out_prop $= \bot$, then v.out_prop $= \bot$ for every node $v \in \mathcal{V}(T)$. (OP2) If r.out_prop $\in N(v)$, then there exists exactly one *crossing port* $x \in \mathcal{V}(T)$, namely, a node satisfying x.out_prop $\in N(x) - x$.T_nbrs. Taking P to be the unique simple (r, x)-path in T, the variable v.out_prop points to the successor of v along P for every node $v \in \mathcal{V}(P) - \{x\}$; and v.out_prop $= \bot$ for every node $v \in \mathcal{V}(T) - \mathcal{V}(P)$. (OP3) If x is a crossing port, then x.out_prop $= y \in V - \mathcal{V}(T)$. Refer to Figure 1 for an illustration.

If Cross does not find a crossing edge, indicated by r.out_prop $= \bot$, then the phase terminates (which means that the root transfer and proposing epochs are empty). Assuming that Cross returns the edge $\{x, y\}$ that connects the crossing port $x \in \mathcal{V}(T)$ with a node $y \in V - \mathcal{V}(T)$, the root transfer epoch is dedicated to modifying the parent-child relations in T so that the root role transfers from r to x; this is handled by Trnsfer.

When Trnsfer terminates, the tree T is rooted at x, that also holds the (cold) token, and x.out_prop $= y$. The proposing epoch is now dedicated to the attempt of x to merge T and y's tree T' over the crossing edge $e = \{x, y\}$. To this end, in the first round of the proposing epoch, x sends a propose message to y. Following that, x waits silently for a reply from y in the form of an accept message. If this reply does not arrive during the subsequent

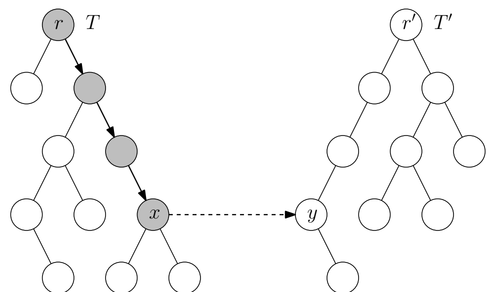


Figure 1 Towards a merger of the proposing tree T rooted at r and the accepting tree T' rooted at r' over the crossing edge $\{x, y\}$. The tree edges are depicted by the solid lines whereas the crossing edge is depicted by the dashed line. The nodes along the unique (r, x)-path in T are marked in gray and the values of their `out_prop` variables are depicted by the oriented edges.

$3C_{\mathrm{tr}}N - 1$ rounds, then x sets x.`out_prop` $\leftarrow \perp$ and the proposing epoch terminates (with no tree merger), also terminating this `Propose` phase (the next phase starts with x as the root of T).

If x receives an `accept` message from y before the end of the proposing epoch, then T is merged with T'. From the perspective of x, this merger is executed by setting (1) x.`prnt` $\leftarrow y$; (2) x.`tkn` $\leftarrow 0$; (3) x.`tkn_d` $\leftarrow y$; and (4) x.`out_prop` $\leftarrow \perp$. In particular, this means that x is no longer a root and that it expects the next `pass_tkn` message to arrive from its new parent y. Moreover, the token held by x is eliminated, an event referred to hereafter as *dissolving* x's token.

3.2.4 Module `Accept`

Consider some tree T' rooted at r'. `Accept` controls the algorithm's operation during `Accept` phases orchestrated by r'. The role of an `Accept` phase is to allow the nodes of T' to accept merger proposals of neighboring nodes in adjacent trees. To this end, each node $y \in \mathcal{V}(T')$ maintains the binary variable y.`in_prop`$(x) \in \{0, 1\}$ for every (graph) neighbor $x \in N(y) - y$.`T_nbrs`. This variable registers merger proposals received from x so that they can be processed when y is ready (as explained soon). Specifically, y.`in_prop`(x) is turned on when y receives a `propose` message while y.`accpt` $= 1$; it is turned off when y resets y.`accpt` $\leftarrow 0$. We emphasize that any `propose` message received while y.`accpt` $= 0$ is ignored. Moreover, if a node $x \in N(y)$ joins y.`T_nbrs`, then the y.`in_prop`(x) entry in y.`in_prop`$(\cdot)$ is eliminated.

An `Accept` phase consists of $C_{\mathrm{ph}} \log N + 2$ accepting epochs, each one of them begins by signaling r' to invoke `Traverse`, thus initiating a traversal process during which a (hot) token is passed through the nodes of T'; when the traversal terminates, the token is held (cold) at r' until the next epoch. A physical node $y \in \mathcal{V}(T')$ is said to be *retraction ready* if it holds a hot token while y.`tkn_d` $= \pi_y(|y$.`chld`$| - 1)$, indicating that the held token has been received from y's last child. The algorithm is designed so that y may respond to merger proposals of its neighbors only when it is retraction ready. Notice that a shadow node $\tilde{y}$ never receives merger proposals as its only neighbor y in G is always a tree neighbor of $\tilde{y}$.

Consider node y when it becomes retraction ready while y.`accpt` $= 1$ and let w be the (current) last child of y. Node y checks the content of $P_y = \{x \mid y$.`in_prop`$(x) = 1\}$; if $P_y = \emptyset$, then the role of y in the current accepting epoch is over and `Accept` signals `Traverse` that y can release the hot token (after holding it for one full round), which leads to y's retraction round.

If $P_y \neq \emptyset$, then y picks one neighbor $x \in P_y$ (arbitrarily) and sends to it an `accept` message. Then, in the subsequent round, the merger of T' and the tree T of x is executed, where on the T' side, y adds x as its last child in T' by appending x to the end of y.`chld`, turning it into the π_y-successor of (the original last child) w. Following that, `Accept` signals `Traverse` that y can release the hot token, continuing with the traversal of the updated T'; this results in passing the hot token to x as x is now the π_y-successor of $w = y$.`tkn_d`.

Observe that the tree merger is designed so that T is traversed by the token held by y immediately after it is merged into T'. The aforementioned mechanism that controls the `accpt` variables guarantees that v.`accpt` is turned on during this traversal for every node $v \in \mathcal{V}(T)$; this happens during v's retraction round which means that v will be able to process incoming merger proposals only during the next accepting epoch (if any). Node y continues processing merger proposals of other nodes when it gets the token back from x.

3.2.5 Module `Cross`

King et al. [59] developed a distributed procedure called *FindAny*, working in a fault free environment, that given a subtree T of G rooted at r, finds an edge that crosses between $\mathcal{V}(T)$ and $V - \mathcal{V}(T)$ with a positive constant probability if such an edge exists. The procedure is orchestrated by r using a constant number of broadcast-echo iterations over T with messages of size $O(\log n)$.[5]

We use a variant of the procedure of King et al., where each broadcast-echo iteration with $O(\log n)$ size messages is subdivided into $O(\log n)$ broadcast-echo iterations with constant size messages. `Cross` is then implemented using $O(\log n)$ iterations of the traversal process managed by `Traverse`, piggybacking the (constant size) messages of each broadcast-echo iteration over the `pass_tkn` messages of the corresponding traversal process (see Sec. 3.2.1). Each one of these $O(\log n)$ traversal iterations is executed in its own search epoch, where r is signaled to invoke `Traverse` at the beginning of the epoch. The value of C_{ph} is derived from the hidden constant in the O-notation.

Unfortunately, we cannot guarantee that properties (OP1)–(OP3) of the `out_prop` variables in T (see Sec. 3.2.3) hold if these variables sprout from the adversarially chosen initial configuration, rather than being generated by a complete invocation of `Cross` as part of the execution. However, we implement the module so that property (OP2) is guaranteed as long as the last search epoch, referred to hereafter as the *safety* epoch of `Cross`, is completed as part of the execution.

To this end, we implement `Cross` so that variable v.`out_prop` is set during v's retraction round of the traversal associated with the safety epoch for every node $v \in \mathcal{V}(T)$. Moreover, the retraction message of each child $u \in v$.`chld` of v specifies whether the number of crossing ports in u's subtree is zero, one, or at least two; node v then sets v.`out_prop` $\leftarrow u$ (in its own retraction round) if and only if u admits one crossing port in its subtree and every other child of v admits zero crossing ports in its subtree.

We emphasize that a fault free execution of the safety epoch alone does not guarantee properties (OP1) and (OP3); that is, it does not rule out a scenario where r.`out_prop` $= \perp$ and yet, v.`out_prop` $\neq \perp$ for some non-root nodes $v \in \mathcal{V}(T)$, nor does it rule out a scenario where r.`out_prop` $\neq \perp$, but the returned edge e connects the (unique) crossing port x to another node in T (both scenarios require the adversary's "involvement"). We show in the sequel that the latter scenario does not affect the correctness of our algorithm; to overcome

[5] The procedure needed for our purposes is a slightly simplified version of the one developed in [59].

the former scenario, every node v resets $v.\texttt{out_prop} \leftarrow \bot$ upon receiving a traversal discovery message of `Traverse`, that is, when v receives a `pass_tkn` message from $v.\texttt{prnt}$ (and does not ignore it). This is consistent with the fact that `Trnsfer` is implemented using `root_trns` messages, rather than `pass_tkn` messages.

3.2.6 Module `Trnsfer`

Given a tree T rooted at r and a crossing port x, `Trnsfer` modifies the parent-child relations in T so that it is re-rooted at x. When the module is invoked, the `out_prop` variables maintained by the nodes in T mark the unique simple (r, x)-path P in T (assuming that these variables were generated during the safety epoch of `Cross`, completed as part of the execution). `Trnsfer` sequentially shifts the root role from node to node down the path P, using `root_trns` messages that carry a cold token. This is done as follows.

Consider some intermediate node $v \in \mathcal{V}(P)$, as indicated by $v.\texttt{prnt} \neq \bot$ and $v.\texttt{out_prop} \in v.\texttt{chld}$. When v receives a `root_trns` message M from node $u \in N(v)$, it verifies that (i) $v.\texttt{tkn} = 0$; (ii) $v.\texttt{recent} = 0$; (iii) $v.\texttt{tkn_d} = u$; and (iv) $u = v.\texttt{prnt}$; if any of these four conditions is not satisfied, then v ignores M altogether, thus causing the token that M carries to (conceptually) disappear.

Assuming that the four conditions are satisfied, node v sets $v.\texttt{tkn} \leftarrow 1$, indicating that it now holds the (cold) token, and turns itself into the new root by setting $v.\texttt{prnt} \leftarrow \bot$ and appending u to $v.\texttt{chld}$. (Note that v does not reset variable $v.\texttt{tmr}$ upon receiving a cold token.) In the subsequent round, v sends a `root_trns` message to its successor $w = v.\texttt{out_prop}$ in P and sets (1) $v.\texttt{tkn} \leftarrow 0$; (2) $v.\texttt{tkn_d} \leftarrow w$; (3) $v.\texttt{out_prop} \leftarrow \bot$; and (4) $v.\texttt{recent} \leftarrow 1$; thus indicating that v no longer holds the token and that it expects the next `pass_tkn` message to arrive from w. It also turns itself into a child of w by setting $v.\texttt{prnt} \leftarrow w$ and $v.\texttt{chld} \leftarrow v.\texttt{chld} - \{w\}$; to ensure that the parent-child relations between v and w are updated in synchrony, these two operations are performed by v in the round subsequent to sending the `root_trns` message.

The original root r behaves just like an intermediate node except that its actions are triggered by the invocation of `Trnsfer`, rather than by receiving a `root_trns` message from its (non-existent) predecessor in P. The crossing port x also behaves just like an intermediate node except that it remains the (cold token holding) root and does not send a `root_trns` message to its (non-existent) successor in P; rather, it initiates a proposing epoch as explained earlier.

3.2.7 Procedure `Restart`

When `Restart` is invoked at a physical node $v \in V$ or at its shadow $\tilde{v}$, we say that both v and $\tilde{v}$ *experience* a restart. Upon invocation of the procedure at either of the two nodes, `Restart` disconnects v from all its tree neighbors other than $\tilde{v}$ and places v and $\tilde{v}$ in a new tree rooted at v that includes only these two nodes. This is implemented by setting $v.\texttt{prnt} \leftarrow \bot$, $\tilde{v}.\texttt{prnt} \leftarrow v$, $v.\texttt{chld} \leftarrow \{\tilde{v}\}$, and $\tilde{v}.\texttt{chld} \leftarrow \emptyset$. `Restart` assigns the new tree's (cold) token to v, setting $v.\texttt{tkn} \leftarrow 1$, $\tilde{v}.\texttt{tkn} \leftarrow 0$, and $v.\texttt{recent}, \tilde{v}.\texttt{recent} \leftarrow 0$, and consistently adjusts the `tkn_d` variables by setting $v.\texttt{tkn_d} \leftarrow \tilde{v}$ and $\tilde{v}.\texttt{tkn_d} \leftarrow v$. The `out_prop` and `in_prop` variables of v and $\tilde{v}$ are initialized, setting $v.\texttt{out_prop}, \tilde{v}.\texttt{out_prop} \leftarrow \bot$ and $v.\texttt{in_prop}(u) \leftarrow 0$ for every $u \in N(v) - \{\tilde{v}\}$ (recall that v is the only graph neighbor of $\tilde{v}$ and is also a tree neighbor of $\tilde{v}$, so $\tilde{v}.\texttt{in_prop}(\cdot)$ is "empty"), thus indicating that v and $\tilde{v}$ do not participate in any active root transfer path and that no tree merger proposals are currently registered at them. Finally, the `tmr` variables are set to $v.\texttt{tmr}, \tilde{v}.\texttt{tmr} \leftarrow C_{\text{tr}} N$, thus allowing the nodes to start a traversal (without experiencing another restart due to a premature discovery), and v initiates a new `Propose` phase.

On top of the fault detection mentioned earlier, `Restart` is also invoked whenever the node's internal variables are inconsistent with each other. This can happen only in the initial configuration.

4 Analysis

Notation and Terminology. We analyze the operation of the algorithm on the communication graph $G = (V, E)$ that contains both the physical nodes and their (virtual) shadow nodes (see Sec. 3). Recalling that a node can send a message only when it holds a token, the algorithm's proof of correctness relies on analyzing the token distribution across G and its dynamics over time. To this end, we regard the conceptual tokens as actual entities that are transitioned across the graph.

Throughout this section, we append a superscript t to the notation of our variables when referring to the value that these variables hold at time $t \in \mathbb{Z}_{\geq 0}$ so $v.\mathtt{dummy}^t$ denotes the value held by variable `dummy` of node $v \in V$ at time t. The superscript t is omitted when t is not important or clear from the context.

Consider a node $v \in V$ and suppose that $v.\mathtt{tkn}^t = 1$ which means that v holds a token κ at time t. This token may be *dispatched* in round t from v to a node $w \in v.\mathtt{T_nbrs}$ by means of a `pass_tkn` message if κ is hot; or a `root_trns` message if κ is cold. We say that the dispatch *fails* and that κ *dies* if this message is ignored by w (in round $t + 1$); otherwise, we say that the dispatch *succeeds* and that κ is *acquired* by w. Notice that in the latter case, we regard the dispatch as successful, saying that w acquires κ, even if w experiences a restart in round $t + 1$ ("after" acquiring the token).

Node $v \in V$ is said to *own* a token at time $t > 0$ if either (1) $v.\mathtt{tkn}^t = 1$; or (2) $v.\mathtt{tkn}^t = 0$ and $v.\mathtt{recent}^t = 1$. In other words, v owns a token at time t if it holds a token at time t or if it has just dispatched a token to a tree neighbor u in round $t - 1$ (which means that v no longer holds it at the beginning of round t). Using this convention, we ensure that if the dispatch of the token from v to u is successful, then the token is delivered smoothly from v to u so that it is owned contiguously by exactly one of the two nodes (avoiding an ownership gap at time t). We say that a token κ is *alive* at time $t > 0$ if it is owned by some node at that time.

When a physical node v and its shadow node $\tilde{v}$ experience a restart, a new token κ is generated at v (and immediately dispatched to $\tilde{v}$). We emphasize that if v or $\tilde{v}$ hold a token κ' in round t prior to the invocation of `Restart`, then we think of κ' as disappearing and regard κ as a new token; this is another case where we say that κ' *dies* in round t.

Edge $e = \{u, v\} \in E$ is said to be *compatible* at time t if $u = v.\mathtt{prnt}^t$ and $v \in u.\mathtt{chld}^t$ (or vice versa). Let $G_c^t = (V, E_c^t)$ be the undirected graph defined by setting $E_c^t \subseteq E$ to be the subset of edges compatible at time t. The (maximal) connected components of G_c^t are referred to as *compatible components*. By definition, G_c^t is an undirected pseudoforest and the compatible components are undirected pseudotrees.

Edge $e = \{v, w\} \in E$ is said to be a *dangling* edge of node v at time t if v regards it as a tree edge, i.e., $w \in v.\mathtt{T_nbrs}^t$, and yet $e \notin E_c^t$. Notice that e may be a dangling edge of v if w does not regard it as a tree edge or if w does regard it as a tree edge, but there is an inconsistency in the parent-child relations between v and w. We say that e is *dangling* without mentioning v if v is not important or clear from the context.

Fix some node $v \in V$. We make an extensive use of the variable $v.\delta \in v.\texttt{T_nbrs} \cup \{v\} \cup \{\bot\}$ defined for the sake of the analysis by setting

$$v.\delta^t = \begin{cases} v, & \text{if } v \text{ owns a token at time } t \\ v.\texttt{tkn_d}^t, & \text{if } v \text{ does not own a token at time } t \text{ and } \{v, v.\texttt{tkn_d}^t\} \in E_c^t \\ \bot, & \text{otherwise} \end{cases} .$$

Let $G_\delta^t = (V, E_\delta^t)$ be the directed graph defined by setting $E_\delta^t = \{(v, v.\delta) \mid v \in V \wedge v.\delta \neq \bot\}$. The weakly connected components of G_δ^t are referred to as *δ-components*. By definition, G_δ^t is a pseudoforest and the δ-components are pseudotrees. Moreover, all edges of the undirected version of G_δ^t that are not self loops are also edges of G_c^t.

A δ-component is said to be *selfish* if it includes a self loop. A non-selfish δ-component may be cycle free or include a cycle that involves two or more nodes. By definition, node $v \in V$ is incident to a self loop in G_δ^t if and only if it owns a token at time t, which means that the nodes of a selfish δ-component own exactly one token (all together), whereas the nodes of a non-selfish δ-component do not own any token.

For a token κ that is alive at time t, let D_κ^t be the selfish δ-component that includes the node that owns κ and let $\overline{D}_\kappa^t$ be the undirected version of D_κ^t excluding the self loop (notice that $\overline{D}_\kappa^t$ is always a tree). We say that a node $v \in V$ is *covered* by κ at time t if $v \in \mathcal{V}(D_\kappa^t)$; v is said to be *uncovered* at time t if it belongs to a non-selfish δ-component in G_δ^t. A token κ is said to be *strong* at time t if (1) κ is alive at time t; (2) the nodes of D_κ^t admit no dangling edges at time t; and (3) $\overline{D}_\kappa^t$ forms a compatible component in G_c^t.

The selfish δ-components obey the following dynamics. Consider a token κ that is dispatched by a node $v \in V$ in round $t - 1 > 0$ over edge $e = \{v, w\}$ and assume that the dispatch is successful so that κ is acquired by node w in round t. If w does not experience a restart in round t, then the δ-component D_κ is updated so that (1) its sole self loop moves from $(v, v) \in E_\delta^t$ to $(w, w) \in E_\delta^{t+1}$; and (2) the direction of e changes from $(w, v) \in E_\delta^t$ to $(v, w) \in E_\delta^{t+1}$.

The Main Theorems. The analysis of our algorithm stands on three main theorems.

▶ **Theorem 1.** *Each node in V may experience at most one restart throughout the execution. Moreover, there exists some $t_r^* = O(N)$ such that from time t_r^* onward, (1) all nodes are covered and do not experience any restart; and (2) no token dies. This means in particular that if a token κ is alive at time $t \geq t_r^*$ with $\mathcal{V}(D_\kappa^t) = V$, then κ is alive at time t' with $\mathcal{V}(D_\kappa^{t'}) = V$ for every $t' > t$.*

The time t_r^* promised in Thm. 1 can be thought of as the time at which the algorithm is guaranteed to recover from the faults that the initial configuration may exhibit. To establish this theorem, we first prove that a token that completes a *natural traversal* – namely, a traversal that has been invoked as part of the execution, rather that being sprouted from the initial configuration – must be strong. Next, we prove that a token that completed a natural traversal and turned cold, remains strong until the next time it turns hot. Finally, we prove that after $O(N)$ time, any strong token that starts a traversal is guaranteed to complete the traversal (without dying in the middle). Refer to [44] for a full proof.

▶ **Theorem 2.** *Let $t_s^* \geq t_r^*$ be the earliest time such that exactly one token is alive at time t_s^*. Then $t_s^* = O(N \log^2 N)$ in expectation and whp.*

Thm. 2 establishes the stabilization properties of our algorithm: Recall that after time t_r^*, a single token is alive if and only if this token covers the whole graph. Therefore, from time t_s^* onward, the graph admits no crossing edges and any invocation of `Propose` excludes

the root transfer and proposing epochs. This means that the prnt and chld variables of the nodes in V remain unchanged and, in particular, the (single) root remains fixed. The main probabilistic argument behind the proof of Thm. 2 is that if a token κ starts a phase ϕ after time t_r^* and it is not the last remaining token in the graph, then with probability low-bounded by a positive constant, κ is dissolved when ϕ ends. Again, the reader is referred to [44] for a full proof.

▶ **Theorem 3.** *Until it stabilizes, the algorithm sends $O(N \log^2 N)$ messages in expectation and whp. After stabilizing, the algorithm sends at most one message per round.*

Proof. Node $v \in V$ sends at most one message per round, thus $O(N)$ messages are sent in round 0. In rounds $t > 0$, node v may send a message only if it holds a token, thus establishing the desired bound on the message complexity after stabilization, when one token is alive. It remains to bound the number of messages sent during the time interval $[1, t_s^*)$. This is done separately for each message type.

Consider a pass_tkn message M sent from node $v \in V$ to a node $u \in N(v)$. If u ignores M, then the token that M carries dies. Thm. 1 ensures that the total number of distinct tokens that existed throughout the execution is up-bounded by $2N$, hence $O(N)$ pass_tkn messages are ignored throughout the execution; we focus hereafter on the pass_tkn messages that are not ignored.

Consider a node $v \in V$. Recalling that v suffers a premature discovery (and invokes Restart) if it receives an (unignored) discovery pass_tkn message while $v.\texttt{tmr} < C_{\mathrm{tr}}N$, we conclude that v receives $O(1)$ discovery pass_tkn messages throughout any time interval of length $C_{\mathrm{tr}}N$. Moreover, v must receive a discovery pass_tkn message between any two retraction pass_tkn messages it sends, thus v sends $O(1)$ retraction pass_tkn messages throughout any time interval of length $C_{\mathrm{tr}}N$.

Since any pass_tkn message sent over edge $e \in E$ is either a discovery message or a retraction message of one of e's endpoints, it follows that we can charge all the (unignored) pass_tkn messages sent throughout the execution to the nodes so that each node $v \in V$ is charged with $O(1)$ messages in the time interval $[1 + iC_{\mathrm{tr}}N, 1 + (i+1)C_{\mathrm{tr}}N)$ for any $i \in \mathbb{Z}_{\geq 0}$. Therefore, the total number of pass_tkn messages sent during the time interval $[1, t_s^*)$ is $O(N) + O(N) \cdot \frac{t_s^*}{N} = O(N + t_s^*)$.

An upper bound of $O(N + t_s^*)$ on the number of root_trns (resp., propose) messages sent during $[1, t_s^*)$ follows from the observation that a node must receive (and send) at least one pass_tkn message between any two root_trns (resp., propose) messages it sends. Finally, each accept message can be charged to either a propose message or to a dying token; the latter accounts to $O(N)$ additional messages.

To sum up, the total number of messages sent during the time interval $[1, t_s^*)$ is $O(N + t_s^*)$. Thm. 3 follows from Thm. 2 ensuring that $t_s^* \leq O(N \log^2 N)$ in expectation and whp. ◀

References

1 Yehuda Afek and Anat Bremler. Self-stabilizing unidirectional network algorithms by power-supply. In *SODA*, volume 97, pages 111–120, 1997.

2 Yehuda Afek and Geoffrey M Brown. Self-stabilization over unreliable communication media. *Distributed Computing*, 7(1):27–34, 1993.

3 Yehuda Afek and Eli Gafni. Time and message bounds for election in synchronous and asynchronous complete networks. *SIAM Journal on Computing*, 20(2):376–394, 1991.

4 Yehuda Afek, Shay Kutten, and Moti Yung. Memory-efficient self stabilizing protocols for general networks. In *International Workshop on Distributed Algorithms*, pages 15–28. Springer, 1990.

5 Yehuda Afek, Shay Kutten, and Moti Yung. The local detection paradigm and its applications to self-stabilization. *Theoretical Computer Science*, 186(1-2):199–229, 1997.

6 Marcos K Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. Stable leader election. In *International Symposium on Distributed Computing*, pages 108–122. Springer, 2001.

7 Thamer Alsulaiman, Andrew Berns, and Sukumar Ghosh. Low-communication self-stabilizing leader election in large networks. In Teruo Higashino, Yoshiaki Katayama, Toshimitsu Masuzawa, Maria Potop-Butucaru, and Masafumi Yamashita, editors, *Stabilization, Safety, and Security of Distributed Systems*, pages 348–350, 2013.

8 Karine Altisen, Stéphane Devismes, Swan Dubois, and Franck Petit. Introduction to distributed self-stabilizing algorithms. *Synthesis Lectures on Distributed Computing Theory*, 8(1):1–165, 2019.

9 Anish Arora and Mohamed Gouda. Distributed reset. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 316–331. Springer, 1990.

10 Hagit Attiya and Jennifer Welch. *Distributed computing: fundamentals, simulations, and advanced topics*, volume 19. John Wiley & Sons, 2004.

11 Baruch Awerbuch. Optimal distributed algorithms for minimum weight spanning tree, counting, leader election, and related problems. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 230–240, 1987.

12 Baruch Awerbuch, Israel Cidon, and Shay Kutten. Communication-optimal maintenance of replicated information. In *Proceedings [1990] 31st Annual Symposium on Foundations of Computer Science*, pages 492–502. IEEE, 1990.

13 Baruch Awerbuch, Oded Goldreich, Ronen Vainish, and David Peleg. A trade-off between information and communication in broadcast protocols. *Journal of the ACM (JACM)*, 37(2):238–256, 1990.

14 Baruch Awerbuch, Shay Kutten, Yishay Mansour, Boaz Patt-Shamir, and George Varghese. Time optimal self-stabilizing synchronization. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 652–661, 1993.

15 Baruch Awerbuch, Boaz Patt-Shamir, and George Varghese. Self-stabilization by local checking and correction. In *FOCS*, pages 268–277, 1991.

16 Baruch Awerbuch, Boaz Patt-Shamir, George Varghese, and Shlomi Dolev. Self-stabilization by local checking and global reset. In *International Workshop on Distributed Algorithms*, pages 326–339. Springer, 1994.

17 Baruch Awerbuch and George Varghese. Distributed program checking: a paradigm for building self-stabilizing distributed protocols. In *[1991] Proceedings 32nd Annual Symposium of Foundations of Computer Science*, pages 258–267. IEEE, 1991.

18 Joffroy Beauquier, Maria Gradinariu, and Colette Johnen. Memory space requirements for self-stabilizing leader election protocols. In *Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, pages 199–207, 1999.

19 Lélia Blin, Maria Potop-Butucaru, Stephane Rovedakis, and Sébastien Tixeuil. A new self-stabilizing minimum spanning tree construction with loop-free property. In *International Symposium on Distributed Computing*, pages 407–422. Springer, 2009.

20 Lélia Blin and Sébastien Tixeuil. Compact self-stabilizing leader election for general networks. *Journal of Parallel and Distributed Computing*, 144:278–294, 2020.

21 Janna Burman and Shay Kutten. Time optimal asynchronous self-stabilizing spanning tree. In *International Symposium on Distributed Computing*, pages 92–107. Springer, 2007.

22 Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

23 Miguel Castro, Peter Druschel, A-M Kermarrec, and Antony IT Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in communications*, 20(8):1489–1499, 2002.

24 Tushar D Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407, 2007.

25 Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):1–26, 2008.

26 Johanne Cohen, Jonas Lefèvre, Khaled Maamra, George Manoussakis, and Laurence Pilard. The first polynomial self-stabilizing 1-maximal matching algorithm for general graphs. *Theoretical Computer Science*, 782:54–78, 2019.

27 Zeev Collin and Shlomi Dolev. Self-stabilizing depth-first search. *Information Processing Letters*, 49(6):297–301, 1994.

28 Alain Cournier, Stéphane Rovedakis, and Vincent Villain. The first fully polynomial stabilizing algorithm for bfs tree construction. *Information and Computation*, 265:26–56, 2019.

29 Ajoy K. Datta, Colette Johnen, Franck Petit, and Vincent Villain. Self-stabilizing depth-first token circulation in arbitrary rooted networks. *Distributed Computing (DC)*, 13(4):207–2018, 2000.

30 Ajoy K. Datta, Lawrence L. Larmore, and Toshimitsu Masuzawa. Communication efficient self-stabilizing algorithms for breadth-first search trees. In *International Conference On Principles Of DIstributed Systems*, pages 293–306. Springer, 2014.

31 Ajoy K Datta, Lawrence L Larmore, and Priyanka Vemula. A self-stabilizing o (k)-time k-clustering algorithm. *The Computer Journal*, 53(3):342–350, 2010.

32 B DeCleene, L Dondeti, S Griffin, T Hardjono, D Kiwior, J Kurose, D Towsley, S Vasudevan, and C Zhang. Secure group communications for wireless networks. In *2001 MILCOM Proceedings Communications for Network-Centric Operations: Creating the Information Force (Cat. No. 01CH37277)*, volume 1, pages 113–117. IEEE, 2001.

33 Carole Delporte-Gallet, Stéphane Devismes, and Hugues Fauconnier. Robust stabilizing leader election. In *SSS'07*, volume 4838, pages 219–233. Springer, 2007. `doi:10.1007/978-3-540-76627-8_18`.

34 Stéphane Devismes and Colette Johnen. Silent self-stabilizing bfs tree algorithms revisited. *Journal of Parallel and Distributed Computing*, 97:11–23, 2016.

35 Stéphane Devismes, Toshimitsu Masuzawa, and Sébastien Tixeuil. Communication efficiency in self-stabilizing silent protocols. In *ICDCS'09*, pages 474–481. IEEE, 2009. `doi:10.1109/ICDCS.2009.24`.

36 Edsger W Dijkstra. Self-stabilization in spite of distributed control. In *Selected writings on computing: a personal perspective*, pages 41–46. Springer, 1982.

37 Danny Dolev, Maria Klawe, and Michael Rodeh. An o (n log n) unidirectional distributed algorithm for extrema finding in a circle. *Journal of algorithms*, 3(3):245–260, 1982.

38 Shlomi Dolev. *Self-stabilization.* MIT Press, Cambridge, MA, USA, 2000.

39 Shlomi Dolev, Mohamed G Gouda, and Marco Schneider. Memory requirements for silent stabilization. *Acta Informatica*, 36(6):447–462, 1999.

40 Shlomi Dolev and Ted Herman. Superstabilizing protocols for dynamic distributed systems. *Chicago J. Theor. Comput. Sci.*, 1997, 1997.

41 Shlomi Dolev, Amos Israeli, and Shlomo Moran. Resource bounds for self stabilizing message driven protocols. In *Proceedings of the tenth annual ACM symposium on Principles of distributed computing*, pages 281–293, 1991.

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

43 Anaïs Durand and Shay Kutten. Reducing the number of messages in self-stabilizing protocols. In *International Symposium on Stabilizing, Safety, and Security of Distributed Systems*, pages 133–148. Springer, 2019.

44 Xavier Défago, Yuval Emek, Shay Kutten, Toshimitsu Masuzawa, and Yasumasa Tamura. Communication efficient self-stabilizing leader election (full version), 2020. `arXiv:2008.04252`.

45 Michael Elkin. A simple deterministic distributed mst algorithm with near-optimal time and message complexities. *Journal of the ACM (JACM)*, 67(2):1–15, 2020.

46 Greg N Frederickson and Nancy A Lynch. Electing a leader in a synchronous ring. *Journal of the ACM (JACM)*, 34(1):98–115, 1987.

47 R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. Program. Lang. Syst.*, 5(1):66–77, January 1983.

48 Mohsen Ghaffari and Fabian Kuhn. Distributed mst and broadcast with fewer messages, and faster gossiping. In *32nd International Symposium on Distributed Computing (DISC 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

49 Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, 2003.

50 Robert Gmyr and Gopal Pandurangan. Time-message trade-offs in distributed algorithms. *arXiv preprint arXiv:1810.03513*, 2018.

51 Kostas P Hatzis, George P Pentaris, Paul G Spirakis, Vasilis T Tampakas, and Richard B Tan. Fundamental control algorithms in mobile networks. In *Proceedings of the eleventh annual ACM symposium on Parallel algorithms and architectures*, pages 251–260, 1999.

52 Wendi Rabiner Heinzelman, Anantha Chandrakasan, and Hari Balakrishnan. Energy-efficient communication protocol for wireless microsensor networks. In *Proceedings of the 33rd annual Hawaii international conference on system sciences*, pages 10–pp. IEEE, 2000.

53 Lisa Higham and Zhiying Liang. Self-stabilizing minimum spanning tree construction on message-passing networks. In *International Symposium on Distributed Computing*, pages 194–208. Springer, 2001.

54 Shing-Tsaan Huang and Nian-Shing Chen. Self-stabilizing depth-first token circulation on networks. *Distributed Computing*, 7(1):61–66, 1993.

55 Michael Isard. Autopilot: automatic data center management. *ACM SIGOPS Operating Systems Review*, 41(2):60–67, 2007.

56 Colette Johnen, Gianluigi Alari, Joffroy Beauquier, and Ajoy K Datta. Self-stabilizing depth-first token passing on rooted networks. In *International Workshop on Distributed Algorithms*, pages 260–274. Springer, 1997.

57 Colette Johnen and Joffroy Beauquier. Distributed self-stabilizing depth-first token circulation with constant memory. In *2nd Workshop on Self-Stabilizing System (WSS)*, pages 4–1, 1995.

58 Shmuel Katz and Kenneth J Perry. Self-stabilizing extensions for meassage-passing systems. *Distributed Computing*, 7(1):17–26, 1993.

59 Valerie King, Shay Kutten, and Mikkel Thorup. Construction and impromptu repair of an mst in a distributed network with o (m) communication. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, pages 71–80, 2015.

60 Valerie King, Jared Saia, Vishal Sanwalani, and Erik Vee. Towards secure and scalable computation in peer-to-peer networks. In *2006 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*, pages 87–98. IEEE, 2006.

61 Ephraim Korach, Shlomo Moran, and Shmuel Zaks. Tight lower and upper bounds for some distributed algorithms for a complete network of processors. In *Proceedings of the third annual ACM symposium on Principles of distributed computing*, pages 199–207, 1984.

62 Alex Kravchik and Shay Kutten. Time optimal synchronous self stabilizing spanning tree. In *International Symposium on Distributed Computing*, pages 91–105. Springer, 2013.

63 Shay Kutten and Dmitry Zinenko. Low communication self-stabilization through randomization. In *International Symposium on Distributed Computing*, pages 465–479. Springer, 2010.

64 Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

65 Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

66 Butler W Lampson. How to build a highly available system using consensus. In *International Workshop on Distributed Algorithms*, pages 1–17. Springer, 1996.

67 Gérard Le Lann. Distributed systems - towards a formal approach. In *IFIP Congress*, pages 155–160, 1977.

68 Mikel Larrea, Antonio Fernández, and Sergio Arévalo. Optimal implementation of the weakest failure detector for solving consensus (brief announcement). In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, page 334, 2000.

69 Edward K Lee and Chandramohan A Thekkath. Petal: Distributed virtual disks. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pages 84–92, 1996.

70 Nancy A Lynch. *Distributed algorithms*. Elsevier, 1996.

71 Navneet Malpani, Jennifer L Welch, and Nitin Vaidya. Leader election algorithms for mobile ad hoc networks. In *Proceedings of the 4th international workshop on Discrete algorithms and methods for mobile computing and communications*, pages 96–103, 2000.

72 Ali Mashreghi and Valerie King. Broadcast and minimum spanning tree with $o(m)$ messages in the asynchronous congest model. *arXiv preprint arXiv:1806.04328*, 2018.

73 Ali Mashreghi and Valerie King. Brief announcement: Faster asynchronous mst and low diameter tree construction with sublinear communication. In *33rd International Symposium on Distributed Computing (DISC 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

74 Toshimitsu Masuzawa. Silence is golden: self-stabilizing protocols communication-efficient after convergence. In *Symposium on Self-Stabilizing Systems*, pages 1–3. Springer, 2011.

75 Toshimitsu Masuzawa, Taisuke Izumi, Yoshiaki Katayama, and Koichi Wada. Brief announcement: Communication-efficient self-stabilizing protocols for spanning-tree construction. In *OPODIS'09*, pages 219–224, 2009.

76 Gopal Pandurangan, Peter Robinson, and Michele Scquizzato. A time-and message-optimal distributed algorithm for minimum spanning trees. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, pages 743–756, 2017.

77 Boaz Patt-Shamir and Mor Perry. Proof-labeling schemes: Broadcast, unicast and in between. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems*, pages 1–17. Springer, 2017.

78 Charles E Perkins and Elizabeth M Royer. Ad-hoc on-demand distance vector routing. In *Proceedings WMCSA'99. Second IEEE Workshop on Mobile Computing Systems and Applications*, pages 90–100. IEEE, 1999.

79 Radia Perlman. *Interconnections: bridges, routers, switches, and internetworking protocols*. Addison-Wesley Professional, 2000.

80 Franck Petit. Fast self-stabilizing depth-first token circulation. In *International Workshop on Self-Stabilizing Systems*, pages 200–215. Springer, 2001.

81 Sudarshan Vasudevan, Jim Kurose, and Don Towsley. Design and analysis of a leader election algorithm for mobile ad hoc networks. In *Proceedings of the 12th IEEE International Conference on Network Protocols, 2004. ICNP 2004.*, pages 350–360. IEEE, 2004.

Gathering on a Circle with Limited Visibility by Anonymous Oblivious Robots

Giuseppe A. Di Luna
DIAG, Sapienza University of Rome, Italy
diluna@diag.uniroma1.it

Ryuhei Uehara
School of Information Science, JAIST, Ishikawa, Japan
uehara@jaist.ac.jp

Giovanni Viglietta[1]
School of Information Science, JAIST, Ishikawa, Japan
johnny@jaist.ac.jp

Yukiko Yamauchi
Department of Informatics, Graduate School of ISEE, Kyushu University, Japan
yamauchi@inf.kyushu-u.ac.jp

Abstract

A swarm of anonymous oblivious mobile robots, operating in deterministic Look-Compute-Move cycles, is confined within a circular track. All robots agree on the clockwise direction (chirality), they are activated by an adversarial semi-synchronous scheduler (SSYNCH), and an active robot always reaches the destination point it computes (rigidity). Robots have limited visibility: each robot can see only the points on the circle that have an angular distance strictly smaller than a constant ϑ from the robot's current location, where $0 < \vartheta \leq \pi$ (angles are expressed in radians).

We study the Gathering problem for such a swarm of robots: that is, all robots are initially in distinct locations on the circle, and their task is to reach the same point on the circle in a finite number of turns, regardless of the way they are activated by the scheduler. Note that, due to the anonymity of the robots, this task is impossible if the initial configuration is rotationally symmetric; hence, we have to make the assumption that the initial configuration be rotationally asymmetric.

We prove that, if $\vartheta = \pi$ (i.e., each robot can see the entire circle except its antipodal point), there is a distributed algorithm that solves the Gathering problem for swarms of any size. By contrast, we also prove that, if $\vartheta \leq \pi/2$, no distributed algorithm solves the Gathering problem, regardless of the size of the swarm, even under the assumption that the initial configuration is rotationally asymmetric and the visibility graph of the robots is connected.

The latter impossibility result relies on a probabilistic technique based on random perturbations, which is novel in the context of anonymous mobile robots. Such a technique is of independent interest, and immediately applies to other Pattern-Formation problems.

2012 ACM Subject Classification Computing methodologies → Distributed algorithms; Theory of computation → Self-organization

Keywords and phrases Mobile robots, Gathering, limited visibility, circle

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.12

Related Version A full version of the paper is available at `https://arxiv.org/abs/2005.07917`.

Acknowledgements The authors would like to thank the anonymous reviewers for greatly improving the readability of this paper.

[1] Corresponding author

© Giuseppe A. Di Luna, Ryuhei Uehara, Giovanni Viglietta, and Yukiko Yamauchi;
licensed under Creative Commons License CC-BY

1 Introduction

Background

One of the most popular models for distributed mobile robotics is the *Look-Compute-Move* (LCM) one [17, 18]. In this model, a Euclidean space, usually the real plane $\mathbb{R}^2$, is inhabited by a "swarm" of punctiform and autonomous computational entities, the *robots*. Each robot, upon activation, takes a snapshot of the space (Look), uses this snapshot to compute its destination (Compute), and then reaches its destination point (Move).

The activation pattern of the robots is controlled by an external scheduler. At one end of the synchrony spectrum, there is the *fully-synchronous* scheduler (FSYNCH): in this case, time is divided into discrete units (the *turns*). At each turn, the entire swarm is activated, and all robots synchronously execute one LCM cycle. At the other end, there is the *asynchronous* scheduler (ASYNCH), where robot activations are independent, and LCM cycles are not synchronized. Somewhere halfway, there is the *semi-synchronous* scheduler (SSYNCH), which activates an arbitrary subset of robots at each turn, with the restriction of activating each robot infinitely often.

A common assumption in the context of mobile robots is the lack of persistent memory: a robot does not remember anything about past activations (*obliviousness*). Other assumptions are *anonymity*, where robots do not have visible and distinguishable identifying features, and *silence*, where robots do not have explicit communication primitives (such as lights [14]).

Obliviousness, anonymity, and silence are practical, useful, and desirable properties: an algorithm for oblivious robots is inherently resilient to transient memory failures; one for anonymous robots is ideal in privacy-sensitive contexts; an algorithm for silent robots works even in scenarios where communication is jammed or unfeasible (e.g., hostile environments or underwater deployment).

The purpose of such an ensemble of weak robots is to reach a common goal in a coordinated way. Interestingly, it has been shown that mobile robots can solve an extensive set of problems [18], ranging from forming patterns [8, 19, 21, 28, 30] to simulating a powerful Turing-complete movable entity [12].

Among all tasks, a particularly relevant one is *Gathering* [1, 4, 5, 9, 13, 15, 25, 26]: in finite time, all robots have to reach the same point and stop there. Initial works assumed robots to see the entire space (*full visibility*). However, a more realistic assumption [3, 24] is that a robot be able to see only a portion of the space (*limited visibility*).

In this paper, we study the Gathering problem for a swarm of oblivious robots with limited visibility constrained to move within a circle: each robot can see only the points on the circle that have an angular distance strictly smaller than a certain *visibility range* ϑ. We assume that robots have no agreement on common coordinates apart from sharing the same notion of clockwise direction on the circle.

From a practical perspective, the restriction of moving along a predetermined path arises in wide variety of scenarios: railway lines, roads, tunnels, waterways, etc. We argue that, among all topologically equivalent curves, the circle is the most meaningful to study: a solution for it readily extends to all other closed curves.

From a theoretical perspective, confining the swarm on a circle (hence, a non-simply connected space) rules out all the strategies typically used for robots in the plane, such as moving toward the center of the visible set of robots (an example is in [2]). Moreover, robots cannot use any asymmetries in the environment to identify a gathering point: this makes the circle the most challenging setting for Gathering (and in general, for any problem where symmetry breaking helps).

Apart from [16], which examined the problem of scattering on a circle (reaching a final configuration where the robots are uniformly spaced out), no other works studied the computational power of oblivious robots when confined to curves: this is rather surprising, considering the copious existing literature on oblivious robots [17, 18]. To the best of our knowledge, the present paper is the first to investigate the Gathering problem for oblivious, silent, and anonymous robots on a circle with limited visibility (some works investigated the Gathering problem in a ring graph, which is a discretization of the circle [6, 7, 11, 22, 23]).

Our contributions

We consider a swarm of n oblivious, anonymous, and silent robots that start at distinct locations on a circle. Robots do not agree on a common system of coordinates, but they do share the same *handedness* (i.e., they have a common notion of clockwise direction). When a robot decides to move, it reaches its destination point (robots are *rigid*). Moreover, robots have no information on the swarm's size, n. Each robot can see only the points on the circle that have an angular distance strictly smaller than a certain visibility range ϑ. We must assume that the initial configuration is rotationally asymmetric, otherwise the scheduler may activate robots in a such a way as to preserve the rotational symmetry, and Gathering cannot be achieved.

After giving all the necessary definitions and some preliminary results (Section 2), our first contribution is to show that there is no distributed algorithm that solves the Gathering problem in SSYNCH when $\vartheta \leq \pi/2$, i.e., each robot is only able to see at most half of the circle (Section 3). Surprisingly, this holds even if the initial configuration is rotationally asymmetric, the visibility graph of the swarm (i.e., the graph of intervisibility between robots) is connected, and all robots know n.

Our proof uses a novel technique based on random perturbations, of which we offer an intuitive probabilistic argument, as well as a formal and more elementary proof by derandomization. We show that, for any given distributed algorithm, either there exists an asymmetric configuration of robots that can evolve into a symmetric one within one time unit (in SSYNCH), or there is an asymmetric configuration where no robot can move. In either case, Gathering is impossible.

We stress that our result has a profound meaning, since it shows that, when $\vartheta \leq \pi/2$, any distributed algorithm, including the ones that do not aim to solve Gathering, has an initial asymmetric configuration that either repeats forever or evolves into a symmetric configuration in one step. This implies a novel impossibility result for geometric Pattern Formation on circles: even when robots start from an asymmetric configuration, they cannot form a target asymmetric pattern. This is in striking contrast with the unlimited-visibility setting, where, even under the ASYNCH scheduler, from any asymmetric configuration any pattern is formable [18].

To the best of our knowledge, this the first impossibility proof for oblivious robots that neither relies on invariants induced by symmetries (e.g., [21, 29]) nor on the disconnection of the visibility graph (e.g., [12, 31]). Due to the above, we think that our technique is of independent interest, and its core ideas could be applied to other settings, as well.

On the possibility side, we show that, if $\vartheta = \pi$ (i.e., each robot can see the entire circle except its antipodal point), there is a distributed algorithm that solves the Gathering problem in SSYNCH for swarms of any size (Section 4). The algorithm's strategy is to attempt to elect a unique leader and form a multiplicity point, where all robots will subsequently gather. The main challenge is that, since a robot ignores whether its antipodal point is occupied by another robot or not (robots do not know n), there may be an ambiguity on who is the true

leader. Several robots may believe to be the leader, but this also comes with the awareness of the possibility of being wrong: these "undecided" robots will make some adjustment moves, which eventually result in a configuration where one robot is absolutely certain of being the true leader. The leader will then form a multiplicity point by moving to another robot, and finally all other robots will join them.

2 Model definition and preliminaries

Measuring angles

Let $C \subset \mathbb{R}^2$ be a circle, and let a and b be points of C. The *angular distance* between a and b (with respect to C) is the measure of the angle subtended at the center of C by the shorter arc with endpoints a and b. It follows that the angular distance between two points is a real number in the interval $[0, \pi]$, where angles are expressed in radians. Two points of C are *antipodal* of each other if their angular distance is π. The *α-neighborhood* of a point $q \in C$ is the set of points of C whose angular distance from q is strictly smaller than α. The $(\pi/2)$-neighborhood of q is also called the *open semicircle* centered at q.

Furthermore, if a and b are distinct points of C, we define $cw(a, b)$ as the measure of the *clockwise* angle $\angle acb$, where c is the center of C. Note that the order of the two arguments matters, and so for instance $cw(a, b) + cw(b, a) = 2\pi$. We also define $cw(a, a) = 0$ for every a.

Rotational symmetry

Let S be a finite multiset of points on a circle C. We say that S is *rotationally symmetric* if there is a non-identical rotation around the center of C that leaves S unchanged (also preserving multiplicities). If S is not rotationally symmetric, it is said to be *rotationally asymmetric*.

Angle sequences

Let S be a multiset of n points on a circle C, and let $p \in S$. Let p_1, p_2, $\dots$, p_n be the points of S taken in clockwise order starting from $p = p_1$ (coincident elements of S are ordered arbitrarily). We define the *angle sequence* of p (with respect to S) as the n-tuple $(cw(p_1, p_2), cw(p_2, p_3), \dots, cw(p_n, p_1))$. The case where all the elements of S are coincident is an exception, and in this case the angle sequence of the ith point of S, with $1 \le i \le n$, is defined as the n-tuple $(0, 0, \dots, 0, 0, 2\pi, 0, 0, \dots, 0, 0)$, where the term 2π appears in the ith position. Note that, with this convention, the sum of the elements of any angle sequence is always 2π.

The following is an easy observation.

▶ **Proposition 2.1.** *A non-empty multiset of points on a circle is rotationally asymmetric if and only if all its points have distinct angle sequences.* ◀

Mobile robots

Our model of mobile robots is among the standard ones defined in [17, 18]. A swarm of $n > 1$ robots is located on a circle $C \subset \mathbb{R}^2$, where each robot is a computational unit that occupies a point of C (which may change over time) and operates in deterministic Look-Compute-Move cycles.

Time is discretized and subdivided into units, and at each time unit an adversarial (*semi-synchronous*) *scheduler* decides which robots are active and which are inactive. An inactive robot remains idle for that time unit, whereas an active robot takes a *snapshot* of

its surroundings, consisting of an arc $B \subseteq C$ and a list of points of B that are currently occupied by robots, it computes a destination point in B as a function of the snapshot, and it instantly moves to the destination point. The only restriction to the scheduler is that no robot should remain inactive for infinitely many consecutive time units.

Robots may have *full visibility*, in which case the arc B defining a snapshot coincides with the entire circle C, or they may have *limited visibility*, in which case the arc B consists of the ϑ-neighborhood of the current position of the robot taking the snapshot, where ϑ is a positive constant called the *visibility range* of the robots.

Furthermore, each robot has its own *local coordinate system*, meaning that each snapshot it takes of an arc $B \subseteq C$ is actually a roto-translated copy of B and the positions of the robots within B. Such a copy of B has its midpoint at the origin of the coordinate system (this corresponds to the location of the robot taking the snapshot) and its endpoints have non-negative x coordinate and the same y coordinate.

Robots are also capable of *weak multiplicity detection*, meaning that the snapshots they take contain some information on how many robots occupy each location. Specifically, a robot can tell if a point in a snapshot contains no robots, exactly one robot, or more than one robot: no information on the precise number of robots is given if this number is greater than one. A point occupied by more than one robot is also called a *multiplicity point*.

In order to simplify our notation, when no confusion arises, we will often identify a robot with its position on the circle. So, we may improperly refer to a robot as a point $p \in C$ or to a swarm of robots as a set $S \subset C$.

Gathering

A *distributed algorithm* is a function that maps a snapshot to a point within the snapshot itself. A robot *executes* a distributed algorithm A if, whenever it is activated and takes a snapshot Q, it moves to the destination point corresponding to $A(Q)$. In other words, at each time unit, an active robot chooses its destination point deterministically within its visibility range, based solely on the snapshot it currently has.

We stress that, as a consequence of the previous definitions, the robots in this model are *oblivious* (i.e., they have no memory of past observations), *anonymous* (i.e., a robot only identifies other robots by their positions in its local coordinate system, and not for instance by their IDs), *silent* (i.e., they cannot send messages to one another), *deterministic* (i.e., they cannot flip coins), *rigid* (i.e., they always reach the destination points they compute), they have *chirality* (i.e., they all agree on the clockwise direction on the circle), and they have no knowledge of n (i.e., a robot can only see other robots within its visibility range, and it does not know whether there are further robots outside of it).

We say that a distributed algorithm A solves the *Gathering problem* under condition P if, whenever all the $n > 1$ robots of a swarm located on a circle execute A, they eventually reach a configuration where all robots are in the same point of the circle and no robot ever moves again, provided that their initial configuration satisfies condition P, and regardless of the activation choices of the adversarial scheduler. Equivalently, we say that A is a *Gathering algorithm* under condition P.

We remark that all the robots in the swarm must execute the same algorithm A (i.e., robots are *uniform*), and the algorithm has to work for swarms of any size $n > 1$, where n is *not* a parameter of A. Also note that the robots' positions should not simply converge to the same limit, but they must actually become coincident in a finite number of time units for Gathering to be achieved (albeit there is no bound on the number of time units this process may take). Such a distinction will be important for the design of a correct Gathering algorithm in Section 4, while our impossibility proof of Section 3 shows that robots cannot even converge to a point, and much less gather.

Initial conditions

There are several meaningful options concerning our choice of the initial condition P for the Gathering problem. A typical assumption is that the n robots be initially located in n distinct points of the circle: while not strictly necessary, this is a common requirement for the Gathering problem (e.g., [4, 15, 28]).

Another assumption that we may make is that the *visibility graph* of the robots be initially connected. By "visibility graph" we mean the graph whose nodes are the n robots, where there is an edge between two robots if and only if they are mutually visible, i.e., if their angular distance is less than ϑ. This assumption is another common one (e.g., [2, 3, 10, 20, 27]) and, although not strictly necessary, it is justified by the intuition that different connected components of the visibility graph may never become aware of each other, and therefore may fail to gather. We will make this assumption in Section 3 to strengthen our impossibility result, and we will not need to explicitly make it in Section 4, because it will come as a consequence of other assumptions.

An important mandatory condition is that the multiset of the robots' positions on the circle should be rotationally asymmetric, due to the following.

▶ **Proposition 2.2.** *Let S be any rotationally symmetric multiset of $n > 1$ points on a circle. There is no Gathering algorithm under the initial condition that the multiset of robots' positions is S.* ◀

Since the robots are oblivious, this condition should hold true not only at the beginning, but at all times during the execution of a Gathering algorithm: the robots should never "accidentally" form a rotationally symmetric multiset, or they will be unable to gather.

▶ **Corollary 2.3.** *Throughout the execution of any Gathering algorithm, the robots' positions must always form a rotationally asymmetric multiset.* ◀

3 Impossibility of Gathering for $\vartheta \leq \pi/2$

Outline

In this section we prove that, if each robot can see at most an open semicircle (i.e., $\vartheta \leq \pi/2$), then no distributed algorithm solves the Gathering problem, even under some strong assumptions on the initial configuration, and even if the robots know the size of the swarm.

Our technique is essentially probabilistic, and it starts by defining a set of perturbations of a regular configuration. Then, by analyzing the behavior of a generic distributed algorithm on all perturbations of a swarm that satisfy some initial conditions, we will show that the algorithm either (i) allows the construction of a rotationally asymmetric configuration that can evolve into a rotationally symmetric one (under a semi-synchronous scheduler) or (ii) leaves the configuration unchanged forever. In both cases, the algorithm does not solve the Gathering problem on some configurations.

Perturbations

For the rest of this section, we will denote by C the unit circle centered at the origin. A finite set $S \subset C$ is *regular* if $(1, 0) \in S$ and all points of S have the same angle sequence. Hence, for every positive integer n, there is a unique regular set of size n: for $n \geq 3$, this is the set of vertices of the regular n-gon centered at the origin and having a vertex in $(1, 0)$.

Let S be the regular set of size n, and let $p_1, p_2, \ldots, p_n$ be the points of S taken in clockwise order, starting from $p_1 = (1, 0)$. Let $\varepsilon \in \mathbb{R}$ with $0 < \varepsilon < 2\pi/n$, and let $\gamma = (\gamma_1, \gamma_2, \ldots, \gamma_n) \in [0, 1]^n \subset \mathbb{R}^n$. The *$\varepsilon$-perturbation* of S with *coefficients* γ is the set

$S' \subset C$ of size n such that, for all $1 \le i \le n$, there is a (unique) point $p'_i \in S'$ with $cw(p_i, p'_i) = \gamma_i \cdot \varepsilon$, called the *perturbed copy* of p_i. So, any ε-perturbation of S is obtained by rotating each point of S clockwise around the origin by an angle in $[0, \varepsilon]$.

Furthermore, for $1 \le i \le n$, we say that two coefficient n-tuples $\gamma = (\gamma_1, \gamma_2, \dots, \gamma_n)$ and $\gamma' = (\gamma'_1, \gamma'_2, \dots, \gamma'_n)$ are *i-related* if and only if they differ at most by their ith terms, i.e., for all $j \ne i$, we have $\gamma_j = \gamma'_j$. Note that the i-relation is an equivalence relation on $[0,1]^n$. With the previous paragraph's notation, we say that the set of all ε-perturbations of S whose coefficients are in a same equivalence class of the i-relation is a *bundle* of ε-perturbations of p_i. Intuitively, a bundle of ε-perturbations of p_i is obtained by first fixing a perturbation of all points of S except p_i, and then perturbing p_i in all possible ways.

Size of the swarm

We will prove that Gathering is impossible for any given visibility range $\vartheta \le \pi/2$, provided that the size of the swarm n is appropriate. Specifically, we say that a positive integer n is *compatible* with ϑ if three conditions hold on the regular set S of size n:

1. For every $p \in S$, the open semicircle centered at p contains exactly half of the points of S.
2. No two points of S have an angular distance of exactly ϑ.
3. There are two distinct points of S whose angular distance is smaller than ϑ.

We can show that there are arbitrarily large such integers:

▶ **Proposition 3.1.** *For any $\vartheta \le \pi/2$, there are arbitrarily large integers compatible with ϑ.* ◀

Choice of ε

For every integer n compatible with ϑ, we define a positive number $\varepsilon_{\vartheta,n}$, which will be used to construct perturbations of the regular set S of size n. We set $\varepsilon_{\vartheta,n} = \delta/2$, where $\delta = \min\{|\vartheta - 2\pi a/n| \mid a \in \mathbb{N},\ 0 \le a \le n\}$.

Since n is compatible with ϑ, it easily follows that $\varepsilon_{\vartheta,n} > 0$. Also, δ is at most half the angular distance between two consecutive points of S, and therefore $\varepsilon_{\vartheta,n} \le \pi/2n$. Moreover, our choice of $\varepsilon_{\vartheta,n}$ has some other desirable properties:

▶ **Proposition 3.2.** *Let n be an integer compatible with $\vartheta \le \pi/2$, let S be the regular set of size n, and let S' be an $\varepsilon_{\vartheta,n}$-perturbation of S. If $p \in S$, and $p' \in S'$ is the perturbed copy of p, the following hold:*

1. *The ϑ-neighborhood of p contains a point $q \in S$ if and only if the ϑ-neighborhood of p' contains the perturbed copy of q in S'.*
2. *The open semicircle D centered at p contains exactly half of the points of S', which are the perturbed copies of the points of S contained in D.*
3. *If D' is the open semicircle centered at p', then $S' \cap D = S' \cap D'$, and hence D' contains exactly half of the points of S'.* ◀

▶ **Corollary 3.3.** *Let n be an integer compatible with $\vartheta \le \pi/2$, and let S' be an $\varepsilon_{\vartheta,n}$-perturbation of the regular set S of size n. If two swarms of robots form S and S' respectively, their visibility graphs are isomorphic.* ◀

Combining configurations

Next we will describe a way of combining two configurations of robots into a new one that takes an open semicircle from each. This operation will be used to construct configurations of robots where a given distributed algorithm fails to make the robots gather.

Let S_1 and S_2 be two subsets of C, and let D be an open semicircle. The D-*combination* of S_1 and S_2 is defined as the set $(S_1 \cap D) \cup \rho(S_2 \cap D)$, where ρ is the rotation by π around the origin. In other words, this operation takes S_1, discards the points that do not lie in D, and replaces them with the points of S_2 that lie in D, by mapping them to their antipodal points.

Preliminary lemmas

We are now ready to give our first two lemmas, which deal with swarms forming perturbations of a regular configuration, and analyze the distributed algorithms that make robots move in some ways. The first lemma states that, if an algorithm makes a robot move to a point not currently occupied by another robot, then the algorithm cannot solve the Gathering problem.

▶ **Lemma 3.4.** *Let A be a distributed algorithm, let n be compatible with $\vartheta \leq \pi/2$, and consider a swarm of robots that forms an $\varepsilon_{\vartheta,n}$-perturbation S' of the regular set of size n. If there is a robot that, executing A, moves to a point not in S', then A does not solve the Gathering problem, even under the condition that the swarm initially forms a rotationally asymmetric set of n distinct points with a connected visibility graph.*

Proof (sketch). Assume that, if a robot located in $p' \in S'$ executes A, it moves to a point $q \notin S'$. Let $S'' = (S' \setminus \{p'\}) \cup \{q\}$, and let Q be the D-combination of S' and S'', where D is the open semicircle centered at p'. Consider a swarm initially forming Q, which is a rotationally asymmetric set of n distinct points with a connected visibility graph. If the scheduler activates only the robot in p', the configuration becomes rotationally symmetric. So, by Corollary 2.3, A is not a Gathering algorithm. ◀

The second lemma states that, if a distributed algorithm makes a robot r move on top of another robot r', and there is a perturbation of r such that the same algorithm makes r move on top of the same robot r', then the algorithm does not solve the Gathering problem.

▶ **Lemma 3.5.** *Let A be a distributed algorithm, let n be compatible with $\vartheta \leq \pi/2$, let S be the regular set of size n, and let S' and S'' be two distinct sets in the same bundle of $\varepsilon_{\vartheta,n}$-perturbations of $p \in S$, where $p' \in S'$ and $p'' \in S''$ are the perturbed copies of p. Assume that, if a swarm of robots forms S' and the robot in p' executes A, it moves to another robot, located in $q \in S'$. Also assume that, if a swarm of robots forms S'' and the robot in p'' executes A, it moves to the same point q. Then, A does not solve the Gathering problem, even under the condition that the swarm initially forms a rotationally asymmetric set of n distinct points with a connected visibility graph.*

Proof (sketch). Let D be the open semicircle centered at p, and let Q be the D-combination of S' and S''. Consider a swarm initially forming Q, which is a rotationally asymmetric set of n distinct points with a connected visibility graph. Suppose the scheduler activates only two robots: the one in p' and the one in the point antipodal to p''. After these two robots have executed A, the swarm's configuration becomes rotationally symmetric. ◀

Probabilistic argument

Our concluding argument goes as follows. Suppose for a contradiction that there is a Gathering algorithm A for some $\vartheta \le \pi/2$. Let n be an arbitrarily large integer compatible with ϑ, and let S be the regular set of size n. We will derive a contradiction by studying the behavior of A on the swarms forming the $\varepsilon_{\vartheta,n}$-perturbations of S. Specifically, let p_1, p_2, ..., p_n be the points of S taken in clockwise order, starting from $p_1 = (1, 0)$. Suppose that a swarm of n robots forms some $\varepsilon_{\vartheta,n}$-perturbation of S, with robot r_i occupying the perturbed copy of p_i, and let all robots execute algorithm A.

Let us first restrict ourselves to a bundle P of $\varepsilon_{\vartheta,n}$-perturbations of some $p_i \in S$, and let us analyze the possible behaviors of the robot r_i. Recall that, by definition of bundle, the perturbations in P have fixed coefficients for all the points of S except p_i, and perturb p_i in every possible way, by varying the coefficient $\gamma_i \in [0, 1]$. Observe that, by Lemma 3.4, A should never make r_i move to some unoccupied location, or A would not be a Gathering algorithm. Also, if two or more perturbations in the bundle P made r_i move to the same robot, then A would not be a Gathering algorithm, due to Lemma 3.5. However, by the pigeonhole principle, if n perturbations in P made r_i move to some other robot, then at least two of them would make it move to the same robot. It follows that at most $n - 1$ perturbations in P can make r_i move at all. So, all perturbations in P except a finite number of them must make r_i stay still.

Now, let us pick an $\varepsilon_{\vartheta,n}$-perturbation of S by choosing its coefficients $\gamma \in [0, 1]^n$ uniformly at random. Let us also define n random variables $X_i \colon [0, 1]^n \to \{0, 1\}$, with $1 \le i \le n$, such that $X_i(\gamma) = 0$ if and only if algorithm A makes the robot r_i stay still when the swarm's configuration is the $\varepsilon_{\vartheta,n}$-perturbation of S defined by the coefficients γ. By the above argument, for every bundle P of $\varepsilon_{\vartheta,n}$-perturbations of p_i, we have $\Pr[X_i(\gamma) = 1 \mid \gamma \in P] = 0$. Then, integrating $X_i(\gamma)$ over $[0, 1]^n$, we obtain $\Pr[X_i = 1] = 0$.

Hence, the probability that A will make the robot r_i stay still when the swarm's configuration is picked at random among all $\varepsilon_{\vartheta,n}$-perturbations of S is 1. Since this is true of all robots separately, it is also true of all robots collectively, by the inclusion-exclusion principle. In other words, with probability 1, on a random $\varepsilon_{\vartheta,n}$-perturbation of S, no robot will be able to move, and therefore the robots will be unable to gather. Moreover, with probability 1, a random $\varepsilon_{\vartheta,n}$-perturbation of S is rotationally asymmetric. As a consequence, there is at least one initial configuration (actually, a great deal of configurations) where the swarm forms a rotationally asymmetric set of n distinct points with a connected visibility graph, and where no robot is able to move. We conclude that A cannot be a Gathering algorithm, even under such strong conditions.

Technical hindrances

The probabilistic proof we outlined above is sound for the most part, but unfortunately making it rigorous is a delicate matter. The problem is that, in order for X_i to be a random variable, it has to be a measurable function. For this to be true, the set of coefficients corresponding to perturbations where algorithm A makes the robot r_i stay still should be a measurable subset of $[0, 1]^n$. In turn, this requires some assumptions on the nature of A, whereas we only defined A as a generic function mapping a snapshot to a point.

However, since the function A actually implements an algorithm, which typically is a finite sequence of operations that are well-behaved in an analytic sense, most reasonable assumptions on A would rule out the pathological non-measurable cases, and would therefore make X_i a properly defined random variable, allowing the rest of the proof to go through.

Nonetheless, we choose to adopt a different approach, which is both less technical and more general in scope. Indeed, we will give a "derandomized" version of the above proof, which will not deal with probability spaces and random variables, and will not require a more restrictive re-definition of which functions are computable by mobile robots.

Derandomization

Next we will show how to complete the previous argument without the use of probability. Note that we do not need to prove that a random perturbation causes all robots to stay still with probability 1: we merely have to show that there is at least one perturbation with such a property. This is significantly easier, and is achieved by the next lemma, where X_i no longer denotes a random variable but simply a set of coefficients.

▶ **Lemma 3.6.** *Let $m, n \in \mathbb{N}^+$, and let $X_1, X_2, \ldots, X_n$ be subsets of the unit hypercube $[0,1]^n \subset \mathbb{R}^n$ such that every line parallel to the ith coordinate axis intersects X_i in less than m points, for all $1 \le i \le n$. Then, there is a point in $[0,1]^n$ whose n coordinates are all distinct that does not lie in any of the sets $X_1, X_2, \ldots, X_n$.* ◀

We can now prove the main result of this section.

▶ **Theorem 3.7.** *If $\vartheta \le \pi/2$, and for arbitrarily large n, there is no Gathering algorithm under the condition that the swarm initially forms a rotationally asymmetric set of n distinct points with a connected visibility graph.*

Proof. Let n be an arbitrarily large integer compatible with ϑ, which exists due to Proposition 3.1. Note that all $\varepsilon_{\vartheta,n}$-perturbations of the regular set of size S have a connected visibility graph, by Corollary 3.3. As before, we assume for a contradiction that A is a Gathering algorithm, and we consider a swarm of size n where all robots execute A, and each robot r_i is initially located in the perturbed copy of point $p_i \in S$, for some $\varepsilon_{\vartheta,n}$-perturbation of S.

For $1 \le i \le n$, let $X_i \subseteq [0,1]^n$ be the set of coefficients corresponding to perturbations where algorithm A causes r_i to make a non-null movement. As we already proved, due to Lemmas 3.4 and 3.5, in each bundle of $\varepsilon_{\vartheta,n}$-perturbations of p_i, at most $n-1$ perturbations cause r_i to move. Rephrased in geometric terms, every line in $\mathbb{R}^n$ parallel to the ith coordinate axis intersects X_i in less than n points.

So, the sets $X_1, X_2, \ldots, X_n$ satisfy the hypotheses of Lemma 3.6 with $m = n$. As a consequence, there exists $\gamma = (\gamma_1, \gamma_2, \ldots, \gamma_n) \in [0,1]^n$, where the coefficients γ_i are all distinct, such that, in the perturbation corresponding to γ, algorithm A causes all robots to stay still, and therefore does not allow them to gather.

It remains to check that the perturbation S' corresponding to γ is rotationally asymmetric. Let $p_1, p_2, \ldots, p_n$ be the points of S in clockwise order, and let $p'_i \in S'$ be the perturbed copy of p_i, for $1 \le i \le n$. Suppose for a contradiction that S' has a k-fold rotational symmetry with $k > 1$, implying that the angular distance between p'_1 and $p'_{n/k+1}$ is $\alpha = 2\pi/k$. Note that α is also the angular distance between p_1 and $p_{n/k+1}$. Moreover, by definition of perturbation, $\alpha = cw(p_1, p_{n/k+1}) - cw(p_1, p'_1) + cw(p_{n/k+1}, p'_{n/k+1}) = 2\pi/k - \gamma_1 \cdot \varepsilon_{\vartheta,n} + \gamma_{n/k+1} \cdot \varepsilon_{\vartheta,n}$. It follows that $\gamma_1 = \gamma_{n/k+1}$, which contradicts the fact that the coefficients γ_i are all distinct (indeed, $k \ge 2$ implies that $1 < n/k + 1 \le n$). ◀

We remark that, throughout the proofs of Lemmas 3.4 and 3.5 and Theorem 3.7, only swarms of the same size n appear, and so our impossibility result holds even when n is fixed. It follows that Gathering is impossible even if all robots know the size of the swarm.

4 Gathering algorithm for $\vartheta = \pi$

Overview

In this section we give a Gathering algorithm for robots that can see the entire circle except their antipodal point (i.e., $\vartheta = \pi$), under the condition that the initial configuration is a rotationally asymmetric set with no multiplicity points.

First we will describe a simple Gathering algorithm for robots with full visibility, which already provides some useful ideas: elect a leader, form a unique multiplicity point, and gather there. We will then extend the same ideas to the limited-visibility case with $\vartheta = \pi$. There are some difficulties arising from the fact that not all robots will necessarily agree on the same leader, because they may have different views of the rest of the swarm. For instance, two antipodal robots will not see each other, and, if the configuration is rotationally asymmetric, they will get two non-isometric snapshots, which perhaps will cause them to elect two different leaders.

We will show how to cope with these difficulties. Essentially, based on what a robot r knows, there are only two possibilities on who the "true" leader may be, depending on whether there is a robot antipodal to r or not. If r happens to be elected leader in both scenarios, then r has no doubt of being the leader, and therefore behaves as in the full-visibility algorithm, creating a multiplicity point. In most cases, however, no robot will be so fortunate, but the swarm will still have to make some sort of progress toward gathering. So, the robots that see themselves as possible leaders (but could be wrong) make some preparatory moves that will ideally "strengthen their leadership" in the next turns. We will argue that, after a finite number of turns, one robot will become aware of being the leader and will create a multiplicity point, even under a semi-synchronous scheduler.

The design and analysis of our Gathering algorithm are further complicated by some undesirable special cases, where two distinct multiplicity points end up being created, or the multiplicity point is antipodal to some robot, and therefore invisible to it.

Full visibility and leader election

We will describe a simple Gathering algorithm for the scenario where robots have full visibility.

Let S be a rotationally asymmetric finite set of points on a circle. Recall from Proposition 2.1 that all points of S have distinct angle sequences, and therefore there is a unique point $p \in S$ with the lexicographically smallest angle sequence: p is called the *head* of S.

The Gathering algorithm uses the fact that all robots agree on where the head of the swarm is, and the robot located at the head is elected the leader. The algorithm makes the leader move clockwise to the next robot, while all other robots wait. As soon as there is a multiplicity point, the closest robot in the clockwise direction moves counterclockwise to the multiplicity point. The process continues until all robots have gathered.

Note that this algorithm also works in ASYNCH and with non-rigid robots (i.e., robots that can be stopped by an adversary before reaching their destination). Indeed, as the leader moves toward the next robot, its angle sequence remains the lexicographically smallest, and so it remains the leader. After a multiplicity point has been created, only one robot is allowed to move at any time, and therefore no other multiplicity points are accidentally formed.

Undecided leaders and cognizant leader

Let us now consider a swarm of robots with visibility range $\vartheta = \pi$ forming a rotationally asymmetric set S of n distinct points. We say that the *true leader* of the swarm is the robot located at the head of S.

For each robot r, we define the *visible configuration* $V(r)$ as the set of robots that are visible to r, and the *ghost configuration* as $G(r) = V(r) \cup \{r'\}$, with r' antipodal to r. Note that exactly one between $V(r)$ and $G(r)$ is isometric to the “real” configuration S, and therefore at least one between $V(r)$ and $G(r)$ is rotationally asymmetric.

The *visible head* $v(r)$ is defined as follows: if $V(r)$ is rotationally asymmetric, $v(r)$ is the head of $V(r)$; otherwise, $v(r)$ is the head of $G(r)$. The *ghost head* $g(r)$ is defined similarly: if $G(r)$ is rotationally asymmetric, $g(r)$ is the head of $G(r)$; otherwise, $g(r)$ is the head of $V(r)$. Note that the true leader of the swarm must be $v(r)$ or $g(r)$.

If $v(r) \neq g(r)$, and either $r = v(r)$ or $r = g(r)$, then r is said to be an *undecided leader*: a robot that is possibly the true leader, but does not know for sure. If $r = v(r) = g(r)$, then r is a *cognizant leader*: a robot that is aware of being the true leader of the swarm.

▶ **Proposition 4.1.** *In a rotationally asymmetric swarm with no multiplicity points, the true leader is either an undecided or a cognizant leader, and no robot other than the true leader can be a cognizant leader.* ◀

Point-addition lemma

The following lemma has important implications for the design of our algorithm.

▶ **Lemma 4.2.** *Let S be a finite non-empty set of points on a circle C, and let $S' = S \cup \{p\}$, where $p \in C \setminus S$. Assume that S and S' are rotationally asymmetric, and let $h \in S$ be the head of S and $h' \in S'$ be the head of S'. Then, either $h = h'$ or $cw(h, p) > 2 \cdot cw(h', p)$.* ◀

▶ **Corollary 4.3.** *In a rotationally asymmetric swarm with no multiplicity points, a robot r is a cognizant leader if and only if $r = g(r)$.* ◀

Gathering algorithm

Our Gathering algorithm for $\vartheta = \pi$ is illustrated in Listing 1. A robot r executing the algorithm first checks if the current configuration falls under some special cases (which will be discussed later), and then it attempts to determine the true leader of the swarm. By Corollary 4.3, checking if $r = g(r)$ is equivalent to checking if r is a cognizant leader. In this case, by Proposition 4.1, r is the true leader, and hence it behaves like in the full-visibility algorithm: it moves clockwise to the next robot, s (rule 3: see Figure 1a).

If r is not a cognizant leader, it checks if it is at least an undecided leader: $r = v(r)$. In this case, r cannot commit itself to moving to s, because several robots may be undecided leaders, and this would create more than one multiplicity point. Instead, r attempts to “strengthen its leadership” by moving halfway toward s (rule 4.c: see Figure 1d): this ensures that, in the next turn, r will have a lexicographically smaller angle sequence than it currently has (unless, of course, s moves as well).

Another goal of r is to be able to see the entire swarm in the next turns. Therefore, if the midpoint of r and s happens to be antipodal to some robot q, then r moves a bit further past the midpoint (rule 4.b: see Figure 1c). This way, r will be sure to see q in the next turn (unless, of course, q moves as well).

An exception to the above is when $g(r)$ is antipodal to r, and s has an antipodal robot s'. In this situation, if r had an antipodal robot r', then r' would be the true leader, which would then either form a multiplicity point with s' (if r' is activated) or would become visible to r (if r is activated but not r'). However, if r' does not exist, then r is the true leader, but r may never find out: it may keep approaching s without ever reaching it, and there

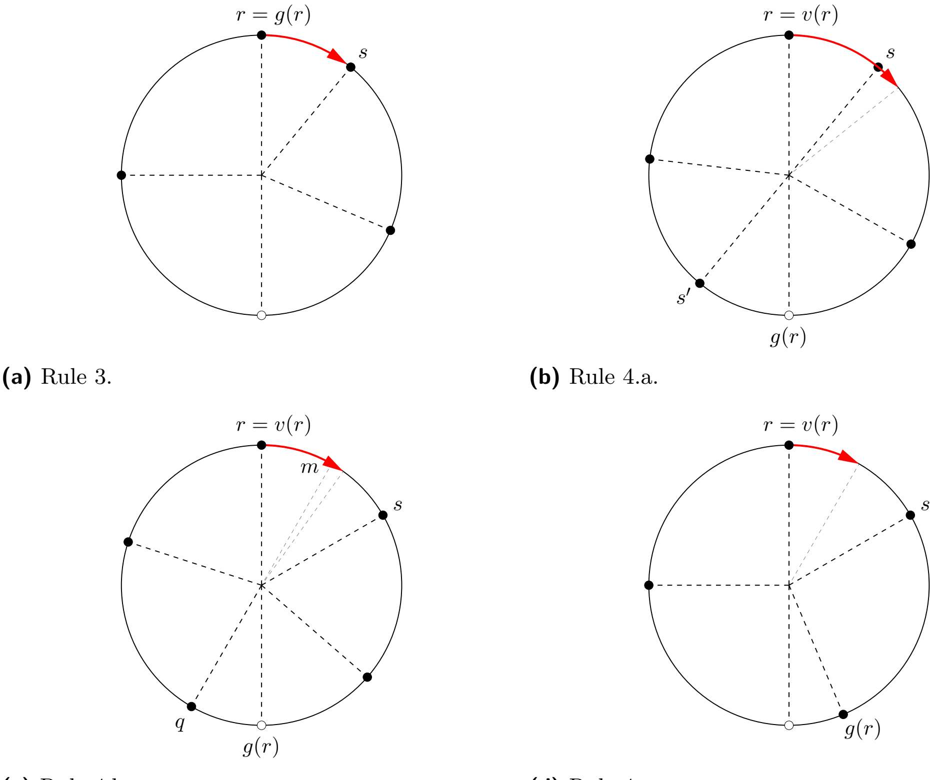
Listing 1 Gathering algorithm for $\vartheta = \pi$.

```
The algorithm is executed by a generic robot r.
Input: V(r), the set of points occupied by robots visible to r
       (expressed in r's coordinate system), with weak multiplicity.
Output: a destination point for r.

Let s ∈ V(r) be such that cw(r,s) > 0 is minimum, if it exists
(s is the visible robot closest to r in the clockwise direction).

Let V'(r) be the set of all the points in V(r) (without multiplicity)
plus their antipodal points.

Let δ be the smallest cw(a,b) > 0 with a,b ∈ V'(r).

 1. If r sees some multiplicity points, then:
   1.a. If r sees a unique multiplicity point, then r moves to it.
   1.b. If r sees two multiplicity points a and b with cw(a,b) > cw(b,a),
        then r moves to a.
 2. Else, if r sees no other robots, then r moves clockwise by π/2.
 3. Else, if r = g(r), then r moves to s.
 4. Else, if r = v(r), then:
   4.a. If g(r) is antipodal to r, and s has an antipodal robot,
        then r moves clockwise by cw(r,s) + δ/3.
   4.b. Else, if there is a point m ∈ V'(r) such that cw(r,m) = cw(r,s)/2,
        then r moves clockwise by cw(r,s)/2 + δ/7.
   4.c. Else, r moves clockwise by cw(r,s)/2.
 5. Else, r does not move.
```

may always be a ghost head antipodal to r. For this reason, if r detects this configuration, it moves slightly past s (rule 4.a: see Figure 1b): this way, s will be the new leader, and it will not be in the same undesirable configuration, because r will not have an antipodal robot.

Note that, when describing rule 4.a and rule 4.b, we mentioned some undefined "small" distances. According to Listing 1, these are respectively $\delta/3$ and $\delta/7$. In turn, δ is defined as the smallest angular distance between two points in $V'(r)$, where $V'(r)$ is the set of all the points in $V(r)$ plus their antipodal points. It is easy to see that all robots that are activated at the same time compute isometric sets V', and therefore they implicitly agree on the value of δ. The reason why the specific values $\delta/3$ and $\delta/7$ have been chosen will become apparent in the proof of correctness of the algorithm *(refer to the full version for details)*.

Finally, let us discuss the special cases, all of which arise when some multiplicity points have been created (due to a cognizant leader moving to some other robot). If only one multiplicity point is visible to r, then r simply moves to it, as in the full-visibility algorithm (rule 1.a). In some exceptional circumstances, two multiplicity points a and b may be created, but we will prove that a and b will not be antipodal to each other, and there will never be a third multiplicity point. In this case, there is an implicit order between a and b on which all robots agree, and so they will all move to the same multiplicity point, say a (rule 1.b). The last special case is when all robots have gathered in a point, except a single robot r located in the antipodal point. r detects this situation because it sees no robots other than itself (and its current location is not a multiplicity point). So, r just moves to another visible point, say, the one forming a clockwise angle of $\pi/2$ with r (rule 2). This ensures that r will see the multiplicity point on its next turn.



(a) Rule 3.

(b) Rule 4.a.

(c) Rule 4.b.

(d) Rule 4.c.

Figure 1 Examples of some of the rules of the Gathering algorithm in Listing 1. Black dots indicate robots that are visible to r. A white dot indicates the point antipodal to r, which may or may not be occupied by a robot.

Correctness

In the following, we will assume that all robots in a swarm of size $n > 1$ execute the algorithm in Listing 1 starting from a rotationally asymmetric initial configuration with no multiplicity points. We will prove that, no matter how the adversarial semi-synchronous scheduler activates them, all robots will eventually gather in a point and no longer move.

We say that, in a given configuration S, a robot r *is able to apply rule* j if, assuming that r is activated when the swarm forms S, r executes rule j (and no other rule).

▶ **Lemma 4.4.** *Assume that the swarm forms a rotationally asymmetric configuration with no multiplicity points, and let ℓ be the true leader. Then:*

1. *No robot is able to apply rule 1 or rule 2.*
2. *At most one robot is able to apply rule 3: the true leader ℓ.*
3. *At most one robot is able to apply rule 4.a: the true leader ℓ (if there is no robot antipodal to ℓ) or the robot antipodal to ℓ.*
4. *A robot $r \neq \ell$ is able to apply rule 4 only if $\pi/2 < cw(r, \ell) \leq \pi$, and only if there is a robot antipodal to r.* ◀

Due to Lemma 4.4, if the swarm forms a rotationally asymmetric configuration with no multiplicity points, we say that a robot *is able to move* if it is able to apply rule 3 or rule 4: indeed, these are the only rules that result in a non-null movement.

▶ **Lemma 4.5.** *In any rotationally asymmetric configuration with no multiplicity points, at most two robots are able to move, and the true leader is always able to move.* ◀

▶ **Lemma 4.6.** *If the swarm has a unique multiplicity point, then all robots eventually gather and no longer move.* ◀

▶ **Lemma 4.7.** *Assume that the swarm forms a rotationally asymmetric configuration with no multiplicity points, and let r and r' be two robots that are able to move. Then:*

1. *If r is activated and executes rule 3, and r' is not activated, then r does not move to r'.*
2. *If r is activated and executes rule 4, then it moves by an angular distance strictly smaller than π, and its destination point is not in $V'(r)$, as defined in Listing 1.*
3. *If both r and r' are activated and execute rule 4.b or rule 4.c, then r and r' are not antipodal of each other, and their destination points are not antipodal of each other.* ◀

▶ **Lemma 4.8.** *If the swarm forms a rotationally asymmetric configuration with no multiplicity points and an active robot executes rule 3 or rule 4.a, then all robots eventually gather in a point and no longer move.* ◀

▶ **Lemma 4.9.** *Assume that, at time t, the swarm forms a rotationally asymmetric configuration with no multiplicity points, and all the robots that move at time t execute rule 4.b or rule 4.c. Then, at time $t+1$, the swarm still forms a rotationally asymmetric configuration with no multiplicity points.* ◀

▶ **Lemma 4.10.** *Assume that, at time t, the swarm forms a rotationally asymmetric configuration with no multiplicity points. If, at all times $t' \geq t$, no robot other than the true leader moves, then all robots eventually gather in a point and no longer move.* ◀

We are now ready to prove the main result of this section.

▶ **Theorem 4.11.** *If $\vartheta = \pi$, there is a Gathering algorithm under the condition that the swarm initially forms a rotationally asymmetric configuration with no multiplicity points.*

Proof. We will prove that the distributed algorithm in Listing 1 solves the Gathering problem under the condition that the swarm initially forms a rotationally asymmetric configuration with no multiplicity points. By the first statement of Lemma 4.4, whenever the swarm forms a rotationally asymmetric configuration with no multiplicity points, any robot that is activated and moves executes either rule 3, or rule 4.a, or rule 4.b, or rule 4.c. In the first two cases, we conclude by Lemma 4.8. In the latter two cases, by Lemma 4.9, the resulting configuration is still rotationally asymmetric and with no multiplicity points. By inductively repeating this argument, we may assume, without loss of generality, that the swarm forms a rotationally asymmetric configuration with no multiplicity points at all times, and all robots that are activated and move execute rule 4.b or rule 4.c.

By Lemma 4.5, at a generic time t, at least one robot r and at most one robot $r' \neq r$ are allowed to move. The semi-synchronous scheduler will activate each of them infinitely often, so let $t' \geq t$ be the first time this happens. Assume that one robot, say r', is not activated at time t', and therefore r is. Then, r does not have an antipodal robot at time $t'+1$, due to the second statement of Lemma 4.7. Similarly, if both r and r' are activated at time t', none of them has an antipodal robot at time $t'+1$, by the second and third statements of Lemma 4.7.

In summary, if a robot is activated and moves at a generic time t, it no longer has antipodal robots at any time after t. Since the robots are finitely many, eventually, say after time t'', only robots without an antipodal robot will move. However, by the fourth statement

of Lemma 4.4, a robot that moves must have an antipodal robot, unless it is the current true leader. So, at all times after t'', no robot other than the current true leader will move. Therefore, Lemma 4.10 allows us to conclude. ◀

References

1 N. Agmon and D. Peleg. Fault-tolerant gathering algorithms for autonomous mobile robots. *SIAM Journal on Computing*, 36(1):56–82, 2006.

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):818–838, 1999.

3 S. Bhagat, K. Mukhopadhyaya, and S. Mukhopadhyaya. Computation under restricted visibility. In *Distributed Computing by Mobile Entities: Current Research in Moving and Computing*, pages 134–183. Springer, 2019.

4 M. Cieliebak, P. Flocchini, G. Prencipe, and N. Santoro. Distributed computing by mobile robots: gathering. *SIAM Journal on Computing*, 41(2):829–879, 2012.

5 P. Courtieu, L. Rieg, S. Tixeuil, and X. Urbain. Impossibility of gathering, a certification. *Information Processing Letters*, 115(3):447–452, 2015.

6 G. D'Angelo, G. Di Stefano, and A. Navarra. Gathering on rings under the Look–Compute–Move model. *Distributed Computing*, 27(4):255–285, 2014.

7 G. D'Angelo, A. Navarra, and N. Nisse. A unified approach for gathering and exclusive searching on rings under weak assumptions. *Distributed Computing*, 30(1):17–48, 2017.

8 S. Das, G.A. Di Luna, P. Flocchini, N. Santoro, G. Viglietta, and M. Yamashita. Oblivious permutations on the plane. In *23rd International Conference on Principles of Distributed Systems*, pages 24:1–24:16, 2019.

9 X. Défago, M. Gradinariu, S. Messika, P. Raipin-Parvédy, and S. Dolev. Fault-tolerant and self-stabilizing mobile robots gathering. In *20th International Symposium on Distributed Computing*, pages 46–60, 2006.

10 B. Degener, B. Kempkes, P. Kling, F. Meyer auf der Heide, P. Pietrzyk, and R. Wattenhofer. A tight runtime bound for synchronous gathering of autonomous robots with limited visibility. In *23rd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 139–148, 2011.

11 G.A. Di Luna, P. Flocchini, L. Pagli, G. Prencipe, N. Santoro, and G. Viglietta. Gathering in dynamic rings. *Theoretical Computer Science*, 811:79–98, 2020.

12 G.A. Di Luna, P. Flocchini, N. Santoro, and G. Viglietta. TuringMobile: a turing machine of oblivious mobile robots with limited visibility and its applications. In *32nd International Symposium on Distributed Computing*, pages 19:1–19:18, 2018.

13 G.A. Di Luna, P. Flocchini, N. Santoro, G. Viglietta, and M. Yamashita. Meeting in a polygon by anonymous oblivious robots. *Distributed Computing (to appear)*, 2019.

14 G.A. Di Luna and G. Viglietta. Robots with lights. In *Distributed Computing by Mobile Entities, Current Research in Moving and Computing*, pages 252–277. Springer, 2019.

15 P. Flocchini. Gathering. In *Distributed Computing by Mobile Entities: Current Research in Moving and Computing*, pages 63–82. Springer, 2019.

16 P. Flocchini, G. Prencipe, and N. Santoro. Self-deployment of mobile sensors on a ring. *Theoretical Computer Science*, 402(1):67–80, 2008.

17 P. Flocchini, G. Prencipe, and N. Santoro. *Distributed Computing by Oblivious Mobile Robots*. Morgan & Claypool, 2012.

18 P. Flocchini, G. Prencipe, and N. Santoro, editors. *Distributed Computing by Mobile Entities, Current Research in Moving and Computing*. Springer, 2019.

19 P. Flocchini, G. Prencipe, N. Santoro, and G. Viglietta. Distributed computing by mobile robots: uniform circle formation. *Distributed Computing*, 30(6):413–457, 2017.

20 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.

21 N. Fujinaga, Y. Yamauchi, S. Kijima, and M. Yamahista. Pattern formation by oblivious asynchronous mobile robots. *SIAM Journal on Computing*, 44(3):740–785, 2015.

22 S. Kamei, A. Lamani, F. Ooshita, S. Tixeuil, and K. Wada. Gathering on rings for myopic asynchronous robots with lights. In *23rd International Conference on Principles of Distributed Systems*, pages 27:1–27:17, 2019.

23 E. Kranakis, D. Krizanc, and E. Markou. *The Mobile Agent Rendezvous Problem in the Ring.* Morgan and Claypool, 2010.

24 A. Monde, Y. Yamauchi, S. Kijima, and M. Yamashita. Self-stabilizing localization of the middle point of a line segment by an oblivious robot with limited visibility. In *19th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, pages 172–186, 2017.

25 L. Pagli, G. Prencipe, and G. Viglietta. Getting close without touching: near-gathering for autonomous mobile robots. *Distributed Computing*, 28(5):333–349, 2015.

26 P. Poudel and G. Sharma. Universally optimal gathering under limited visibility. In *19th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, pages 323–340, 2017.

27 S. Souissi, X. Défago, and M. Yamashita. Using eventually consistent compasses to gather memory-less mobile robots with limited visibility. *ACM Transactions on Autonomous and Adaptive Systems*, 4(1):9:1–9:27, 2009.

28 I. Suzuki and M. Yamashita. Distributed anonymous mobile robots: formation of geometric patterns. *SIAM Journal on Computing*, 28(4):1347–1363, 1999.

29 Y. Yamauchi. Symmetry of anonymous robots. In *Distributed Computing by Mobile Entities, Current Research in Moving and Computing*, pages 109–133. Springer, 2019.

30 Y. Yamauchi, T. Uehara, S. Kijima, and M. Yamashita. Plane formation by synchronous mobile robots in the three-dimensional euclidean space. *Journal of the ACM*, 64(3):16:1–16:43, 2017.

31 Y. Yamauchi and M. Yamashita. Pattern formation by mobile robots with limited visibility. In *20th International Colloquium on Structural Information and Communication Complexity*, pages 201–212, 2013.

Tight Bounds for Deterministic High-Dimensional Grid Exploration

Sebastian Brandt
ETH Zürich, Switzerland
brandts@ethz.ch

Julian Portmann
ETH Zürich, Switzerland
pjulian@ethz.ch

Jara Uitto
Aalto University, Finland
jara.uitto@aalto.fi

Abstract

We study the problem of exploring an oriented grid with autonomous agents governed by finite automata. In the case of a 2-dimensional grid, the question how many agents are required to explore the grid, or equivalently, find a hidden treasure in the grid, is fully understood in both the synchronous and the semi-synchronous setting. For higher dimensions, Dobrev, Narayanan, Opatrny, and Pankratov [ICALP'19] showed very recently that, surprisingly, a (small) constant number of agents suffices to find the treasure, independent of the number of dimensions, thereby disproving a conjecture by Cohen, Emek, Louidor, and Uitto [SODA'17]. Dobrev et al. left as an open question whether their bounds on the number of agents can be improved. We answer this question in the affirmative for deterministic finite automata: we show that 3 synchronous and 4 semi-synchronous agents suffice to explore an n-dimensional grid for any constant n. The bounds are optimal and notably, the matching lower bounds already hold in the 2-dimensional case.

Our techniques can also be used to make progress on other open questions asked by Dobrev et al.: we prove that 4 synchronous and 5 semi-synchronous agents suffice for *polynomial-time* exploration, and we show that, under a natural assumption, 3 synchronous and 4 semi-synchronous agents suffice to explore *unoriented* grids of arbitrary dimension (which, again, is tight).

2012 ACM Subject Classification Computing methodologies → Mobile agents

Keywords and phrases Mobile agents, finite automata, grid search

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.13

Related Version A full version of the paper is available at `https://arxiv.org/abs/2005.12623`.

Funding *Julian Portmann*: Supported by the Swiss National Foundation, under project number200021_184735.

1 Introduction

Grid search by mobile agents is one of the fundamental primitives in swarm robotics and a natural abstraction of foraging behavior of animals. For example in the case of cost-efficient robots or insects, a single agent has relatively limited computation and communication capabilities and hence, many independent agents are required to efficiently solve tasks. To understand such collective problem solving better, knowledge from distributed computing has proven valuable. For instance, Feinerman et al. gave tight bounds on the time complexity of a collective grid search problem inspired by desert ants [17]. In this paper, we focus on the minimum *number* of agents required to solve the grid search problem. A series of papers [7,9,15] nailed down the exact complexity of the 2-dimensional case, that is, discovered the exact number of synchronous/semi-synchronous and deterministic/randomized finite

© Sebastian Brandt, Julian Portmann, and Jara Uitto;
licensed under Creative Commons License CC-BY

automata needed to explore a 2-dimensional grid. However, the approaches in these works do not generalize (well) to higher dimensions. The only known tight bound achieved by such a generalization is obtained by the recent protocol for the deterministic semi-synchronous 3-dimensional setting by Dobrev, Narayanan, Opatrny, and Pankratov [13].

The authors of [13] also gave a more general result: they showed how to implement a stack data structure using only a constant number of agents governed by finite automata. By employing this stack in their search protocols, they show how to explore an n-dimensional grid using only a (small) constant number of agents, for any positive integer n. In particular, the number of agents is independent of the dimension n.

For the case of a 2-dimensional grid the required number of agents is fully understood. However, for higher dimensions there are still gaps between the best upper and lower bounds. Indeed, Dobrev et al. left as open questions the tight complexities of exploring high-dimensional grids in the synchronous/semi-synchronous and deterministic/randomized settings. In this work, we answer these questions for the deterministic setting. Moreover, building on our techniques we make progress on other open questions by Dobrev et al.

1.1 Results and Techniques

Similarly to the approach by Dobrev et al. [13], our search protocols rely on an efficient implementation of a stack data structure. One agent is dedicated to do the actual search while the remaining agents implement a stack (together with the searching agent that indicates the base of the stack) with their positions on the grid. On a high level, the size of the stack encodes the cell the searching agent is supposed to explore next, relative to the current position of the searching agent. Both our protocol and the protocol from [13] explore the grid by repeatedly reading the stack, moving the searching agent to the cell indicated by the stack, moving the searching agent back to its original cell, and incrementing the stack. The difficult part is to be able to effectively read the stack (without destroying the stack in the process) despite the fact that the size of the stack grows arbitrarily far beyond the number of states in the finite automaton reading the stack. The authors of [13] managed to implement this data structure using 4 agents in the synchronous and 5 agents in the semi-synchronous setting and showed how to explore oriented grids with as many agents.

One of our main contributions is to implement this stack and the operations required for reading it with only 3 (synchronous) agents (including the searching agent), which is optimal given the grid exploration lower bound by Emek et al. [15]. We achieve this by an encoding scheme that transforms the location of a cell to be explored into a single integer (that can be represented by the stack size) by interpreting the coordinates of the cell (relative to the current location of the searching agent) as exponents of distinct prime factors. This scheme is also known as Gödel's encoding. One crucial advantage of this specific encoding is that there is a way to read the stack (using 3 synchronous agents), i.e., to repeatedly provide the searching agent with different parts of the encoded information, that does not destroy the encoded information, but instead changes the encoding slightly: replacing the base (prime) for one of those exponents by a different prime (and then switching to the next base prime). The technical details why such replacement operations can be performed by 3 synchronous agents and why they allow the searching agent to obtain the desired information are covered in Sections 3 and 4. Moreover, by adding one agent as a synchronizer, the protocol can be made to work in the semi-synchronous setting.

▶ **Theorem 1.** *For any positive integer n, the n-dimensional (oriented) grid can be explored by* 3 *synchronous finite automata, resp.* 4 *semi-synchronous finite automata.*

Unoriented Grids. An underlying assumption of the setting considered so far is that the agents are aware of the $2n$ cardinal directions, i.e., they know for each of the n dimensions of the grid which two adjacent cells are neighbors in that dimension, and each dimension is oriented. Or, to put it simply, the agents know which directions are north, south, etc.; in particular the directions are globally consistent. In contrast, in the *unoriented* setting considered in [13], each cell is endowed with a labeling that indicates for each cell which neighbor is north, south, etc. (and for each of the $2n$ directions there is exactly one neighbor), but there is no consistency guarantee between the directions indicated by the labels of different cells, e.g. by going north twice, an agent could end up in the cell it started.

In their work, Dobrev et al. also ask "How many additional agents are necessary to solve the problem in *unoriented grids*?". We show, perhaps surprisingly, that the unoriented case is no harder than the oriented case given the following (natural) assumption: If we follow some fixed direction, we never end up back in the same cell where we started.

▶ **Theorem 2** (Simplified). *Under a natural assumption, for any positive integer n,* 3 *synchronous finite automata, resp.* 4 *semi-synchronous finite automata, suffice to explore any n-dimensional unoriented grid.*

The key idea to obtain Theorem 2 is that, even without a globally consistent orientation, we can implement a (virtual) stack. Due to the missing consistency, the same cell may occur repeatedly in the stack, but we can show that we can bound the number of occurrences for each cell and that the agents can distinguish between the different occurrences of the same cell. In essence, we will show that the stack corresponds to (a part of) a DFS exploration of an infinite tree consisting of those edges (between cells) that point north.

Polynomial Time Protocol. The task of exploring the entire grid is equivalent to finding a treasure located at some distance D from the starting point. This allows us to discuss the *efficiency* of a protocol, i.e., its runtime with respect to D. We observe that our encoding scheme for the oriented grid using only 3 synchronous, resp. 4 semi-synchronous, agents might result in exponential time. However, with one additional agent, certain useful stack operations can be extended to work for non-constant values. This allows us to use a different exploration scheme, proposed by Dobrev et al. [13], resulting in a polynomial runtime.

▶ **Theorem 3.** *For any positive integer n, the n-dimensional (oriented) grid can be explored by: (1) 4 synchronous agents in time $O(V(D)^2)$, and (2) 5 semi-synchronous agents in time $O(V(D)^3)$, where $V(D) = \Theta(D^n)$ is the volume of the ℓ_1-ball of radius D.*

Due to space constraints, our discussion of polynomial-time exploration, as well as the proofs of all lemmas and theorems are deferred to the full version.

1.2 Further Related Work

In a typical graph exploration setting, we are given a graph where initially, one or more mobile agents are placed on some vertices of the graph. The agents are able to traverse along the edges and their goal is to *explore* the graph, that is, visit every node or edge of the graph. Equivalently, one can think of searching for a treasure hidden on an edge or a node of the graph. Graph exploration has been widely studied in the literature (see, for example, [1,10,11,18,21,22]) and it comes in many variants.

A classic setting is the *cow-path* problem, where a single agent, the cow, is searching for adversarially hidden food on a path [3,4]. The goal for the cow is to minimize the number of edge-traversals until the food is found. It is known that a simple spiral search is optimal

and this algorithm also generalizes to the case of grids. This problem was also studied in the case of many cows [20]. Closely related to our work is the exploration of *labyrinths*, i.e., 2-dimensional grids where some cells are blocked [8]. It is known that two finite automata or one automaton with two pebbles (movable marker) suffice for co-finite labyrinths, where a finite amount of cells are not blocked [6]. Finite labyrinths, where a finite amount cells are blocked, can be explored with one automaton and four pebbles, whereas one automaton and one pebble is not enough [5, 19]. An agent with $\Theta(\log\log n)$ pebbles can explore all graphs and this bound is tight [12].

In the case of many agents, the agents typically operate in *look-compute-move* cycles. First, the agents simultaneously take a local snapshot, then decide on the next operation, and finally, execute the operation. Graph exploration can be divided into *synchronous* ($\mathcal{FSYNC}$), *semi-synchronous* ($\mathcal{SSYNC}$), and *asynchronous* ($\mathcal{ASYNC}$) variants [23–25]. In the $\mathcal{FSYNC}$ setting, the execution is divided into synchronous rounds, where in every round, every agent executes one cycle. The execution in $\mathcal{SSYNC}$ consists of discrete time steps, where in each step, a subset of the agents executes one atomic cycle. In the $\mathcal{ASYNC}$ setting, the cycles are not (necessarily) atomic. In this paper, we study the $\mathcal{FSYNC}$ and the $\mathcal{SSYNC}$ settings.

For finite graphs, a *random walk* provides a simple algorithm that explores the graph in polynomial time [2]. In the case of an infinite n-dimensional grid, a random walk finds the treasure with probability 1 if $n \leq 2$. However, the expected hitting time, i.e., the time to find the treasure, is infinite. Cohen et al. showed that even for the case of two (collaborating) randomized agents governed by finite automata, one cannot achieve any finite hitting time for $n \geq 2$ [9]. Very recently, Dobrev et al. showed that 3 randomized $\mathcal{FSYNC}$ and 4 randomized $\mathcal{SSYNC}$ agents suffice to achieve a finite hitting time for any n [13]. In this work, we achieve the same bounds with deterministic agents.

This work follows a series of papers inspired the work by Feinerman et al., where they studied the time it takes to find a treasure in a 2-dimensional grid by k non-communicating agents governed by Turing machines [17]. They showed that the time complexity of this task is $\Theta(D^2/k + D)$, where D is the distance from the origin to the treasure. This bound can be matched by finite automata that are allowed to communicate within the same cell [16]. Emek et al. asked what is the minimum number of finite automata agents required to find the treasure [15]. They showed that at least 3 synchronous deterministic agents are required and that 3 synchronous deterministic, 4 semi-synchronous deterministic, and 3 semi-synchronous randomized agents are enough. Cohen et al. [9] and Brandt et al. [7] showed the matching lower bounds for the randomized and deterministic semi-synchronous cases, respectively.

2 Preliminaries

Grids. We consider the problem of exploring the infinite n-dimensional grid, whose vertices are the elements of $\mathbb{Z}^n$, which we refer to as *cells*. A cell $c = (c_1, \ldots, c_i, \ldots, c_n)$ is described by its coordinates and two cells c and c' are connected by an edge if there is a dimension i such that $|c_i - c'_i| = 1$ and $c_j = c'_j$ for $j \neq i$. When talking about distance, we will use the ℓ_1 or *Manhattan distance*, which is defined as $d(c, c') = \sum_i |c_i - c'_i|$.

In the *oriented* case, we assume that there is a consistent labeling of the edges by both of its endpoints, which in the 2-dimensional case can be thought of as the directions of a compass: north, south, east, and west. In general, an edge (c, c') is labeled by $(+1, i)$ from the side of c (and thus $(-1, i)$ from the side of c') if we have that $c_i + 1 = c'_i$.

For *unoriented* grids, we assume that each endpoint of an edge has a label from $\{1, \ldots, 2n\}$. We will also refer to these labels as the *ports* of a cell. The only assumption we make is that the labels around each cell are pairwise distinct, i.e., each cell has every port from 1 to $2n$ exactly once. Thus, each edge can receive any pair of labels from $\{1, \ldots, 2n\}$.

Exploration. The exploration is performed by m agents, $a_1, \ldots, a_m$, which are initially all placed in the same cell, called the *origin*. W.l.o.g. we assume the origin to have coordinates $(0, \ldots, 0)$. The agents cannot distinguish different cells (including the origin); in particular, they do not know the coordinates of the cell they are in. Their behavior and movement is controlled by a deterministic finite automaton. While we require all agents to use the same automaton, they may start in different initial states. (As we only consider protocols with constantly many agents, one can equivalently assume each agent to be controlled by an individual automaton, as we can combine m automata into one by using disjoint state spaces.) Agents can only communicate if they are in the same cell: each agent senses the states for which there is an agent that occupies the same cell, and performs its next move and state transition based on this information. For oriented grids, such a move is described by a direction and dimension to move in.

In the case of unoriented grids, we assume that agents can also see both labels of each incident edge, and perform their decisions based on this information as well. A move is then described by choosing a port of the current cell and moving along this edge. Previous work by Dobrev et al. [13] used an essentially equivalent definition: Each agent could only see the label on its side of each incident edge, but once it arrived in the new cell by traversing some edge, it would obtain the information about the second label on the edge it traversed. We choose to formalize the model in a slightly different way, as it will simplify the description of our algorithms. However, we emphasize that for our purposes, the two models can be used interchangeably since within $2n$ steps in the model of Dobrev et al., the agents can learn all information that we assume the agents can immediately see.

Formally, we have a state space Q, a transition function δ, and an initial state q_i^0 for every agent a_i. For oriented grids, the transition function has the form: $\delta : Q \times 2^Q \to Q \times (\{-1, +1\} \times \{0, 1, \ldots, n\})$. The function maps an agent in state $q \in Q$, which observes the set of states for which there is an agent occupying the same cell, to a new state $q' \in Q$ and a movement, which is described by the direction (-1 or $+1$) and the dimension (from 1 to n) along which the agent moves to the respective neighboring cell, where an agent can also choose to *stay* in the same cell which is described by dimension 0. We will say that an agent moves *north* if its movement is $(+1, 1)$, and *south* if it is $(-1, 1)$.

For unoriented grids, we change the definition of the transition function slightly to $\delta : Q \times 2^Q \times \{1, \ldots, 2n\}^{2n} \to Q \times (\{-1, +1\} \times \{0, 1, \ldots, n\})$. The function maps an agent in state $q \in Q$, which observes both the set of states for which there is an agent occupying the same cell, and, for each port, the other label on the edge corresponding to that port, to a new state $q' \in Q$ and a movement, which is specified by the port via which the agent leaves the current cell, or 0, in which case the agent does not move.

The Schedule. Time is divided into discrete units, where in each time step, a set of *active* agents performs a *look-compute-move* cycle. First, an agent senses the states of all agents in the same cell (and in the case of unoriented grids both of the labels on all incident edges), then it applies the transition function to its own state and all sensed information, and finally it changes its state and moves as indicated by the result. We assume that one such cycle is atomic, i.e., cycles that start at different times do not overlap.

For the *synchronous* or $\mathcal{FSYNC}$ model, we assume that all agents are active at every time step. We call the system *semi-synchronous*, or the $\mathcal{SSYNC}$ variant, if at every time step only a subset of agents, chosen by an adversary, is active. While the adversary knows all information about the agents and their behavior, it must schedule each agent infinitely often, to avoid trivial impossibilities.

Exploration Cost. Finally, if we discuss the efficiency of a protocol, we consider the following problem, which is equivalent to exploring the grid: the agents are tasked to find a treasure, which is hidden at some distance D from the origin (without the agents knowing the value of D). This enables us to measure the time or *exploration cost* it takes to find the treasure with respect to D. In the synchronous setting, we measure the exploration cost as the number of time steps needed for an agent to arrive at the cell containing the treasure. As, in the semi-synchronous model, this number of steps depends on the schedule, we instead define the exploration cost as the total distance traveled by all agents in this setting.

3 Building Blocks

Encoding Information as a Stack. Dobrev et al. [13] introduced the idea of using multiple agents to implement a *stack*. In its simplest form, a stack is just a pair of agents, whose distance encodes some information. However, to allow for manipulations of the stack, more agents are needed. Our protocol for exploring n-dimensional grids with 3 synchronous, resp. 4 semi-synchronous, agents will consist of subroutines that involve manipulations of the stack. The relevant parameter will be the *stack size*, denoted by X, which is defined as the distance between the *base* of the stack and the *end* of the stack. The base of the stack is the location of agent a_1, and the end of the stack is the location of the other agents. We will only be interested in the stack and its size at the very beginning and very end of each subroutine; at these points in time all agents except a_1 are guaranteed to be in the same cell, and this cell is guaranteed to be reachable from the cell containing a_1 by going repeatedly north, making the notion of a stack well-defined. Whenever we refer to the base, end, or size of the stack *during* some subroutine, we mean the respective notion at the beginning of the subroutine.

In this section, we will describe the subroutines that form the building blocks of our exploration algorithm. Moreover, we will show for both the synchronous and the semi-synchronous setting how to implement the subroutines with the desired number of agents.

In [13], the authors show how to multiply the current stack size by 2, resp. divide it by 2, using 3 synchronous agents. This also provides a way to check whether the current stack size is divisible by 2. The idea behind the implementation is simple: while agent a_1 stays at the base of the stack, the other two agents, initially located at the end of the current stack, move with different speeds[1], a_3 either away from or towards the base of the stack, and a_2 first towards the base, and then reversing direction when the base is reached. The operation is completed when a_2 and a_3 meet again (after a_2 visited the base). By choosing a speed of 1 for a_2, and a speed of $1/3$ for a_3, and letting move a_3 *towards* the base, we achieve that the stack size is halved; by choosing the same speeds and letting move a_3 *away from* the base, we achieve that the stack size is doubled.

We will need similar subroutines as building blocks for our synchronous and semi-synchronous protocols. More precisely, given a positive integer $k \geq 2$, we want the agents to be able to perform the following operations.

- MultiplyStackSize(k): Multiply the stack size by k.
- IsDivisible(k): Check whether the current stack size is divisible by k.
- DivideStackSize(k): If the stack size is divisible by k, divide the stack size by k.

We will only require the agents to be able to perform these operations for constantly many k, where the constant depends (only) on the dimension n of the grid.

[1] An agent moves with speed $1/j$ in some direction if it repeatedly performs the following behavior: first it takes one step in the chosen direction, and then it waits for $j-1$ steps. Note that our speed of $1/j$ is the same as speed j in [13].

To implement these operations, we simply adapt the protocols for the case $k = 2$ from [13] by choosing the speeds of $1/(k-1)$ (instead of 1) for a_2 and $1/(k+1)$ (instead of $1/3$) for a_3. More precisely, we implement the desired operations using 3 synchronous agents as follows.

MultiplyStackSize(k). While it is usually easier to understand the behavior of an agent if it is described without specifying the exact states and the transition function, we will provide the latter for subroutine MultiplyStackSize(k) to give an example how to translate the agents' behaviors described in this work into the formal specification of a finite automaton. Let $k \geq 2$ be a positive integer. As usual we assume that a_2 and a_3 are in the same cell c', and a_1 is in a cell $c \neq c'$ such that c' can be reached from c by going north repeatedly (i.e., c and c' differ only in the first coordinate, and c has a smaller first coordinate than c').

In subroutine MultiplyStackSize(k), we denote the starting state of each agent a_i by $\textsc{Mult}^0_{i,k}$. Apart from state $\textsc{Mult}^0_{i,k}$, we will use $2k-2$ other states for agent a_2, denoted by $\textsc{Mult}^1_{2,k}, \ldots, \textsc{Mult}^{k-2}_{2,k}$, $\textsc{MultBack}^0_{2,k}, \ldots, \textsc{MultBack}^{k-2}_{2,k}$ and $\textsc{Mult}^{\text{fin}}_{2,k}$, and $k+1$ other states for agent a_3, denoted by $\textsc{Mult}^1_{3,k}, \ldots, \textsc{Mult}^k_{3,k}$, and $\textsc{Mult}^{\text{fin}}_{3,k}$. Agent a_1 always stays in state $\textsc{Mult}^0_{1,k}$ and cell c. Agents a_2 moves and changes its state according to the following rules, where "stay" indicates that the agents does not move to another cell.

$$
\begin{aligned}
(\textsc{Mult}^0_{2,k}, S) &\to (\textsc{Mult}^1_{2,k}, \text{south}) && \text{for any } S \in 2^Q \text{ satisfying } \textsc{Mult}^0_{1,k} \notin S \\
(\textsc{Mult}^0_{2,k}, S) &\to (\textsc{MultBack}^1_{2,k}, \text{north}) && \text{for any } S \in 2^Q \text{ satisfying } \textsc{Mult}^0_{1,k} \in S \\
(\textsc{Mult}^j_{2,k}, S) &\to (\textsc{Mult}^{j+1}_{2,k}, \text{stay}) && \text{for any } 1 \leq j \leq k-3 \text{ and any } S \in 2^Q \\
(\textsc{Mult}^{k-2}_{2,k}, S) &\to (\textsc{Mult}^0_{2,k}, \text{stay}) && \text{for any } S \in 2^Q \\
(\textsc{MultBack}^0_{2,k}, S) &\to (\textsc{MultBack}^1_{2,k}, \text{north}) && \text{for any } S \in 2^Q \text{ satisfying } \textsc{Mult}^0_{3,k} \notin S \\
(\textsc{MultBack}^0_{2,k}, S) &\to (\textsc{Mult}^{\text{fin}}_{2,k}, \text{stay}) && \text{for any } S \in 2^Q \text{ satisfying } \textsc{Mult}^0_{3,k} \in S \\
(\textsc{MultBack}^j_{2,k}, S) &\to (\textsc{MultBack}^{j+1}_{2,k}, \text{stay}) && \text{for any } 1 \leq j \leq k-3 \text{ and any } S \in 2^Q \\
(\textsc{MultBack}^{k-2}_{2,k}, S) &\to (\textsc{MultBack}^0_{2,k}, \text{stay}) && \text{for any } S \in 2^Q
\end{aligned}
$$

For agent a_3, the rules are as follows.

$$
\begin{aligned}
(\textsc{Mult}^0_{3,k}, S) &\to (\textsc{Mult}^1_{3,k}, \text{north}) && \text{for any } S \in 2^Q \text{ satisfying } \textsc{MultBack}^0_{2,k} \notin S \\
(\textsc{Mult}^0_{3,k}, S) &\to (\textsc{Mult}^{\text{fin}}_{3,k}, \text{stay}) && \text{for any } S \in 2^Q \text{ satisfying } \textsc{MultBack}^0_{2,k} \in S \\
(\textsc{Mult}^j_{3,k}, S) &\to (\textsc{Mult}^{j+1}_{3,k}, \text{stay}) && \text{for any } 1 \leq j \leq k-1 \text{ and any } S \in 2^Q \\
(\textsc{Mult}^k_{3,k}, S) &\to (\textsc{Mult}^0_{3,k}, \text{stay}) && \text{for any } S \in 2^Q
\end{aligned}
$$

The protocol terminates when both a_2 and a_3 are in states $\textsc{Mult}^{\text{fin}}_{2,k}$ and $\textsc{Mult}^{\text{fin}}_{3,k}$, respectively. The design of the protocol (in particular, of the two rules leading to the two terminal states) ensures that a_2 and a_3 terminate at the same point in time. As the rules of the protocol specify that a_2 walks with speed exactly $1/(k-1)$, and a_3 with speed exactly $1/(k+1)$, we see that the first time a_2 and a_3 are in the same cell in states $\textsc{MultBack}^0_{2,k}$, resp. $\textsc{Mult}^0_{3,k}$ (which is the configuration leading to termination in the next step), they are in a cell in distance kX from the base of the stack. The meeting happens after a_2 traversed $(k+1) \cdot X$ cells (X towards the base, kX away from the base), whereas a_3 traversed $(k-1) \cdot X$ cells.

DivideStackSize(k). Analogously, we can implement division by k by letting a_3 walk *towards* the base, instead of away from the base, i.e., by replacing the first rule for a_3 by

$$
(\textsc{Mult}^0_{3,k}, S) \to (\textsc{Mult}^1_{3,k}, \text{south}) \quad \text{for any } S \in 2^Q \text{ satisfying } \textsc{MultBack}^0_{2,k} \notin S
$$

while leaving all other rules (for all agents) unchanged. However, the two rules leading to the terminal states require a_2 and a_3 to be in states $\text{MULTBACK}^0_{2,k}$ and $\text{MULT}^0_{3,k}$, respectively, to ensure termination. If the initial stack size X is divisible by k, then the states of the two agents will align perfectly in the cell c'' in distance X/k from the base of the stack: after $(k-1)(k+1)$ time steps, a_2 has traversed $k+1$ cells with speed $1/(k-1)$, and a_3 has traversed $k-1$ cells with speed $1/(k+1)$, hence both are in cell c'' in the states leading to the terminal states. If, however, X is not divisible by k, then the states of the two agents do not align when they meet again after a_2 visited the base.

IsDivisible(k). Hence, before dividing by k, we will always check whether the current stack size is divisible by k. This can be achieved by having a_2 walk towards the base with speed 1 while increasing a counter modulo k each time it takes a step. If the counter is at 0 when a_2 reaches a_1, the stack size is divisible by k; if not, then the stack size is not divisible by k. The subroutine of checking for divisibility by k terminates after a_2 has walked back to a_3 and informed it whether the current stack size is divisible by k or not.

Further Building Blocks. In order to be able to write our synchronous exploration protocol concisely, it will be useful to define a few other subroutines. As before, we will assume that, in the beginning of the subroutines, agents a_2 and a_3 will be in the same cell c', representing the end of the stack, and a_1 is in a cell c representing the base of the stack that differs from c' only in that its coordinate in dimension 1 is strictly smaller. The only exception will be the subroutine $\text{INITIALIZESTACKSIZE}(k)$ that initializes the stack to some positive integer k by having a_2 and a_3 walk k steps away from a_1 – here, all three agents are initially in the same cell. Apart from $\text{INITIALIZESTACKSIZE}(k)$, we define the subroutines $\text{INCREASESTACKSIZE}(k)$ for positive integers k, and $\text{MOVESTACK}(g, i)$, where $g \in \{-1, 1\}$ and $i \in \{1, \ldots, n\}$. Subroutine $\text{INCREASESTACKSIZE}(k)$ simply increases the stack size by k (additively) by having a_2 and a_3 walk k steps away from a_1.

A subroutine similar to our $\text{MOVESTACK}(g, i)$ was already introduced in [13]. The purpose of this subroutine is to move the whole stack in some direction specified by dimension i and sign g. In our definition, $\text{MOVESTACK}(g, i)$ moves every agent to a new cell that differs from the old cell only by having its ith coordinate increased by g, i.e., effectively each agent takes one step in dimension i. However, one has to be a bit careful when implementing this subroutine as we want to be able to concatenate it with other subroutines. In particular, in all other subroutines, agent a_1 does not know when the subroutine is started or terminates, while the other agents do know. In order to also obtain this property for $\text{MOVESTACK}(g, i)$, we implement the desired movement by having a_2 walk towards a_1, notifying it about the desired step and the chosen direction (upon which a_1 performs the step) and then returning to a_3, where both a_2 and a_3 perform the desired step as well.

Semi-Synchronous Agents. All of the above subroutines can also be performed by (at most) 4 semi-synchronous agents, as we show in the following. Similar to the approach in [13], we will use one agent (a_4) to effectively synchronize the behavior of the other agents, which allows us to essentially execute the 3-agent synchronous subroutines described above with the remaining 3 agents. In more detail, agent a_4 will visit the other agents in a suitable order, and each of the other agents will only move when they are in the same cell as a_4 (while a_4 will not leave the cell of the agent it wants to move next until the agent actually left the cell). We start by showing how this can be achieved for subroutine $\text{MULTIPLYSTACKSIZE}(k)$.

As in the synchronous version of the subroutine, we would like the two agents a_2 and a_3 to move with (relative) speeds $1/(k-1)$ (first towards a_1 and, after meeting a_1, away from a_1) and $1/(k+1)$ (away from a_1), respectively, while a_1 simply stays at the base of the stack. The purpose of this design – that when a_2 and a_3 meet next, they are in a cell that has the k-fold distance to a_1 as they have currently – can also be achieved by having a_3 move $k-1$ steps, then having a_2 move $k+1$ steps, and so on, always alternating between the two agents, until they are both in the same cell again (which, by their relative "speeds" must have the desired distance to the base of the stack). This behavior can be ensured by using a_4:

Agents a_2 and a_3 follow their designated route, but they only take one step of those routes if they are in the same cell as a_4 and a_4 is in a state indicating that a_2, resp. a_3 should move (the latter condition is not strictly necessary, but simplifies things by ensuring that a_2 and a_3 never move at the same time). Agent a_4 alternates between visiting a_2 and a_3, during each "visit" making sure that the respective agent takes the desired number of steps (i.e., $k-1$ or $k+1$). It does so by going to the cell of the respective agent a_i ($i \in \{2,3\}$), indicating that a_i should take a step of its route, waiting until a_i takes a step and leaves the cell, incrementing an internal counter, following agent a_i to the next cell, and repeating this behavior until the counter indicates that the desired number of steps has been taken by a_i, upon which a_4 visits the other agent a_{5-i}. Note that a_4 always knows in which direction it has to move to find the desired agent as the coordinates of a_2 and a_3 only differ in dimension 1, and a_2 always has a smaller (or equally large) first coordinate. Moreover, a_4 also knows in which direction it has to go to follow the agents to the next cell as the only change in direction is performed by a_2 and the reason for the change, namely meeting a_1, is an information known to a_4 since when a_2 meets a_1, it stays in the cell containing a_1 until a_4 also arrives there.

In an analogous fashion, we can implement DivideStackSize(k) with 4 semi-synchronous agents. For the other four subroutines, the picture is even simpler: it is straightforward to check that these subroutines can already be implemented by 3 semi-synchronous agents by having the agents perform the same steps as in the respective synchronous subroutines. The reason that these subroutines also work in the semi-synchronous setting is that either the synchronous version already contain one agent that effectively acts as a synchronizer (in the sense that every action is performed by that agent or directly instigated by a visit of that agent), as in IsDivisible(k) and MoveStack(g, i), or the actions of the agents are independent of each other, as in InitializeStackSize(k) and IncreaseStackSize(k). For these four subroutines, we will simply assume that a_4 is treated the same as a_3; in particular, at the beginning and end of each subroutine, we will always have a_2, a_3, and a_4 in the same cell, indicating the end of the stack.

We have to be careful with the termination of each subroutine as we want to be able to concatenate the subroutines. To this end, we will again use a_4 as a synchronizer: before terminating itself, a_4 will wait that a_2 and a_3 (which are in the same cell at the end of each subroutine) have terminated. Similarly, we can assume that a_4 will initialize the next subroutine by changing its state suitably, thereby making sure that the start and end of the subroutines align across all agents. A last detail is that in the semi-synchronous version of MoveStack(g, i) (which is the only subroutine where a_1 moves), after meeting a_1, agent a_2 has to wait until a_1 takes its step before moving back to a_3 and a_4, in order to make sure that a_4 does not terminate and initialize the next subroutine before a_1 takes its step.

4 The Exploration Protocol

In this section, we will combine the building blocks of Section 3 to a protocol that allows 3 synchronous, resp. 4 semi-synchronous, agents to explore the n-dimensional (oriented) grid, and prove the protocol's viability. Our protocol is given by algorithm Explore.

Algorithm 1 Explore.

```
1: InitializeStackSize(3)
2: repeat
3:     for each function g : {1, ..., n} → {−1, 1} do
4:         FollowRoute(g)
5:         FollowRoute(−g)
6:     end for
7:     IncreaseStackSize(2)
8: procedure FollowRoute(g)
9:     for i = 1 to n do
10:        while IsDivisible(p_i) do
11:            DivideStackSize(p_i)
12:            MultiplyStackSize(2)
13:            MoveStack(g(i), i)
14:        end while
15:        while IsDivisible(2) do
16:            DivideStackSize(2)
17:            MultiplyStackSize(p_i)
18:        end while
19:    end for
20: end procedure
```

The underlying idea of algorithm Explore is the same as in the algorithms from [13]: We generate each (non-zero) n-dimensional vector $(v_1, \dots, v_n)$ with non-negative integer coordinates, and for each such vector, we let one agent walk from the origin to each cell $(c_1, \dots, c_n)$ such that $c_i \in \{v_i, -v_i\}$ for all $1 \leq i \leq n$, and then back to the origin. More precisely, in each execution of FollowRoute(g), agent a_1 walks to the respectively specified cell $(c_1, \dots, c_n)$, and in each execution of FollowRoute($-g$), a_1 walks back to the origin. To generate $(v_1, \dots, v_n)$, a counter, represented by the stack size, is used that is incremented gradually, thereby iterating through the positive integers. Each time the counter is incremented, the new value X will be transformed into some n-dimensional vector, $(v_1, \dots, v_n)$, where the design of the transformation has to make sure that every (non-zero) vector with non-negative integer coordinates is generated by some value X. We require the stack size to be odd, as we will be using powers of 2 to store some intermediate values. Thus, we will always increase the counter by 2, while still ensuring that all vectors are generated.

However, as we have one fewer agent available than in the protocols in [13], our protocol requires a new way to implement this idea. In particular, we avoid using a separate agent to remember the stack size when the stack is read, instead making sure that even after the stack is read, no information about the previous stack size(s) is lost[2]. To this end, we define the vector $(v_1, \dots, v_n)$ we want to transform X into as follows. Let $p_1, \dots, p_n$ denote the first n odd primes, where $p_1 < \dots < p_n$. For all $1 \leq i \leq n$, we define v_i to be the largest non-negative integer such that $p_i^{v_i}$ divides X. In other words, v_i represents how often p_i occurs as a prime factor of X.

[2] Note that such information is still required after reading the stack: we will need it both to guide a_1 back to the origin and to retrieve the counter value X that we want to increase repeatedly.

Consider procedure FollowRoute(g). The for loop of this procedure iterates through the n dimensions. For each dimension i, the first while loop repeatedly replaces one prime factor p_i by prime factor 2, by dividing by p_i and multiplying by 2. Each time such a replacement is performed, the whole stack is moved one cell w.r.t. dimension i (either increasing or decreasing the respective coordinate by 1, depending on the value of $g(i)$). After all (i.e., v_i) occurrences of p_i as prime factors have been replaced by factors 2, the stack manipulations are reversed in the second while loop, resulting in the original stack size X. Note that in the very beginning of algorithm Explore, the stack size is initialized to 3, and each time the counter represented by the stack size is increased, it is increased by 2; hence, before starting the first while loop, the stack size is odd, ensuring that the second while loop goes through exactly the same number of iterations as the first one. Note further that we do not revert the steps that a_1 took (yet) when reversing the stack manipulations. After iterating through all dimensions, agent a_1 is now in cell $(c_1, \ldots, c_n)$, and we can consider this cell as explored, concluding the execution of FollowRoute(g).

The execution of FollowRoute($-g$) is identical to the execution of FollowRoute(g), except that each step of a_1 is performed in the opposite direction. Hence, at the end of the execution of FollowRoute($-g$), agent a_1 is back at the origin, while the stack size is (again) X. The (outer) for loop in algorithm Explore simply iterates through all possible assignments of signs $\in \{-1, +1\}$ to the dimensions, making sure that for each generated vector $(v_1, \ldots v_n)$, each corresponding cell $(v'_1, \ldots, v'_n)$ is explored.

In the following, we state our main result for oriented grids. The proof is given in the full version of this work.

▶ **Theorem 1.** *For any positive integer n, the n-dimensional (oriented) grid can be explored by 3 synchronous finite automata, resp. 4 semi-synchronous finite automata.*

5 Unoriented Grids

In [13], the authors showed that any protocol for the oriented grid can be transformed into a protocol for unoriented grids by adding sufficiently many agents such that, at all times, each original agent moving across a non-constant distance is accompanied by one of the additional agents. In particular, for both their protocol and our improved protocol, this implies that 2 additional agents are required in the synchronous case and 1 additional agent in the semi-synchronous case (since in the protocols for the oriented grid, 2 synchronous agents are traversing non-constant distances at the same time, while in the semi-synchronous case only 1 agent does so). Hence, our protocol for the oriented grid improves also the state of the art for the minimum number of required agents on *unoriented* grids from 6 to 5 (in both the synchronous and the semi-synchronous setting).

On an informal level, it seems unlikely that our upper bound of 5 can be improved since intuitively, as Dobrev et al. [14, Section 7, arXiv] write, “a lone agent cannot cross any non-constant distance, as the irregular nature of the port labels would lead it astray, never to meet any other agent”. The tightness of our bound on the oriented grid combined with the perceived necessity of having moving agents accompanied by a partner seems to indicate that we cannot do better. However, there is no formal proof of any lower bound beyond the synchronous 3-agent and semi-synchronous 4-agent lower bounds [7, 15] that carry over from the case of the oriented grid. Admittedly, as such a formal lower bound might require us to find a “bad” input instance (i.e., a bad input edge labelings of the infinite n-dimensional grid) for every potential protocol with more than 3 synchronous, resp. 4 semi-synchronous, agents, it is not particularly surprising that we do not have better lower bounds – yet, making at least some progress would be desirable.

In this section, we will show that under a natural assumption the current lower bounds are actually optimal by providing tight upper bounds. Our assumption states that you cannot walk in a cycle if you always follow the same direction, or, more formally:

▶ **Assumption 4.** *Let $\ell \in \{1, \ldots, 2n\}$ be any port, z any positive integer, and $c_0, \ldots, c_z$ any sequence of cells such that, for each $0 \leq j \leq z-1$, we reach cell c_{j+1} by leaving cell c_j via port ℓ. Then $c_0 \neq c_z$.*

Surprisingly, this assumption does not contradict the intuition about agents traveling alone discussed above, yet it still allows us to prove upper bounds for unoriented grids matching the lower bounds obtained on *oriented* grids. While our upper bounds answer the question for the minimally required number of agents in a natural[3] setting very close to truly unoriented grids, we think that they also constitute a useful step on the way to a lower bound construction for the general unoriented setting (assuming that the current lower bounds are not optimal): any such construction necessarily has to contain cycles that violate Assumption 4.

Our Approach. The general idea behind our approach is to find a way to construct a stack also for unoriented grids. The natural idea of simply selecting one port ℓ and interpreting the sequence of cells obtained by successively leaving cells via port ℓ as the stack does not work: while it is easy for an agent to traverse the stack in the direction away from the base (it just has to leave each cell via port ℓ), traversing the stack in the opposite direction runs into the problem that there might be different neighboring cells from which the current cell can be reached via port ℓ and the traversing agent cannot know which is the one that belongs to the intended stack. Instead, we will build the desired virtual stack by constructing an auxiliary (infinite) directed labeled forest and then traversing (a part of) the forest from some starting cell in a DFS-like fashion, which will ensure that agents can traverse the stack in both directions. In particular, the same cell can occur in the stack several times; to distinguish the occurrences (and make it possible for an agent to traverse the virtual stack), our stack will formally consist of pairs (cell, integer), where the integers come from the set $\{1, \ldots, 2n\}$. For an illustration of the auxiliary graph and the virtual stack, we refer to Figures 1 and 2.

The Auxiliary Graph. We start by defining our auxiliary graph $G = (V, E)$. The vertices of G are the cells of our grid, and we have a directed edge (c, c') between two cells c, c' if c and c' are neighbors in the grid and cell c' is reached by leaving cell c via port 1.[4] In particular, this implies that each cell c has exactly one outgoing edge in G; we call the cell c' reached by traversing this edge the *parent of* c, and c a *child of* c'. Note that Assumption 4 ensures that G does not contain cycles, and hence, is an infinite forest. In particular, for any two neighboring cells c, c', at most one of the two possible edges (c, c') and (c', c) is present in E.

Let indegree(c) denote the *indegree* of a cell c, i.e., the number of edges from E incoming to c. We assign to each edge $e = (c, c')$ a level $L(e)$ as follows. For each cell c', we order the incoming edges (c, c') increasingly by the corresponding port of c', and then assign (distinct) *levels* from 1 to indegree(c') to the edges according to this order. For instance, if c' has two incoming edges (c, c') and (c'', c'), corresponding to ports 5 and 3 of c', respectively, then the order of the edges will be (c'', c'), (c, c'), and we will assign level 1 to (c'', c'), and level $2 = \text{indegree}(c')$ to (c, c').

---

[3] After all, it seems like a reasonable minimal requirement for a sense of direction that if you go north (or in any other direction) repeatedly, then you do not return to the starting point.

[4] The choice of port 1 here is arbitrary; choosing any other label from $\{1, \ldots, 2n\}$ works equally well.

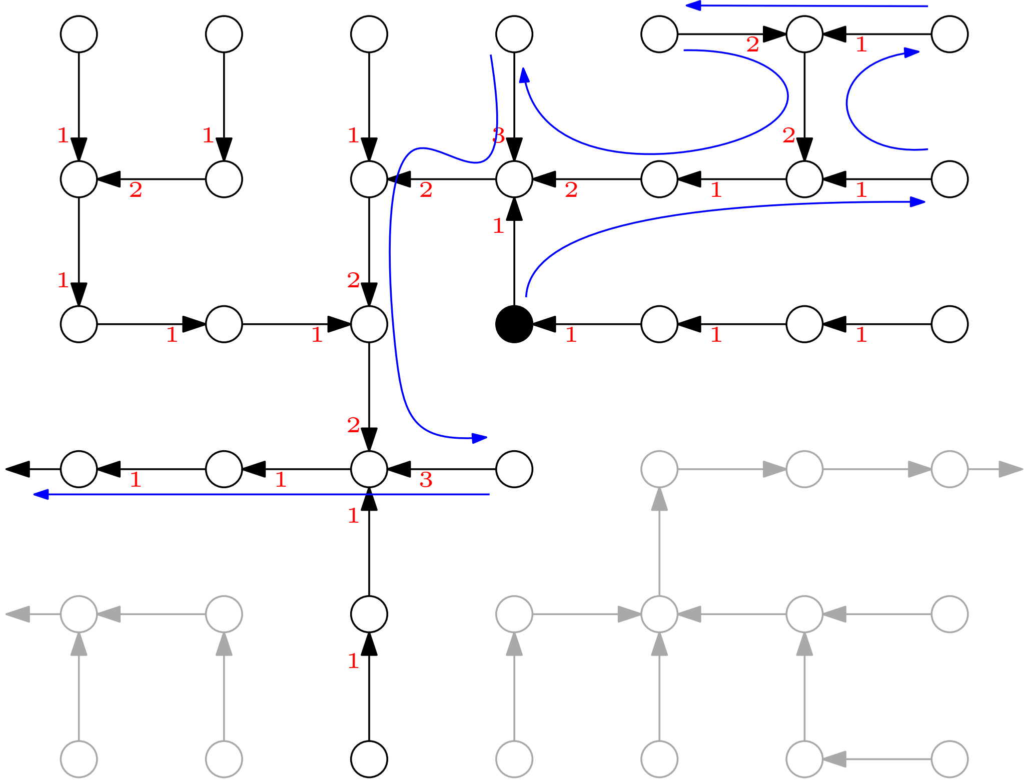


Figure 1 Figure 1 depicts a part of a possible auxiliary graph G for a 2-dimensional unoriented grid and the respective virtual stack. Vertices, i.e., cells, are represented by circles, and directed edges by arrows. The parts grayed out belong to different trees than the one containing the cell where a_1 is located (colored black). The edges are labeled with their respective levels. The physical cells of the virtual stack rooted in the black cell (i.e., the first component of the pairs the virtual stack consist of) are indicated by the route that starts in the black cell and follows the blue arrows. Each further step on this route leads to the physical cell corresponding to the next higher position in the stack, where the black cell indicates position 0. For each occurrence of a cell on this route, the corresponding level (i.e., the second component of the pairs the virtual stack consists of) is 1 if the current cell is a child of the previous cell, and equal to the level of the arrow traversed last plus 1 if the current cell is the parent of the previous cell. When the blue route goes from a cell to its parent, then the edge traversed next will have a level that is higher by 1 than the previously traversed edge; when the route goes from a cell to one of its children, then the edge traversed next has level 1. This leads to a virtual stack that corresponds to a part of a DFS exploration on the (infinite) tree containing the black cell, as can be seen in Figure 2, where the same route on the same auxiliary graph is depicted as a rooted tree.

The Virtual Stack. Using the auxiliary graph G, we now define, for each cell c, the *virtual stack $Virt_c$ rooted in c* as follows. Recall that Virt_c consists of pairs (cell, integer). We will use the functions $\mathrm{Cell}(\cdot)$ and $\mathrm{Level}(\cdot)$ to retrieve the first, resp. second, component of such a pair. The base of the stack is defined as $\mathrm{Virt}_c[0] := (c, \mathrm{indegree}(c) + 1)$. For each integer $j \geq 1$, we inductively define $\mathrm{Virt}_c[j]$ according to the following case distinction.

- If $\mathrm{Level}(\mathrm{Virt}_c[j-1]) = \mathrm{indegree}(\mathrm{Cell}(\mathrm{Virt}_c[j-1])) + 1$, then
  - $\mathrm{Cell}(\mathrm{Virt}_c[j])$ is defined as the parent of $\mathrm{Cell}(\mathrm{Virt}_c[j-1])$, and
  - $\mathrm{Level}(\mathrm{Virt}_c[j]) := L((\mathrm{Cell}(\mathrm{Virt}_c[j-1]), \mathrm{Cell}(\mathrm{Virt}_c[j]))) + 1$.

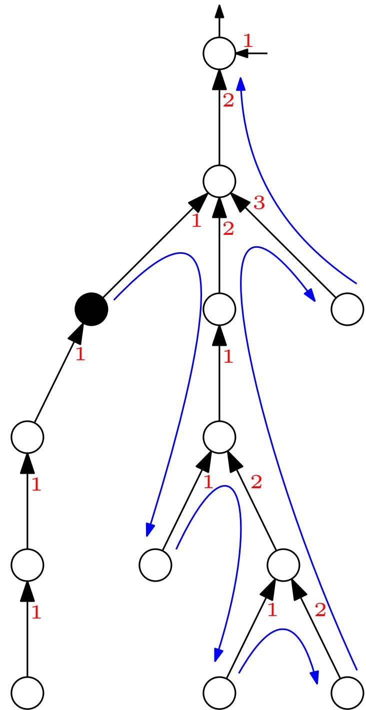


Figure 2 The same virtual stack as in Figure 1, depicted as a rooted tree.

- If $\text{Level}(\text{Virt}_c[j-1]) \leq \text{indegree}(\text{Cell}(\text{Virt}_c[j-1]))$, then
  - $\text{Cell}(\text{Virt}_c[j])$ is defined as the child of $\text{Cell}(\text{Virt}_c[j-1])$ connected to $\text{Cell}(\text{Virt}_c[j-1])$ via an (outgoing) edge of level $\text{Level}(\text{Virt}_c[j-1])$, and
  - $\text{Level}(\text{Virt}_c[j]) := 1$.

In other words, we inductively build the virtual stack rooted in c as follows. We start in c and leave c via the unique outgoing edge. Each time we enter a cell c' via an incoming edge, i.e., coming from a child c'', the next cell we visit is the next higher child of c', i.e., the child that is connected to c' via an edge of level $L(c'', c') + 1$. If no higher child remains, i.e., if (c'', c') has level $\text{indegree}(c')$, then the next cell we visit is the parent of c'. Each time we enter a cell c' from its parent, the next cell we visit is the first child of c', i.e., the child that is connected to c' via an edge of level 1. Hence, our stack corresponds to a DFS exploration of the tree in forest G containing c, where we assume that the part of the DFS that is executed before traversing the edge from c to its parent has already happened. As the tree is infinite, we may not reach every cell contained in the tree in finite time, but to use such a DFS exploration as a stack, this is not relevant. What is relevant, however, is that once we traverse an edge from a child to its parent, the DFS will never return to the child in finite time as Assumption 4 ensures that the parent chain starting from c (and therefore also any parent chain starting from any other cell visited by the partial DFS) is infinite. Combining this fact with the cyclic fashion in which each visited cell iterates through its children and parent to determine the neighbor visited next, we obtain the following observation.

▶ **Observation 5.** *Fix an arbitrary cell c. For any two non-negative integers $i \neq j$, we have $\mathit{Virt}_c[i] \neq \mathit{Virt}_c[j]$.*

In order to make use of the defined virtual stack, we need the agents to be able to represent their position in the stack in some way. However, given the specific design of the virtual stack, this is not difficult: each agent allocates a part of its state to keep track of the

level Level($\mathrm{Virt}_c[j]$) of the current position $\mathrm{Virt}_c[j]$ in the stack, while the first component Cell($\mathrm{Virt}_c[j]$) of the current position in the stack is simply represented by the cell the agent currently occupies. An advantage of this design is that each agent a_i can determine which other agents are in the same stack position as a_i, and which are not (despite possibly being in the same physical cell). In other words, each agent has all the necessary information to evaluate its transition function, even for moving on the virtual stack.

However, there is one piece still missing for using the virtual stack similar to a physical stack: we have to show that even a lone agent can traverse the virtual stack in either direction, i.e., that a finite automaton is sufficient to determine the physical cell that corresponds to the previous, resp. subsequent, position in the virtual stack, and similarly, to determine the level of that stack position. The following lemma takes care of this.

▶ **Lemma 6.** *There is a finite automaton that, when located in cell $Cell(Virt_c[j])$ in state $(i, Level(Virt_c[j]))$, where c is an arbitrary cell, $i \in \{-1, 1\}$, and $j \geq 1$ an arbitrary integer, moves to cell $Cell(Virt_c[j+i])$ and changes its state to $(i, Level(Virt_c[j+i]))$ in 2 time steps.*

Note that when applying Lemma 6, the finite automaton from Lemma 6 will only constitute a part of the finite automaton governing our agents in the final protocol for the unoriented case. Using Observation 5, Lemma 6, and the so-called handrail technique discussed in the full version, we are finally set to prove Theorem 2.

▶ **Theorem 2.** *Suppose that Assumption 4 holds. Then, for any positive integer n, 3 synchronous finite automata, resp. 4 semi-synchronous finite automata, suffice to explore any n-dimensional unoriented grid.*

6 Open Problems

While we provided tight bounds for a number of settings, still a number of open questions remain: Can we prove a higher lower bound on the number of agents required to explore an unoriented grid, than we can for oriented grids? Is there a protocol that achieves both an optimal number of agents and polynomial time exploration? For $n \geq 3$, can we improve the semi-synchronous protocol using randomness (the best known lower bound states that at least 3 agents are required)? How much can we reduce the computational power of the agents without compromising the optimal bounds? In our protocols, we can, e.g., replace agent a_1 with a movable marker, and in the semi-synchronous protocols, we can replace all agents except one with a movable marker; can we allow further/other restrictions?

References

1. Susanne Albers and Monika Henzinger. Exploring Unknown Environments. *SIAM Journal on Computing*, 29:1164–1188, 2000.
2. Romas Aleliunas, Richard M. Karp, Richard J. Lipton, Laszlo Lovasz, and Charles Rackoff. Random Walks, Universal Traversal Sequences, and the Complexity of Maze Problems. In *Foundations of Computer Science (FOCS)*, pages 218–223, 1979.
3. Ricardo A. Baeza-Yates, Joseph C. Culberson, and Gregory J. E. Rawlins. Searching in the Plane. *Information and Computation*, 106:234–252, 1993.
4. Anatole Beck. On the Linear Search Problem. *Israel Journal of Mathematics*, 1964.
5. M. Blum and W. J. Sakoda. On the Capability of Finite Automata in 2 and 3 Dimensional Space. In *Foundations of Computer Science (FOCS)*, pages 147–161, 1977.
6. Manuel Blum and Dexter Kozen. On the Power of the Compass (or, Why Mazes Are Easier to Search Than Graphs). In *Foundations of Computer Science (FOCS)*, pages 132–142, 1978.

7 Sebastian Brandt, Jara Uitto, and Roger Wattenhofer. A Tight Lower Bound for Semi-Synchronous Collaborative Grid Exploration. In *International Symposium on Distributed Computing (DISC)*, pages 13:1–13:17, 2018.

8 Lothar Budach. Automata and Labyrinths. *Mathematische Nachrichten*, 86(1):195–282, 1978.

9 Lihi Cohen, Yuval Emek, Oren Louidor, and Jara Uitto. Exploring an Infinite Space with Finite Memory Scouts. In *Symposium on Discrete Algorithms (SODA)*, pages 207–224, 2017.

10 Xiaotie Deng and Christos Papadimitriou. Exploring an Unknown Graph. *Journal of Graph Theory*, 32:265–297, 1999.

11 Krzysztof Diks, Pierre Fraigniaud, Evangelos Kranakis, and Andrzej Pelc. Tree Exploration with Little Memory. *Journal of Algorithms*, 51:38–63, 2004.

12 Yann Disser, Jan Hackfeld, and Max Klimm. Undirected Graph Exploration with $\Theta(\log\log n)$ Pebbles. In *Symposium on Discrete Algorithms (SODA)*, pages 25–39, 2016.

13 Stefan Dobrev, Lata Narayanan, Jaroslav Opatrny, and Denis Pankratov. Exploration of High-Dimensional Grids by Finite Automata. In *International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 139:1–139:16, 2019.

14 Stefan Dobrev, Lata Narayanan, Jaroslav Opatrny, and Denis Pankratov. Exploration of high-dimensional grids by finite state machines. *CoRR*, abs/1902.03693, 2019. `arXiv:1902.03693`.

15 Yuval Emek, Tobias Langner, David Stolz, Jara Uitto, and Roger Wattenhofer. How Many Ants Does it Take to Find the Food? *Theor. Comput. Sci.*, 608:255–267, 2015.

16 Yuval Emek, Tobias Langner, Jara Uitto, and Roger Wattenhofer. Solving the ANTS Problem with Asynchronous Finite State Machines. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 471–482, 2014.

17 Ofer Feinerman, Amos Korman, Zvi Lotker, and Jean-Sebastien Sereni. Collaborative Search on the Plane Without Communication. In *Principles of Distributed Computing (PODC)*, pages 77–86, 2012.

18 Pierre Fraigniaud, David Ilcinkas, Guy Peer, Andrzej Pelc, and David Peleg. Graph Exploration by a Finite Automaton. *Theoretical Computer Science*, 345(2-3):331–344, 2005.

19 Frank Hoffmann. One Pebble Does Not Suffice to Search Plane Labyrinths. In *FCT*, pages 433–444, 1981.

20 Alejandro López-Ortiz and Graeme Sweet. Parallel Searching on a Lattice. In *CCCG*, pages 125–128, 2001.

21 Petrişor Panaite and Andrzej Pelc. Exploring Unknown Undirected Graphs. In *Symposium on Discrete Algorithms (SODA)*, pages 316–322, 1998.

22 H. A. Rollik. *Automaten in Planaren Graphen*, pages 266–275. Springer Berlin Heidelberg, Berlin, Heidelberg, 1979.

23 Kazuo Sugihara and Ichiro Suzuki. Distributed Algorithms for Formation of Geometric Patterns with many Mobile Robots. *Journal of Robotic Systems*, 13(3):127–139, 1996.

24 Ichiro Suzuki and Masafumi Yamashita. Distributed Anonymous Mobile Robots: Formation of Geometric Patterns. *SIAM Journal on Computing*, 28(4):1347–1363, 1999.

25 Ichiro Suzuki and Masafurni Yamashita. Distributed Anonymous Mobile Robots - Formation and Agreement Problems. In *Structural Information and Communication Complexity (SIROCCO)*, pages 1347–1363, 1996.

Distributed Dispatching in the Parallel Server Model

Guy Goren
Technion – Israel Institute of Technology, Haifa, Israel
sgoren@campus.technion.ac.il

Shay Vargaftik
VMware Research
shayv@vmware.com

Yoram Moses
Technion – Israel Institute of Technology, Haifa, Israel
moses@ee.technion.ac.il

Abstract

With the rapid increase in the size and volume of cloud services and data centers, architectures with multiple job dispatchers are quickly becoming the norm. Load balancing is a key element of such systems. Nevertheless, current solutions to load balancing in such systems admit a paradoxical behavior in which more accurate information regarding server queue lengths degrades performance due to herding and detrimental incast effects. Indeed, both in theory and in practice, there is a common doubt regarding the value of information in the context of multi-dispatcher load balancing. As a result, both researchers and system designers resort to more straightforward solutions, such as the power-of-two-choices to avoid worst-case scenarios, potentially sacrificing overall resource utilization and system performance. A principal focus of our investigation concerns the value of information about queue lengths in the multi-dispatcher setting. We argue that, at its core, load balancing with multiple dispatchers is a distributed computing task. In that light, we propose a new job dispatching approach, called *Tidal Water Filling*, which addresses the distributed nature of the system. Specifically, by incorporating the existence of other dispatchers into the decision-making process, our protocols outperform previous solutions in many scenarios. In particular, when the dispatchers have complete and accurate information regarding the server queue lengths, our policies significantly outperform all existing solutions.

2012 ACM Subject Classification Computing methodologies → Distributed algorithms; Networks → Network algorithms; Theory of computation → Online algorithms

Keywords and phrases Distributed load balancing, Join the Shortest Queue, Tidal Water Filling, Parallel Server Model

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.14

Related Version An extended version of this paper appears in `https://arxiv.org/abs/2008.00793`.

Funding *Guy Goren*: Partly supported by a grant from the Technion Hiroshi Fujiwara cyber security research center and the Israel cyber bureau, as well as by a Jacobs fellowship.
Yoram Moses: Yoram Moses is the Israel Pollak academic chair at the Technion. His work was supported in part by the Israel Science Foundation under grant 2061/19, and by the U.S.-Israel Binational Science Foundation under grant 2015820.

1 Introduction

Large software systems that govern the operations of data centers and of cloud-based services are important components of modern-day computing infrastructure. Such systems process a large volume of jobs that often arrive at distinct locations and are serviced by a multitude

© Guy Goren, Shay Vargaftik, and Yoram Moses;
licensed under Creative Commons License CC-BY

of servers. How jobs are assigned to servers, and in particular load balancing the servers' job queues, plays a central role in determining the effectiveness of the system's overall performance.

The traditional approach to load balancing in this setting, known as the *supermarket model* [38, 39, 5], employs a single centralized dispatcher to which all requests are forwarded, and from which they are assigned to the servers. In the last decade, with the increasing size of cloud services and applications, using a single dispatcher has become a problematic bottleneck. System designers have consequently shifted to architectures that employ multiple dispatchers [6, 4, 26]. The load balancing problem for such multi-dispatcher systems is a very natural and urgent distributed systems problem. However, to the best of our knowledge, it has received only limited attention in the literature (see Section 1.1). The current paper considers the problem from a distributed computing perspective. We consider a setting with M dispatchers and N servers. Jobs arrive at the dispatchers according to a stochastic process, and the servers complete jobs at a stochastic rate. Each dispatcher observes the jobs that it receives, but not those received by the other dispatchers. Dispatchers interact with the servers when they send them jobs, and can also communicate with them in order to obtain information regarding the lengths of the servers' queues.

Load balancers are typically evaluated in terms of their behavior under heavy load, which occurs when the rate of job arrivals approaches the rate at which servers are able to process the jobs. A central factor in determining the quality of a load balancing protocol is the response time that it offers the clients. A natural measure is thus the expected response time (or *latency*) for jobs submitted to the system. In some cases, however, a client's task is broken up into multiple jobs, and the task is successfully served only when the last of these jobs is processed. For such instances, it is also important to meet a desired tail latency for the 95th, 99th, or even the 99.9th percentile of the distribution [3, 25]. Indeed, according to publications by Google and Amazon, even a small sub-second addition to response time in dynamic content websites has led to a persistent loss of users and revenue [16, 28].

A principal focus of our investigation concerns the value of information about queue lengths in the multi-dispatcher setting. As discussed below, several well-known load balancing policies behave poorly when the dispatchers' information is highly correlated. They suffer from so-called "herd behavior," in which, when dispatchers identify the same servers as having short queues, they all send their jobs to these servers. As a consequence, the servers become overloaded, resulting in poor performance. This is especially acute when dispatchers share considerable information about the queue lengths. Indeed, in describing their solutions, recent papers have made statements such as "*Inaccurate information can lead to better performance*" [35], or "*Inaccurate information can improve performance*" [40]. Interestingly, the value of accurate information is doubted not only by theoreticians. In fact, open source load balancer deployments such as HAProxy [33] and NGYNX [7] as well as cloud service companies such as Fastly [18] and Netflix [30] report making use of limited queue-size information in order to avoid detrimental herd behavior effects. This all appears to be rather perplexing and counter-intuitive. Is it indeed the case that there is such thing as too much information in the load-balancing context? Naïvely, at least, we would expect information to be valuable in a resource allocation context such as this.

In this paper, we consider why popular policies suffer from herd behavior and suggest new policies that avoid it. We argue that in policies that exhibit herd behavior, a dispatcher can be thought of as optimizing its behavior w.r.t. its information in a manner that is oblivious to the presence of other dispatchers. We suggest that this can be avoided by an analysis that explicitly accounts for the fact that other dispatchers are present. To focus on these issues, we

start by considering an idealized setting in which all dispatchers have complete information regarding the queue sizes. Each dispatcher has access to the job requests that it receives, but does not know how many jobs each of the other dispatchers receives. (The distributions of arrival rates at dispatchers are unknown but assumed to be the same, as are those of server processing rates.) In this setting, we propose a new load-balancing policy called *Tidal Water Filling* (TWF), and prove that it is in a precise sense optimal for the case of complete information. Indeed, we show that it improves on all known policies. Finally, we demonstrate that its performance improves as the amount of information available to the dispatchers increases. **These results establish that, when used appropriately, information can be gainfully used for load balancing in the multi-dispatcher setting**. These are the main contributions of the paper.

Since the performance of Tidal Water Filling improves with available information regarding queue sizes, we turn to the question of how limited communication can be exploited to boost this information, thereby further improving the performance of TWF. To this end, we design a variant of the TWF protocol in which both servers and dispatchers keep track of queue size information. Moreover, whenever a dispatcher interacts with a server, the two share the information they have. The information is consistently updated by using standard distributed systems techniques such as timestamping the size data. We show using simulations that this approach allows a significant increase in the amount of relevant data used by dispatchers even when communication is limited, and, in turn, markedly improves the load balancing performance.

1.1 Related Work

In the standard supermarket (i.e., parallel server) model, it is well known that if the (single) dispatcher has complete information regarding the server queues, the protocol that routes each job to the server with the shortest queue (called *Join the Shortest Queue* and denoted by JSQ) offers strong performance and strong theoretical guarantees [38, 39, 5]. JSQ has also motivated the design of reduced-state load balancing techniques for resource-constrained scenarios in which the dispatcher is exposed to only partial information about the server queue lengths. For example, in *Power-of-d-choices*, denoted by JSQ(d) [17, 36, 20], when a job arrives, a dispatcher randomly probes d servers and assigns the job to a server with the shortest queue among them. A related strategy is called *Power of memory*, denoted by JSQ(d, m) [29, 22]. In JSQ(d, m), the dispatcher samples the m shortest queues to whom it sent jobs in the latest round, in addition to $d \geq m \geq 1$ new randomly chosen servers. The job is then routed to the shortest among these $d + m$ queues.

In the last decade, with the increasing size of cloud services and applications, the need to scale horizontally drove system designers to introduce multiple dispatchers into their design as a single dispatcher could no longer utilize hundreds and thousands of servers [6, 4, 26]. In such multi-dispatcher systems, traditional solutions such as JSQ suffer from detrimental herd behaviour and therefore systems operators abandon the use of readily available information and turn to reduced-state approaches such as JSQ(d) potentially sacrificing overall system performance and reduced resource utilization in order to prevent worst-case scenarios [30, 33, 7, 18, 16, 28].

In the search for a better alternative for the multi-dispatcher load balancing scenario, Join-the-idle-queue (JIQ) [31, 16, 21, 32, 37] was recently proposed. In JIQ, dispatchers are notified only by idled servers. In turn, a dispatcher sends jobs to an idle server when it is aware of one, or to a randomly selected server otherwise. JIQ was shown to significantly

improve performance at low and moderate loads over JSQ(d) due to its immediate prevention of server starvation [16]. However, at higher loads, its performance resembles random routing due to the absence of idle servers and its performance deteriorates quickly [41].

The most recent advances on load balancing for multi-dispatcher systems appeared in [35, 40]. In the Local Shortest Queue policy (LSQ), proposed in [35], each dispatcher keeps a local state with possibly outdated queue size information, which is infrequently updated. A dispatcher then sends jobs to a shortest queue by its local estimation. This can be viewed as a generalization of similar ideas suggested for single-dispatcher systems [2, 34]. LSQ was followed by LED [40], which extended the theoretical performance guarantees to a wider family of *tilted* dispatching policies. These local-state-driven policies were shown to outperform previous policies considered for the multi-dispatcher model such as JSQ, JSQ(d) and JIQ. Intuitively, this is because they use considerably less information than JSQ, which reduces herding, and, on the other hand, maintain local states at the dispatchers allowing for a long term memory and preventing many events in which no single good server is discovered.

However, the aforementioned policies admit a paradoxical behavior in which accurate information, or even partial but correlated information among the dispatchers, degrades their performance. This is because, like their complete state-information JSQ counterpart, having shared information about good servers leads them to herd-behavior and detrimental incast effects that increase tail latency. That is, in the multiple-dispatchers case, when updated information is available to different dispatchers, they all send at once all incoming jobs to the servers with the currently-shorter queues, overwhelming them with the accumulated traffic. This phenomenon has already been pointed out in [19], which suggested the importance of using randomness to break the symmetry.

Another line of work concerning load balancing in distributed systems is based on the balls-into-bins model [1]. Recent approaches commonly apply regret minimization (e.g., [11]) and adaptive techniques (e.g., [15]), producing more precise theoretical bounds. However, their model assumptions are not aligned with our model (e.g., we consider stochastic arrivals at each dispatcher and stochastic departures at each server). Consequently, their analysis does not directly apply in our model and vice versa.

2 Model

We consider a system with a set $\mathcal{D}$ of M dispatchers and a set $\mathcal{S}$ of N servers. Dispatchers can communicate with servers over communication channels or via shared memory, but there is no direct communication among dispatchers. The network is the complete undirected bipartite graph with edge set $E = \mathcal{D} \times \mathcal{S}$, and both jobs and standard messages can be sent over the edges of E.[1] The system proceeds over discrete synchronous rounds. Time starts at time 0, and round $t+1$ occurs between time t and time $t+1$. Each round consists of four phases: First, every dispatcher receives an external input with a set of job requests. Second, every dispatcher sends each of the jobs it received to a server for processing, and every job received by a server is added to its job queue. Third, each of the servers completes processing a set of (zero or more) jobs from the head of its queue and reports the results to the appropriate clients. (This is where jobs depart from the system.) Finally, in a potential fourth phase of the round, dispatchers and servers may communicate information about the status of the server queues. More formally, the four phases are:

[1] The high rate of incoming jobs at the dispatchers makes interaction among them undesirable. As a result, the assumption that dispatchers do not interact is standard practice [19, 40, 35, 37, 32, 21, 31, 16, 33, 7].

1. **Arrivals:** Some number, $a^{(m)}(t)$, of exogenous jobs arrive at dispatcher m at the beginning of round t. (We denote $a(t) = \sum_{m \in \mathcal{D}} a^{(m)}(t)$.) Job arrivals are governed by stochastic processes. For simplicity, we assume that $a^{(m)}(t)$ are *i.i.d.* random variables governed by the same distribution which, according to standard practice, *is not assumed to be known.*
2. **Dispatching.** In every round, each dispatcher forwards the jobs it received to the servers for service. We consider two variants of dispatching that address two distinct affinity constraints. In one, termed *splittable dispatching*, the dispatcher assigns each of the jobs to a server of its choice. This handles a setting with no affinity constraints. That is, when the jobs arriving at a dispatcher m can be processed independently. We separately consider *unsplittable dispatching*, in which all jobs that arrive at m in a given round must be forwarded to the same server. This handles a setting with strict affinity constraints. Where, e.g., the jobs arriving at m in a given round share data or resources.
3. **Departures.** Each server maintains a FIFO queue that keeps track of its pending jobs. We denote by $Q_n(t)$ the length of server n's queue at the beginning of round t (before any job arrivals and departures) and denote $Q(t) \triangleq \langle Q_1(t), \ldots, Q_N(t)\rangle$. Moreover, we denote by $g_n^{(m)}(t)$ the number of jobs that server n receives from dispatcher m in round t. Moreover, $g_n(t) = \sum_{m \in \mathcal{D}} g_n^{(m)}(t)$ denotes the total number of jobs sent to server n in round t. We denote server n's job completion rate (i.e., the number of jobs that it is able to complete) in round t by $s_n(t)$. We thus have that $Q_n(t+1) = \max\{0, Q_n(t)+g_n(t)-s_n(t)\}$. I.e., server n completes $s_n(t)$ jobs in round t if it has that many jobs to process. Otherwise, it completes all $Q_n(t) + g_n(t) < s_n(t)$ jobs in its possession. As in the case of arrivals, $s_n(t)$ is assumed to be stochastically determined. Again, for ease of exposition their distribution is assumed to be the same for all servers.
4. **Communication.** In the fourth phase of a round, dispatchers obtain information about the queue sizes at the servers. We will consider two settings. In the *complete information* case, every dispatcher is informed of all queue sizes[2] and so, at the start of round t, it knows $Q(t)$. The *incomplete information* setting is one in which queue information is communicated over the channels of $E = \mathcal{D} \times \mathcal{S}$, without all servers communicating with all dispatchers in every round.

Admissibility. For a setting as above to be feasible, it must be the case that the servers' processing power is sufficient for handling the incoming job requests. Denoting $s(t) = \sum_{n \in \mathcal{S}} s_n(t)$, we define the *load* to be $\rho = \mathbb{E}[a(0)]/\mathbb{E}[s(0)]$. To be admissible, it must hold that $\rho < 1$.[3]

3 Stochastic Coordination

As a first step towards designing an effective multi-dispatcher policy, let us consider the problem in the simpler, single dispatcher, case. In round t, this dispatcher has access to the vector $Q(t)$ of queue sizes at the servers, and to the number $a(t)$ of jobs that have been submitted to the system. If the dispatcher is free to send different jobs to different servers then it will, intuitively, dispatch jobs one by one according to the JSQ principle. It will send the first job to a queue of shortest length, and then iterate through the jobs, each time sending the next job to a queue of smallest size given the jobs that it has already assigned.

[2] In practice, such information could be gathered in different ways. E.g., by a shared bulletin board or shared memory, as well as by having servers update a central process, who can forward a single message to with $Q(t)$ to each dispatcher.

[3] In [8], for the purpose of mathematical stability analysis, we also make the standard assumption that the arrival and departure processes admit a finite variance, i.e., $\mathrm{Var}(a(0)) < \infty$ and $\mathrm{Var}(s(0)) < \infty$.

Algorithm 1 Computing the water level.

```
1: function WaterLevel(Q, a)   ▷ Q is the multiset of queue lengths; a is the total number of arrivals;
2:                             ▷ Min returns the minimal value in a multiset.
3:   while a > 0 do
4:     MinSet ← {Q_n ∈ Q | Q_n = Min(Q)}          ▷ The set of all minimal queues.
5:     if |MinSet| = |Q| then                     ▷ |·| denotes the cardinality of a set.
6:       return Min(Q) + a/|Q|
7:     NextMin ← Min(Q\MinSet)
8:     δ ← NextMin − Min(Q)
9:     if δ · |MinSet| < a then
10:      a ← a − δ · |MinSet|
11:      for Q_n ∈ MinSet do
12:        Q_n ← Q_n + δ
13:    else
14:      return Min(Q) + a/|MinSet|
```

Consequently, all queues to which jobs are sent end up with the same sizes (give or take 1). We can view this as being analogous to a process of filling water into a container: Consider a water container with an uneven bottom at heights that correspond to the histogram (rearranged in sorted order) defined by $Q(t)$. If we should pour a volume of $a(t)$ units of water into the container, then the water level will coincide with the height of the queues to which jobs were dispatched, up to a rounding error due to the fact that water is continuous and jobs are discrete. The largest amount of water would be poured into the deepest column, just as the largest number of jobs would be sent to the shortest queue.

On a given input (Q, a), Algorithm 1 computes the water level $\mathrm{WL} = \textsc{WaterLevel}(Q, a)$ that results from pouring a units of water into a container with bottom shaped according to Q. The height of each column once the water is poured would be $Q^* \triangleq (Q_1^*, \ldots, Q_N^*)$ where $Q_n^* \triangleq \max\{Q_n, \mathrm{WL}\}$. We remark that Q_n^* is not always an integer but might also be a rational number. If queue lengths are maintained in sorted order, then WL is efficiently computable in $O(\min(N, a))$ time complexity.

In a multi-dispatcher system dispatchers make decisions independently of each other. It is thus only natural for them to independently optimize their load balancing decisions. Namely, to dispatch jobs in exactly the same manner as if they were alone in the system. This is indeed the case in current solutions. However, in a system where one is not alone, such oblivious behavior of disregarding the others may result in sub-optimal performance [7, 18, 35, 40, 30]. This occurs when the same server is identified as the best destination by different dispatchers. These dispatchers then simultaneously forward jobs to this server, causing its queue to grow rapidly, increasing delay times and sometimes even causing the server to drop jobs. However, the fact that dispatchers make decisions independently does not mean that their decision making protocols must be independently optimized.

In the multi-dispatcher context that we are considering, dispatchers cannot directly coordinate their actions in every given round, since they do not directly communicate with each other. Nevertheless, it is possible to design their protocols in such a way that their actions will be compatible with each other, and will not conflict. The key to doing so is employing *randomized protocols*, in which the dispatchers' moves are stochastic. Indeed, randomization has been a standard tool for symmetry breaking in distributed computing for over four decades [27, 14]. Our goal will be to design probabilistic load balancing protocols that will provide good performance by optimizing the cumulative behavior of the dispatchers. This will provide the dispatchers with a silent form of stochastic coordination.

4 Tidal Water Filling

Focusing on the complete information setting, we assume that each dispatcher m has access to the number $a^{(m)} = a^{(m)}(t)$ of jobs that it has received in the current round, and to the vector of server queue sizes $Q = Q(t)$ (we shall omit the round number t when it is clear from context). Based on Q and $a^{(m)}$, it needs to decide where to send each job. We seek a solution that will be feasible to compute and amenable to analysis. In particular, we seek a policy that (1) is uniform for all dispatchers; given the same Q and $a^{(m)} = a^{(m')}$, both m and m' should act in the same way, and in which (2) a dispatcher treats all jobs uniformly. Hence, the output of m's computation is a vector $\langle p_1, \ldots, p_N \rangle$, where p_n is the probability that any given job will be sent by m to server n, for every $n \in \mathcal{S}$.

We denote by $\bar{g}_n^{(m)}$ the random variable specifying the number of jobs sent to server n by dispatcher m. The total number of jobs received by n is $\bar{g}_n = \sum_{m \in \mathcal{D}} \bar{g}_n^{(m)}$, and its queue size once it receives them is the random variable $\bar{Q}_n \triangleq Q_n + \bar{g}_n$. Finally, we shall denote $\bar{Q} \triangleq \langle \bar{Q}_1, \ldots, \bar{Q}_N \rangle$.

Recall that Q and the total number of jobs $a = a(t)$ determine a water-filling solution $Q^* = Q^*(Q, a)$ as described in Section 3. Our goal will be to design a policy that computes the dispatching probabilities P in such a way that the resulting queue sizes $\bar{Q}$ approximate Q^* as well as possible. More formally, we wish to minimize the L_2 distance between Q^* and $\bar{Q}$. Intuitively, a large distance from the water level WL=WaterLevel(Q,a) induces a large delay in response times (for a positive difference) or server starvation and lesser resource utilization (negative difference). Since we seek to avoid long delay tails as well as unnecessary server idleness, we consider a large deviation from the WL to be worse than several small ones. This rules out linear or sub-linear distance measures such as the L_1 distance. On the other hand, giving too much weight to large deviations may miss opportunities to optimize the mean. For example, the L_∞ distance (i.e., min-max) addresses only the largest deviation. We therefore choose to use the L_2 distance, since it balances these two desires and is amenable to formal analysis.

We denote the vector of job arrivals at the dispatchers by $\vec{a} = \langle a^{(1)}, \ldots, a^{(M)} \rangle$. As an interim step, we derive a policy that computes the dispatching probabilities based on Q and the full vector $\vec{a}$ of jobs that arrive in the round, and not only the allocation $a^{(m)}$ of a single dispatcher m. We will later discuss how this analysis can be applied to an individual dispatcher's computation. Notice that Q and $\vec{a}$ uniquely determine a (fixed) vector Q^* resulting from water filling Q with $a = \sum_{m \in \mathcal{D}} a^{(m)}$ new jobs. Now, a policy $P(Q, \vec{a})$ gives rise to the random variable vector $\bar{Q}$ of queue sizes, as described above.

Recall that $\bar{g}_n = \bar{Q}_n - Q_n$. Similarly, we denote $g_n^* \triangleq Q_n^* - Q_n$. Our goal is to minimize

$$\begin{aligned} \mathbb{E}\|Q^* - \bar{Q}\|_2^2 \;&=\; \mathbb{E}\|(Q_1^* - \bar{Q}_1, \ldots, Q_N^* - \bar{Q}_N)^T\|_2^2 \;= \\ &\mathbb{E}\|(Q_1 + g_1^* - Q_1 - \bar{g}_1, \ldots, Q_N + g_N^* - Q_N - \bar{g}_N)^T\|_2^2 \;= \\ &\mathbb{E}\|(g_1^* - \bar{g}_1, \ldots, g_N^* - \bar{g}_N)^T\|_2^2 \;= \sum_{n \in \mathcal{S}} \mathbb{E}\left[(g_n^* - \bar{g}_n)^2\right] \;= \\ &\sum_{n \in \mathcal{S}} {g_n^*}^2 - 2 \sum_{n \in \mathcal{S}} \left(g_n^* \mathbb{E}\left[\bar{g}_n\right]\right) + \sum_{n \in \mathcal{S}} \mathbb{E}\left[\bar{g}_n^2\right] \end{aligned} \tag{1}$$

We perform separate analyses for the splittable and for the unsplittable cases.

4.1 The Splittable Case

In the splittable case, every job is sent to a server n with a probability of p_n. This implies, in particular, that the random variable $\bar{g}_n^{(m)}$ admits a binomial distribution, that is, $\bar{g}_n^{(m)} \sim \text{Bin}\left(a^{(m)}, p_n\right)$. Since each decision at each dispatcher is done independently, $\{\bar{g}_n^{(m)} \mid m \in \mathcal{D}\}$

are independent binomial variables with probability p_n. Thus, $\bar{g}_n = \sum_{m\in\mathcal{D}} \bar{g}_n^{(m)}$, where $\bar{g}_n \sim \text{Bin}\left(\sum_{m\in\mathcal{D}} a^{(m)}, p_n\right) \sim \text{Bin}\,(a, p_n)$. Hence,

$$\mathbb{E}[\bar{g}_n] = ap_n \quad \text{and} \quad \mathbb{E}[\bar{g}_n^2] = ap_n(1-p_n) + a^2 {p_n}^2. \tag{2}$$

Given Q and a we can rewrite (1) using (2) as a function of $P = \langle p_1, \ldots, p_N \rangle$:

$$\begin{aligned} f(P) &= \mathbb{E}\|Q^* - \bar{Q}\|_2^2 = \sum_{n\in\mathcal{S}} {g_n^*}^2 - 2a\sum_{n\in\mathcal{S}} g_n^* p_n + \sum_{n\in\mathcal{S}} \left(ap_n - a{p_n}^2 + a^2{p_n}^2\right) \\ &= \sum_{n\in\mathcal{S}} {g_n^*}^2 - 2a\sum_{n\in\mathcal{S}} g_n^* p_n + a - a\sum_{n\in\mathcal{S}} {p_n}^2 + a^2\sum_{n\in\mathcal{S}} {p_n}^2 \end{aligned} \tag{3}$$

Now, to simplify the analysis, we first make the observation that for any strictly positive number of arrivals to the system (i.e., $a > 0$),

$$\arg\min f(P) \;=\; \arg\min\,(a-1)\sum_{n\in\mathcal{S}} {p_n}^2 - 2\sum_{n\in\mathcal{S}} g_n^* p_n. \tag{4}$$

We thus turn to solve the expression on the right-hand side. As can be seen from (4), for a single arrival to the system (i.e., $a = 1$), the solution would be to divide the probabilities arbitrarily among all shortest queues. Thus, we next assume that $a > 1$. Recall that we aim to compute a probability assignment P that optimizes $\arg\min f(P)$. In particular, we have that $\sum_{n\in\mathcal{S}} p_n = 1$ and $p_n \geq 0\ \forall n \in \mathcal{S}$. The optimization problem to solve in standard form is,

$$\begin{aligned} \min_P \quad & \tilde{f}(P) = (a-1)\sum_{n\in\mathcal{S}} {p_n}^2 - 2\sum_{n\in\mathcal{S}} g_n^* p_n \\ \text{s.t.} \quad & \sum_{n\in\mathcal{S}} p_n - 1 = 0, \quad -p_n \leq 0\ \forall n \in \mathcal{S}. \end{aligned} \tag{5}$$

Notice that this is not a linear program, because the objective function is not linear. However, the problem is convex with affine constraints. To solve the problem, we employ the Karush-Kuhn-Tucker method (KKT) [10, 12]. The associated Lagrangian function is

$$L(P,\Lambda) = (a-1)\sum_{n\in\mathcal{S}} {p_n}^2 - 2\sum_{n\in\mathcal{S}} g_n^* p_n - \sum_{n\in\mathcal{S}} \Lambda_n p_n + \Lambda_0(\sum_{n\in\mathcal{S}} p_n - 1). \tag{6}$$

The respective KKT conditions are

$$\begin{aligned} &\frac{\partial L}{\partial p_n} = 2(a-1)p_n - 2g_n^* - \Lambda_n + \Lambda_0 = 0 \quad \forall\, n \in \mathcal{S} && \text{(Stationarity)} \\ &\sum_{n\in\mathcal{S}} p_n - 1 = 0 \text{ and } p_n \geq 0 \quad \forall n \in \mathcal{S} && \text{(Primal feasibility)} \\ &\Lambda_n \geq 0 \quad \forall n \in \mathcal{S} && \text{(Dual feasibility)} \\ &p_n \Lambda_n = 0 \quad \forall n \in \mathcal{S} && \text{(Complementary slackness)} \end{aligned} \tag{7}$$

By Stationarity in (7) we obtain that, for any p_n,

$$p_n = \frac{2g_n^* - \Lambda_0 + \Lambda_n}{2(a-1)}, \tag{8}$$

and adding Complementary slackness from (7) yields that for any $p_n > 0$ we have,

$$p_n = \frac{2g_n^* - \Lambda_0}{2(a-1)}. \tag{9}$$

We can now substitute for p_n according to (9) in our objective function (5) to obtain a function of a single variable Λ_0. This yields,

$$\tilde{f}(P(\Lambda_0)) = (a-1)\sum_{p_n>0}\left(\frac{2g_n^* - \Lambda_0}{2(a-1)}\right)^2 - 2\sum_{p_n>0} g_n^*\left(\frac{2g_n^* - \Lambda_0}{2(a-1)}\right) = \frac{\sum_{p_n>0}\left(\Lambda_0^2 - (g_n^*)^2\right)}{a-1}. \quad (10)$$

Clearly, to minimize the objective function, we seek the smallest Λ_0 that satisfies the KKT conditions given in (7). Observe that we can lower bound Λ_0 by combining (8) with the Primal feasibility in (7) to obtain:

$$1 \;=\; \sum_{n\in\mathcal{S}} p_n \;=\; \sum_{n\in\mathcal{S}} \frac{2g_n^* - \Lambda_0 + \Lambda_n}{2(a-1)} \;\geq\; \sum_{g_n^*>0} \frac{2g_n^* - \Lambda_0 + \Lambda_n}{2(a-1)}. \quad (11)$$

Using $\sum_{n\in\mathcal{S}} g_n^* = \sum_{g_n^*>0} g_n^* = a$, and rearranging (11) yields

$$2a - \Lambda_0 \sum_{g_n^*>0} 1 \;+\; \sum_{g_n^*>0} \Lambda_n \;\leq\; 2(a-1). \quad (12)$$

Thus, due to the Dual feasibility in (7), we obtain

$$\Lambda_0 \geq \frac{2 + \sum_{g_n^*>0}\Lambda_n}{g_+^*} \geq \frac{2}{g_+^*}, \quad \text{where} \quad g_+^* \;\triangleq\; \sum_{g_n^*>0} 1. \quad (13)$$

Setting $\Lambda_0 = \frac{2}{g_+^*}$ and $\Lambda_n = 0$ for all $g_n^* > 0$ respects the KKT conditions and minimizes the objective function with respect to Λ_0. Finally, substituting $\frac{2}{g_+^*}$ for Λ_0 in Equation (9), we obtain that the optimal solution for $a > 1$ in the splittable case is

$$p_n \;=\; \max\{0, \frac{g_n^* - 1/g_+^*}{a-1}\}\;. \quad (14)$$

▶ **Definition 1** (Splittable tidal water filling). *Given $Q = Q(t)$ and $a = a(t) > 1$, a stochastic dispatching policy $P(Q, a)$ that, in every round t sends each job to server $n \in \mathcal{S}$ with probability $p_n = \max\{0, \frac{g_n^*-1/g_+^*}{a-1}\}$, implements tidal water filling (sTWF) in the splittable setting.*

Notice that sTWF depends only on $a = a(t)$ and $Q = Q(t)$. It does not depend on the full detail of $\vec{a}$. In the context of complete information, Q is available to the dispatcher. However, a is not. In order to use sTWF an individual dispatcher m must replace a with some estimate. If $\mathbb{E}[a(0)]$, the expected value of a, is known, it can be used. Similarly, if $\mathbb{E}[s(0)]$, the total expected completion rate of the servers is known, it may also be used to replace a. Since, by assumption, dispatcher m has access to $a^{(m)}(t)$, it can use $Ma^{(m)}(t)$ for $a(t)$. This has the nice property that the average of what the dispatchers use equals exactly the total arrivals at that round. That is, $\frac{1}{M}\sum_{m\in\mathcal{D}} Ma^{(m)}(t) = a(t)$. We will hereafter assume that dispatcher m estimates $a(t)$ in this manner.[4] In Section 4.3 we shall discuss the properties and the intuitive interpretation of the probabilities used in sTWF.

---

[4] In Appendix C of [8], we show that the resulting protocol satisfies the desirable *strong stability property* for discrete-time queuing systems. We do the same for all the policies introduced in this paper.

4.2 The Unsplittable Case

In the unsplittable case, we again assume that every dispatcher m knows the vector $Q(t)$ of queue sizes (complete information). It differs from the splittable case only in that m must send all of the jobs that it receives in a given round to a single server. This affects the mathematics of the optimization problem. First of all, knowing the complete vector $\vec{a}$ of arrivals makes a significant difference in this case. Indeed, given Q and $\vec{a}$, computing an optimal job assignment to the servers essentially requires solving an instance of BIN-PACKING. In Appendix A of [8] we prove its NP-hardness by a reduction from the PARTITION problem [9]. More precisely, we show the following.

▶ **Theorem 2.** *Given Q and $\vec{a}$, it is NP-hard to decide if* $\min \mathbb{E}\|Q^* - \bar{Q}\|_2^2 = 0$ *for a system with two servers.*

In general, the values $a^{(1)}, a^{(2)}, \ldots, a^{(M)}$ may be different from each other. Theorem 2 implies that optimizing for general Q and $\vec{a}$ is intractable. Instead, we optimize the unsplittable problem for the case that $a^{(1)} = a^{(2)} = \cdots = a^{(M)}$, which is consistent with the assumption that $a = Ma^{(m)}$, made in the splittable setting. We consider this case as a heuristic means to derive the dispatching probabilities. Our experiments in Section 4.4 show that the resulting policy works well, even when arrivals are governed by *i.i.d.* Poisson distributions, under which the arrival values $a^{(m)}$ are rarely identical.

We now reformulate the optimization problem in the unsplittable setting for this heuristic case. The difference from the splittable setting arises following (1) since now $\bar{g}_n^{(m)}$ does not admit a Binomial distribution. Instead, we have

$$\bar{g}_n^{(m)} = \begin{cases} a^{(m)}, & \text{w.p. } p_n, \\ 0, & \text{otherwise.} \end{cases}$$

Namely, a dispatcher m send all $a^{(m)}$ of its jobs to server n with probability p_n. Therefore, the total number of jobs that server n receives is $\bar{g}_n = \sum_{m\in\mathcal{D}} \bar{g}_n^{(m)}$. Since $\{\bar{g}_n^{(m)} \mid m \in \mathcal{D}\}$ are *i.i.d.* random variables, $\bar{g}_n$ has the following first and second moments,

$$\begin{aligned} \mathbb{E}[\bar{g}_n] &= Ma^{(m)}p_n, \\ \mathbb{E}[\bar{g}_n^2] &= \sum_{m\in\mathcal{D}} (a^{(m)})^2 p_n + \sum_{\substack{m,m'\in\mathcal{D}, \\ m\neq m'}} a^{(m)} g^{(m')} {p_n}^2 = M(a^{(m)})^2 p_n + M(M-1)(a^{(m)}p_n)^2. \end{aligned} \tag{15}$$

Given Q and $a^{(m)}$ we rewrite (1) substituting the first and second moments according to (15). This yields

$$\begin{aligned} f(P) &= \sum_{n\in\mathcal{S}} {g_n^*}^2 - 2Ma^{(m)} \sum_{n\in\mathcal{S}} g_n^* p_n + \sum_{n\in\mathcal{S}} \Big(M(a^{(m)})^2 p_n + M(M-1)(a^{(m)}p_n)^2 \Big) \\ &= \sum_{n\in\mathcal{S}} {g_n^*}^2 - 2Ma^{(m)} \sum_{n\in\mathcal{S}} g_n^* p_n + M(a^{(m)})^2 \sum_{n\in\mathcal{S}} p_n + (M^2-M)(a^{(m)})^2 \sum_{n\in\mathcal{S}} {p_n}^2. \end{aligned} \tag{16}$$

Recall that we aim to minimize $f(P)$ under the constraint that P is a probability assignment. That is, $\sum_{n\in\mathcal{S}} p_n = 1$ and $p_n \geq 0 \ \forall n \in \mathcal{S}$. Observe that,

$$\arg\min f(P) \ = \ \arg\min\, (M-1)a^{(m)} \sum_{n\in\mathcal{S}} {p_n}^2 - 2 \sum_{n\in\mathcal{S}} g_n^* p_n. \tag{17}$$

Again, we proceed to solve for the right hand side. As can be seen from (17), for a single-dispatcher system (i.e., $M = 1$), the solution would be to arbitrarily divide the probabilities

among all shortest queues, i.e., JSQ. Thus, we next assume that $M > 1$. The optimization problem in standard form becomes

$$\begin{aligned} \min_P \quad & \tilde{f}(P) = (M-1)a^{(m)} \sum_{n\in\mathcal{S}} {p_n}^2 - 2\sum_{n\in\mathcal{S}} g_n^* p_n \\ \text{s.t.} \quad & \sum_{n\in\mathcal{S}} p_n - 1 = 0, \quad -p_n \leq 0 \;\forall n \in \mathcal{S}. \end{aligned} \tag{18}$$

Once more, we employ the KKT method. Since the solution mainly follows similar lines to those of the splittable case, the full derivation is deferred to Appendix B of [8]. For the unsplittable case, however, the solution is more involved. In particular, it involves identifying the subset of servers $\mathcal{U} \subseteq \mathcal{S}$ for which positive probabilities should be assigned. We then use a summation on $\mathcal{U}$ to derive a lower bound on Λ_0 as in (11). We find this set of servers to be $\mathcal{U} = \{n \mid Q_n < \textsc{WaterLevel}(Q, (M-1)a^{(m)})\}$.[5] Namely, $\mathcal{U}$ is the set of servers that would be strictly below the water level if the total arrivals were $(M-1)a^{(m)}$ instead of $Ma^{(m)}$.

The solution for the optimization problem in (18) yields the following notion of tidal water filling for the unsplittable case:

▶ **Definition 3** (Unsplittable tidal water filling). *Given $Q = Q(t)$ and $a^{(m)} = a^{(m)}(t)$, let $\mathcal{U} = \{n \mid Q_n < \textsc{WaterLevel}(Q, (M-1)a^{(m)})\}$. An individual stochastic dispatching policy $P(Q, a^{(m)})$ for dispatcher m that, in every round t sends all its jobs to server $n \in \mathcal{S}$ with probability*

$$p_n = \max\left\{0, \frac{g_n^* - (a^{(m)} - \sum_{n'\notin\mathcal{U}} g_{n'}^*)/|\mathcal{U}|}{(M-1)a^{(m)}}\right\} \tag{19}$$

implements tidal water filling (uTWF) in the unsplittable setting.

Observe that, when at most one job arrives at each dispatcher, i.e., $a^{(m)} \leq 1$ for all $m \in \mathcal{D}$, there is no difference between the splittable and unsplittable problems. Indeed, in this case Definition 3 and Definition 1 coincide.

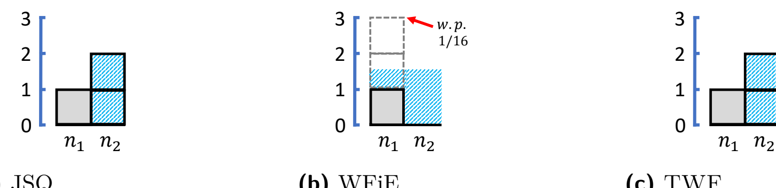
4.3 TWF vs Water Filling in Expectation

Recall that we aim to approximate water filling (i.e., Q^*). Consider the policy by which every dispatcher sends a job to each server n with a probability proportional to the amount of "water" it would receive in the pure water-filling solution. More formally, we define the Water Filling in Expectation policy (WFiE) to assign probabilities $P(Q, a) = \langle p_1, \ldots, p_N\rangle$, where for every n we have

$$p_n = \frac{g_n^*}{a}\ . \tag{20}$$

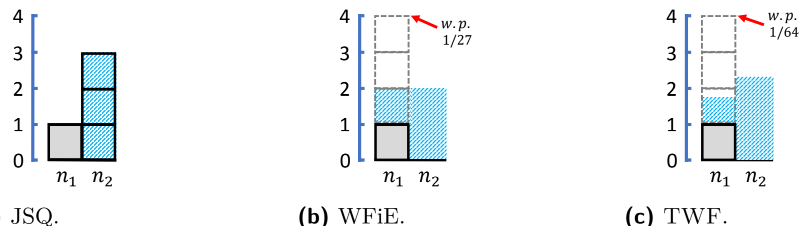
The expected length of each server n's queue under WFiE is precisely Q_n^*. Tidal water filling, however, does not produce the pure water-filling solution in expectation. Nevertheless, the math does not lie, and TWF improves on WFiE. We use the following two examples to demonstrate that WFiE is suboptimal, and to provide intuition for why TWF is better.

[5] Recall that $\textsc{WaterLevel}(\cdot,\cdot)$ is given by Algorithm 1.



(a) JSQ. **(b)** WFiE. **(c)** TWF.

Figure 1 A scenario with 2 dispatchers each of which receives a single job. Illustrating the expected arrivals at each queue for JSQ (Figure 1a), WFiE (Figure 1b) and TWF (Figure 1c).



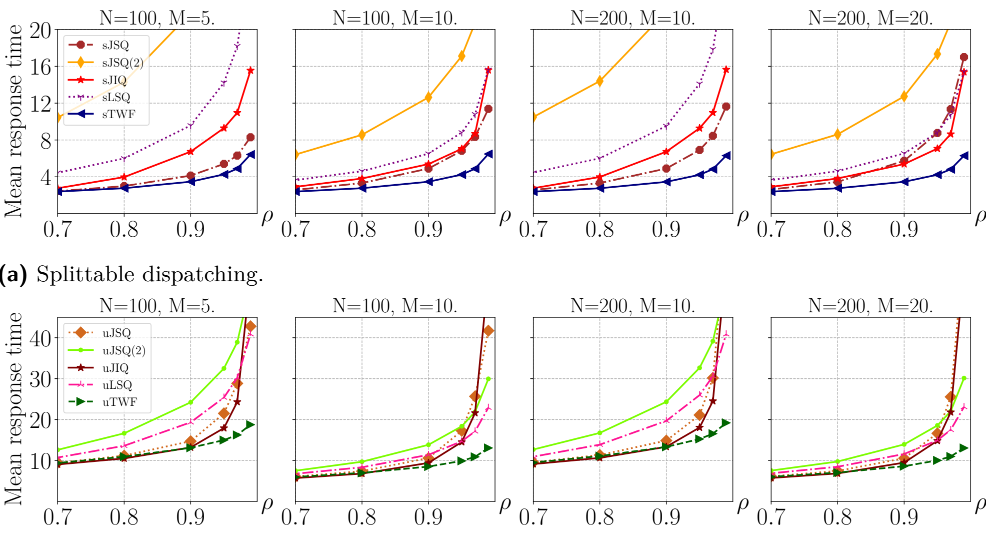
(a) JSQ. **(b)** WFiE. **(c)** TWF.

Figure 2 A scenario with 3 dispatchers each of which receives a single job. Illustrating the expected arrivals at each queue for JSQ (Figure 2a), WFiE (Figure 2b) and TWF (Figure 2c).

Example 1. Figure 1 depicts a system with $N = 2$ servers. At the beginning of the round, server n_1 has a single job in its queue, and server n_2 is idle (its queue is empty). There are $M = 2$ dispatchers, each of which receives a single job to dispatch. In this scenario, a dispatcher that uses JSQ will send its job to n_2; a dispatcher that uses WFiE will send its job to n_1 with probability $p_{n_1} = 1/4$ and to n_2 with probability $p_{n_2} = 3/4$; a dispatcher that uses TWF (since each dispatcher has a single job to dispatch in this scenario, sTWF and uTWF coincide) will send its job to n_2 with probability $p_{n_2} = 1$. The resulting *expected lengths* of the queues are depicted in Figure 1a for JSQ, in Figure 1b for WFiE, and in Figure 1a for TWF. Observe that both JSQ and TWF guarantee the favorable solution in which the longest queue has size 2, while in WFiE there is a non-negligible probability of 1/16 that both jobs will be forwarded to n_1, creating a queue of size 3.

Example 2. Figure 2 illustrates a system with the same two servers and the same initial state, but with $M = 3$ dispatchers. Each of the three dispatchers receives a single job to dispatch. In this scenario, a dispatcher that uses JSQ will again send its job to n_2; a dispatcher that uses WFiE will send its job to n_1 with probability $p_{n_1} = 1/3$ and to n_2 with probability $p_{n_2} = 2/3$; a dispatcher that uses TWF will send its job to n_1 with probability $p_{n_1} = 1/4$ and to n_2 with probability $p_{n_2} = 3/4$. The resulting *expected lengths* of the queues are depicted in Figure 1a for JSQ, in Figure 1b for WFiE, and in Figure 1a for TWF. In this case, JSQ results in herding towards n_2, creating a queue of size 3 there. WFiE results is a probability of 1/27 for ending with 4 jobs queuing at n_1, while n_2 remains idle. Observe that TWF reduces the probability of this worst-case allocation from 1/27 to 1/64. This is precisely the advantage that TWF provides over TFiE in general. We note that TWF also provides, with high probability, a favorable allocation in comparison to JSQ. In the second example, for instance, JSQ will produce a better outcome than TWF with probability $1.56\% = 1/64$, while TWF will be better than JSQ with probability $42.2\% = 27/64$.

Roughly speaking, our TWF policies have a greater bias towards short queues than WFiE does. As illustrated by the examples this, in turn, reduces the probability that queues will grow excessively long, and reduces the probability for short queues to become idle.



(a) Splittable dispatching.

(b) Unsplittable dispatching.

Figure 3 Average job response time as a function of the load over four different systems. The x-axis represents load ρ. The y-axis represents the average response time.

4.4 Evaluation

We conducted an empirical study of our TWF policies via simulations. In all of the simulations, at round t a dispatcher m has access only to $a^{(m)}(t)$. Recall that in both sTWF and uTWF dispatcher m uses $M \cdot a^{(m)}(t)$ as an estimate for $a(t)$.

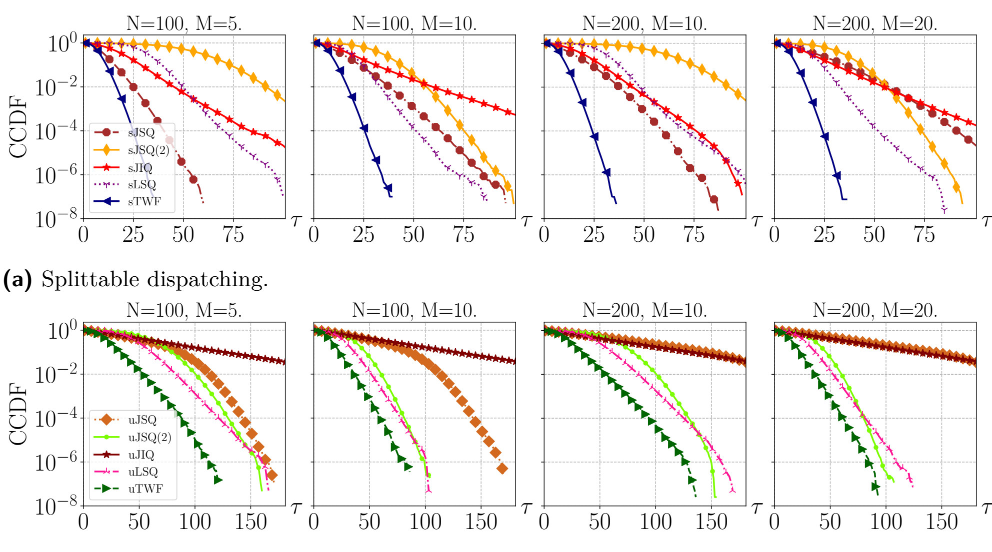
Arrivals and departures. Our modeling of the arrival and departure processes follows standard practice (e.g., [35, 16, 22, 34, 2]). In each round, we set $a^{(m)}(t) \sim Poisson(\lambda)$ for each dispatcher $m \in \mathcal{D}$, and $s^{(n)}(t) \sim Geometric(\mu)$ for each server $n \in \mathcal{S}$. Therefore, the load on the system is $\rho \triangleq M\lambda/(N\frac{\mu}{1-\mu})$.

Dispatching policies. We compare our policies to JSQ, the Power-of-two-choices denoted by JSQ(2), JIQ and the recently proposed LSQ.[6] We use a prefix of s and u to denote the splittable and the unsplittable case. For the splittable case, similarly to sTWF, other policies are also allowed to split jobs for parallel processing.

Setup. We consider systems with different numbers of servers N and dispatchers M. For each system, we run a set of simulations. Each simulation is configured with a different load and lasts for 10^5 time slots (rounds).

Results. Figure 3 shows the mean job response time (y-axis) across the different loads (x-axis) for different systems. It is notable that sTWF outperforms all other policies in the splittable case across all systems, and uTWF does the same in the unsplittable case. Moreover, as the load increases the gap between the TWF policies and the rest grows significantly. We remark that none of the other policies is in second place across the board.

[6] Specifically, we use LSQ-Sample(2). See [35] for details.



(a) Splittable dispatching.

(b) Unsplittable dispatching.

Figure 4 Response-time tail distribution over four different systems at high load ($\rho = 0.99$). The x-axis represents the response time (denoted by τ). The y-axis represents the CCDF.

For example, sJSQ is the second best at the smaller system ($N = 100, M = 5$) while sJIQ is the second best at the largest system ($N = 200, M = 20$).

As mentioned in Section 1 and in [3, 25, 16, 28], tail distributions play a crucial role in the parallel-server setting. We proceed to measure the tail distributions under various system parameters under the challenging scenario of a high load of ($\rho = 0.99$). This is depicted in Figure 4 using the complementary cumulative distribution function (CCDF), which shows for each response time (x-axis, denoted by τ), what is the fraction of jobs that surpass it (y-axis). E.g, for sLSQ in system $(200, 10)$ the response time of the 99th percentile (i.e., 10^{-2}) is ≈ 50 rounds. Again, our TWF policies outperform all other techniques in all systems. Notice, that while uJSQ(2) is competitive in comparison to the other techniques when considering delay-tail decay, it is less favorable compared to them in terms of the average response time (see, e.g., Figure 3b). Intuitively, this is because its increased randomness helps to reduce worst-case herding, but at the price of often not utilizing good available servers. In summary, in the complete information case, our simulations show that the TWF policies consistently outperform the previous approaches when the load is high.

5 Enhancing Performance for Partial Information

In this section we relax the requirement that dispatchers have complete and accurate information regarding Q. That is, we now consider situations in which a dispatcher does not know the exact state of all servers. In line with recent work [35, 40, 2, 34], we consider a system where each dispatcher keeps a local array representing the servers' queue lengths, which may contain inaccurate (e.g., outdated) information. Communication is used to update array entries in the following manner: At the end of each round (i.e., in the fourth phase), a dispatcher establishes communication links with a fraction $\eta \leq 1$ of the servers, which are chosen uniformly at random. The corresponding entries in the dispatcher's local state are then updated with these servers' queue lengths. Additionally, a dispatcher that establishes a communication link to send jobs to server n during round t learns $Q_n(t)$. Notice that $\eta = 1$

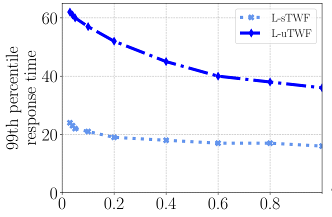


Figure 5 Response times as a function of η at high load ($\rho = 0.99$) for a system with $(N, M) = (100, 10)$.

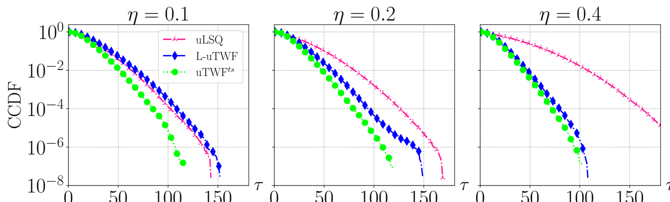


Figure 6 The effect of a distributed communication protocol on performance in a $(N, M) = (200, 20)$ system. Measuring the response time tail distribution (i.e., CCDF) at high load ($\rho = 0.99$).

corresponds to the complete information assumption; we use η as a parameter designating the extent of partial information available to the dispatchers. A dispatcher that uses uTWF based on its local array is said to implement *Local* uTWF (L-uTWF). *Local* sTWF is defined in the same manner. Figure 5 illustrates the simulations results. It shows that the response time improves monotonically as η increases.

This motivates us to increase the available information to the dispatchers. We attempt to do so without increasing the number of links that a dispatcher establishes. This is of interest since in many system the cost of communication lies mainly in the connection establishment rather than in the size of its content [24, 23]. To that end, we keep track of queue-size information at the servers, in a local array of size N. A server updates its local array based on its own queue length and information that it receives from dispatchers with which it has connections. Whenever a communication link between a dispatcher and a server is established, they merge their arrays. This is obtained by assigning time stamps to array entries, and maintaining the most recent information per entry upon the merge (cf. [13]).[7]

To test the effectiveness of the above scheme, we conduct an experiment comparing our protocols with the state-of-the-art LSQ-Sample of [35, 40] for different values of η. We denote by uTWF^{ts} the unsplittable policy from Definition 3 based on local arrays at both dispatchers and servers with time stamps. Figure 6 shows how the performance of uTWF^{ts} improves on that of L-uTWF for given values of η. The figure also illustrates that even the simpler L-uTWF policy is competitive with LSQ already at $\eta = 0.1$. As η grows (and the queue information improves), our protocols perform better, while LSQ's performance degrades.

6 Discussion

This work has demonstrated that, contrary to popular belief, queue-size information can be judiciously used to improve the quality of load balancing. In particular, we provided new policies that avoid herding and outperform all previous solutions for the case of complete information.

More generally, load balancing in the multi-dispatcher parallel server model is a natural question to explore using distributed systems tools and techniques. Our analysis in Section 5, for example, made use of time-stamping and flooding to improve the load balancing performance when information is partial. There are many additional aspects of the partial information setting that should be explored. For example, we note that when queue size information is sparse, the TWF's advantages do not come into play, and its performance

[7] To the best of our knowledge, this method was not previously employed in the parallel server model.

is not better than that of previous load-balancing policies. We state as an open problem how the advantages of previous approaches can be combined with those of TWF to obtain a policy that would make the best use of information across the whole spectrum of possibilities.

In another vein, it would be interesting to investigate how information about the distributions governing a multi-dispatcher systems can be obtained, and how they can be used to improve load-balancing performance. Can they provide good estimates for the TWF policies, and if so, how much benefit can they bring? Much is clearly left to be done.

References

1. Micah Adler, Soumen Chakrabarti, Michael Mitzenmacher, and Lars Rasmussen. Parallel randomized load balancing. *Random Structures & Algorithms*, 13(2):159–188, 1998.
2. Jonatha Anselmi and Francois Dufour. Power-of-d-choices with memory: Fluid limit and optimality. *Mathematics of Operations Research*, 2020.
3. Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.
4. Daniel E Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 523–535, 2016.
5. Atilla Eryilmaz and Rayadurgam Srikant. Asymptotically tight steady-state queue length bounds implied by drift conditions. *Queueing Systems*, 72(3-4):311–359, 2012.
6. Rohan Gandhi, Hongqiang Harry Liu, Y Charlie Hu, Guohan Lu, Jitendra Padhye, Lihua Yuan, and Ming Zhang. Duet: Cloud scale load balancing with hardware and software. *ACM SIGCOMM Computer Communication Review*, 44(4):27–38, 2014.
7. Owen Garrett. NGINX and the "Power of Two Choices" Load-Balancing Algorithm. , published on November 12, 2018. URL: `https://www.nginx.com/blog/nginx-power-of-two-choices-load-balancing-algorithm`.
8. Guy Goren, Shay Vargaftik, and Yoram Moses. Distributed dispatching in the parallel server model, 2020. `arXiv:2008.00793`.
9. Richard M Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.
10. William Karush. Minima of functions of several variables with inequalities as side conditions. *Master's Thesis, Department of Mathematics, University of Chicago*, 1939.
11. Robert Kleinberg, Georgios Piliouras, and Éva Tardos. Load balancing without regret in the bulletin board model. *Distributed Computing*, 24(1):21–29, 2011.
12. Harold W Kuhn and Albert W Tucker. Nonlinear programming. In *Proceedings of the Second Berkeley Symposium on Mathematical Statistics and Probability*, pages 481–492, Berkeley, Calif., 1951. University of California Press. URL: `https://projecteuclid.org/euclid.bsmsp/1200500249`.
13. Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978. `doi:10.1145/359545.359563`.
14. Daniel Lehmann and Michael O Rabin. On the advantages of free choice: A symmetric and fully distributed solution to the dining philosophers problem. In John White, Richard J Lipton, and Patricia C Goldberg, editors, *Conference Record of the 8th Annual ACM Symposium on Principles of Programming Languages*, pages 133–138. ACM Press, 1981. `doi:10.1145/567532.567547`.
15. Christoph Lenzen and Roger Wattenhofer. Tight bounds for parallel randomized load balancing. In *Proceedings of the 43rd annual ACM Symposium on Theory of Computing*, pages 11–20, 2011.

16 Yi Lu, Qiaomin Xie, Gabriel Kliot, Alan Geller, James R Larus, and Albert Greenberg. Join-idle-queue: A novel load balancing algorithm for dynamically scalable web services. *Performance Evaluation*, 68(11):1056–1071, 2011.

17 Malwina J Luczak, Colin McDiarmid, et al. On the maximum queue length in the supermarket model. *The Annals of Probability*, 34(2):493–527, 2006.

18 Tyler McMullen. Load Balancing is Impossible. Scaleconf 2016. URL: `https://www.youtube.com/watch?v=kpvbOzHUakA`.

19 Michael Mitzenmacher. How useful is old information? *IEEE Transactions on Parallel and Distributed Systems*, 11(1):6–20, 2000.

20 Michael Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.

21 Michael Mitzenmacher. Analyzing distributed join-idle-queue: A fluid limit approach. In *2016 54th Annual Allerton Conference on Communication, Control, and Computing*, pages 312–318. IEEE, 2016.

22 Michael Mitzenmacher, Balaji Prabhakar, and Devavrat Shah. Load balancing with memory. In *43rd Annual IEEE Symposium on Foundations of Computer Science.*, pages 799–808. IEEE, 2002.

23 YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. AccelTCP: Accelerating Network Applications with Stateful TCP Offloading. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 77–92, 2020.

24 David Murray, Terry Koziniec, Kevin Lee, and Michael Dixon. Large MTUs and internet performance. In *13th IEEE International Conference on High Performance Switching and Routing*, pages 82–87, 2012.

25 Rajiv Nishtala, Paul Carpenter, Vinicius Petrucci, and Xavier Martorell. Hipster: Hybrid task manager for latency-critical cloud workloads. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 409–420, 2017.

26 George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *26th Symposium on Operating Systems Principles (SOSP)*, pages 325–341, 2017.

27 Michael O Rabin. N-process synchronization by $4 \log_2 N$-valued shared variables. In *21st Annual Symposium on Foundations of Computer Science*, pages 407–410. IEEE Computer Society, 1980. `doi:10.1109/SFCS.1980.26`.

28 Eric Schurman and Jake Brutlag. The user and business impact of server delays, additional bytes, and http chunking in web search. In *Velocity Web Performance and Operations Conference*. O'Reilly, 2009.

29 Devavrat Shah and Balaji Prabhakar. The use of memory in randomized load balancing. In *Proceedings IEEE International Symposium on Information Theory*, page 125, 2002.

30 Mike Smith. Netflix Technology Blog. Rethinking Netflix's Edge Load Balancing. September 2018. URL: `https://netflixtechblog.com/netflix-edge-load-balancing-695308b5548c`.

31 Alexander L Stolyar. Pull-based load distribution in large-scale heterogeneous service systems. *Queueing Systems*, 80(4):341–361, 2015.

32 Alexander L Stolyar. Pull-based load distribution among heterogeneous parallel servers: the case of multiple routers. *Queueing Systems*, 85(1-2):31–65, 2017.

33 Willy Tarreau. HAProxy. Test Driving "Power of Two Random Choices" Load Balancing. , published on February 15, 2019. URL: `https://www.haproxy.com/blog/power-of-two-load-balancing/`.

34 Mark van der Boor, Sem Borst, and Johan van Leeuwaarden. Hyper-scalable jsq with sparse feedback. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 3(1):1–37, 2019.

35 Shay Vargaftik, Isaac Keslassy, and Ariel Orda. LSQ: Load Balancing in Large-Scale Heterogeneous Systems With Multiple Dispatchers. *IEEE/ACM Transactions on Networking*, 2020.

36 Nikita Dmitrievna Vvedenskaya, Roland L'vovich Dobrushin, and Fridrikh Izrailevich Karpelevich. Queueing system with selection of the shortest of two queues: An asymptotic approach. *Problemy Peredachi Informatsii*, 32(1):20–34, 1996.

37 Chunpu Wang, Chen Feng, and Julian Cheng. Distributed join-the-idle-queue for low latency cloud services. *IEEE/ACM Transactions on Networking*, 26(5):2309–2319, 2018.

38 Richard R Weber. On the optimal assignment of customers to parallel servers. *Journal of Applied Probability*, 15(2):406–413, 1978.

39 Wayne Winston. Optimality of the shortest line discipline. *Journal of Applied Probability*, 14(1):181–189, 1977.

40 Xingyu Zhou, Ness Shroff, and Adam Wierman. Asymptotically optimal load balancing in large-scale heterogeneous systems with multiple dispatchers. *arXiv preprint arXiv:2002.08908*, 2020.

41 Xingyu Zhou, Jian Tan, and Ness Shroff. Heavy-traffic delay optimality in pull-based load balancing systems: Necessary and sufficient conditions. *ACM SIGMETRICS Performance Evaluation Review*, 47(1):5–6, 2019.

Distributed Dense Subgraph Detection and Low Outdegree Orientation

Hsin-Hao Su
Boston College, Chestnut Hill, MA, USA
suhx@bc.edu

Hoa T. Vu
San Diego State University, CA, USA
hvu2@sdsu.edu

Abstract

The densest subgraph problem, introduced in the 80s by Picard and Queyranne [Networks 1982] as well as Goldberg [Tech. Report 1984], is a classic problem in combinatorial optimization with a wide range of applications. The lowest outdegree orientation problem is known to be its dual problem. We study both the problem of finding dense subgraphs and the problem of computing a low outdegree orientation in the distributed settings.

Suppose $G = (V, E)$ is the underlying network as well as the input graph. Let D denote the density of the maximum density subgraph of G. Our main results are as follows.

- Given a value $\tilde{D} \leq D$ and $0 < \epsilon < 1$, we show that a subgraph with density at least $(1-\epsilon)\tilde{D}$ can be identified deterministically in $O((\log n)/\epsilon)$ rounds in the LOCAL model. We also present a lower bound showing that our result for the LOCAL model is tight up to an $O(\log n)$ factor. In the CONGEST model, we show that such a subgraph can be identified in $O((\log^3 n)/\epsilon^3)$ rounds with high probability. Our techniques also lead to an $O(\text{diameter}(G) + (\log^4 n)/\epsilon^4)$-round algorithm that yields a $1-\epsilon$ approximation to the densest subgraph. This improves upon the previous $O(\text{diameter}(G)/\epsilon \cdot \log n)$-round algorithm by Das Sarma et al. [DISC 2012] that only yields a $1/2-\epsilon$ approximation.
- Given an integer $\tilde{D} \geq D$ and $\Omega(1/\tilde{D}) < \epsilon < 1/4$, we give a deterministic, $\tilde{O}((\log^2 n)/\epsilon^2)$-round [1] algorithm in the CONGEST model that computes an orientation where the outdegree of every vertex is upper bounded by $(1+\epsilon)\tilde{D}$. Previously, the best deterministic algorithm and randomized algorithm by Harris [FOCS 2019] run in $\tilde{O}((\log^6 n)/\epsilon^4)$ rounds and $\tilde{O}((\log^3 n)/\epsilon^3)$ rounds respectively and only work in the LOCAL model.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Networks → Network algorithms; Mathematics of computing → Graph algorithms

Keywords and phrases Distributed Algorithms, Network Algorithms

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.15

Related Version A full version of the paper is available at `https://arxiv.org/abs/1907.12443`.

Acknowledgements We thank David Harris for pointing out Lemma 12, which can be used to improve the running time of the dual rounding algorithm in our previous version.

1 Introduction

The Dense Subgraph Problem. Given a graph $G = (V, E)$, the *maximum density subgraph* problem (or the densest subgraph problem) is to find a subgraph H, where its density $d(H) = |E(H)|/|V(H)|$ is maximized over all subgraphs of G. We denote the density of the maximum density subgraph of G, $\max_{H \subseteq G} d(H)$, by D.

[1] $\tilde{O}$ hides $\log \log n$ factors.

© Hsin-Hao Su and Hoa T. Vu;
licensed under Creative Commons License CC-BY
34th International Symposium on Distributed Computing (DISC 2020).
Editor: Hagit Attiya; Article No. 15; pp. 15:1–15:18
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

First studied by Picard and Queyranne [37] as well as Goldberg [26], the maximum density subgraph problem has found numerous applications in community detection in social networks [13, 14], link spam identification [8, 25], and computational biology [18, 38]. Faster algorithms [12, 20, 30] have been developed for the problem and its variants since then. Moreover, the problem has been widely studied under different models of computation such as the streaming settings [9, 15, 34], the dynamic setting [9, 40], the massive parallel computation settings [4, 5, 23], and the distributed settings [39].

We study the problem of detecting dense subgraphs in the distributed settings, namely, in both the LOCAL and the CONGEST models. Let n and m be the number of vertices and the number of edges respectively. Furthermore, let Δ be the maximum degree. In such models, vertices are labeled with unique IDs and they operate in synchronized rounds. In each round, each vertex sends a message to each of its neighbors, receives messages from its neighbors, and performs local computations. The time complexity of an algorithm is defined to be the number of rounds used. In the LOCAL model, the message size can be arbitrary. In the CONGEST model, the message size is upper bounded by $O(\log n)$ bits. We consider the following parameterized version of the maximum density subgraph problem in the distributed settings, which may capture the computational nature of some applications such as how a dense community can be found by only communicating with the neighbors in social networks.

DenseSubgraph($\tilde{D}, \epsilon$) : Given a graph $G = (V, E)$, a parameter $\tilde{D} \geq 0$, and $0 < \epsilon < 1$, every vertex u outputs a value $h_u \in \{0, 1\}$ such that $d(H) \geq (1 - \epsilon)\tilde{D}$ where $H = \{u \mid h_u = 1\}$. If $\tilde{D} \leq D$ then H must be non-empty.

The first question we investigate is whether DenseSubgraph($\tilde{D}, \epsilon$) can be solved locally. Intuitively, most dense subgraphs have short diameters because they are well-connected. Thus, they can be detected locally. Our first result justifies this intuition.

▶ **Theorem 1.** *There exists a deterministic algorithm for DenseSubgraph($\tilde{D}, \epsilon$) that runs in $O((\log n)/\epsilon)$ rounds in the LOCAL model.*

We will also present a lower bound showing that the running time of the algorithm is tight up to an $O(\log n)$ factor. The algorithm for the LOCAL model uses large message size. This begs the question of whether the problem can be solved in the CONGEST model while remaining in the poly$(1/\epsilon, \log n)$-round regime. We show that this is indeed possible with randomization.

▶ **Theorem 2.** *There exists a randomized algorithm that solves DenseSubgraph($\tilde{D}, \epsilon$) w.h.p.[2], and runs in $O((\log^3 n)/\epsilon^3)$ rounds in the CONGEST model.*

Finding the densest subgraph in such distributed settings inevitably requires Ω(diameter(G)) rounds (e.g., consider two subgraphs of different densities connected by a path of length Ω(diameter(G))). Das Sarma et al. [39] gave an algorithm for finding a $(1/2 - \epsilon)$ approximation to the densest subgraph in $O(\text{diameter}(G)/\epsilon \cdot \log n)$ rounds in the CONGEST model. We show that the approximation factor can be improved and the dependency on diameter(G) can be made additive:

[2] W.h.p. denotes with high probability, which means with probability at least $1 - 1/n^c$ for an arbitrarily large constant c.

▶ **Corollary 3.** *There exists a randomized algorithm that finds a $(1-\epsilon)$-approximation to the maximum density subgraph w.h.p. and runs in $O(\text{diameter}(G) + (\log^4 n)/\epsilon^4)$ rounds in the* CONGEST *model.*

Inspired by web-graphs, Kannan and Vinay [29] defined the notion of density in directed graphs. Suppose that $G = (V, E)$ is a directed graph. The density of a pair of sets $S, T \subseteq V$ is defined as $d(S,T) = \frac{|E(S,T)|}{\sqrt{|V(S)||V(T)|}}$, where $E(S,T)$ denote the set of edges that go from a vertex in S to a vertex in T. Note that we are assuming messages can go in both directions of an edge. Our result in the LOCAL model can be generalized to the directed version of the problem.

The Low Outdegree Orientation Problem. Given an undirected graph $G = (V, E)$, an α-orientation is an orientation of the edges such that the outdegree of every vertex is upper bounded by α. Picard and Queyranne [37] observed that an α-orientation exists if and only if $\alpha \geq \lceil D \rceil$. Charikar [12] formulated the linear program (LP) for the densest subgraph problem, and its dual is the fractional version of the lowest outdegree orientation problem [4].

An α-orientation can be used to obtain a decomposition of the graph into α *pseudoforests* [37], where a pseudoforest is a collection of disjoint *pseudotrees* (a pseudotree is a connected graph containing at most one cycle). The relation between the pseudoforest decomposition and the maximum density is analogous to that of the forest decomposition and the arboricity shown by Nash-Williams [36].

We consider the low outdegree orientation problem in the distributed setting. The problem can be formally stated as follows.

> Given a graph $G = (V, E)$, an integer parameter $\tilde{D} \geq D$, and $0 < \epsilon < 1$, compute a $(1+\epsilon)\tilde{D}$-orientation. The orientation of an edge is decided by either of its endpoints.

Our contribution for this problem is a deterministic, $\tilde{O}((\log^2 n)/\epsilon^2)$-round algorithm in the CONGEST model. Previously, the best deterministic algorithm and randomized algorithm by Harris [28] run in $\tilde{O}((\log^6 n)/\epsilon^4)$ rounds and $\tilde{O}((\log^3 n)/\epsilon^3)$ rounds respectively and only work in the LOCAL model.

▶ **Theorem 4.** *Given an integer $\tilde{D} \geq D$, for any $32/\tilde{D} \leq \epsilon \leq 1/4$, there exists a deterministic algorithm in the* CONGEST *model that computes a $(1+\epsilon)\tilde{D}$-orientation and runs in*

$$O\left(\frac{\log n}{\epsilon^2} + (\min(\log\log n, \log \Delta) + \log(1/\epsilon))^{2.71} \cdot (1/\epsilon)^{1.71} \cdot \log^2 n\right) \leq \tilde{O}((\log^2 n)/\epsilon^2) \text{ rounds.}$$

Table 1 Previous results on the low outdegree orientation problem in the distributed setting.

Reference	Time	Model	Approx.	Rand. or Det.
Barenboim and Elkin [6]	$O((\log n)/\epsilon)$	CONGEST	$2+\epsilon$	Det.
Ghaffari and Su [24]	$O((\log^4 n)/\epsilon^3)$	LOCAL	$1+\epsilon$	Rand.
Fischer et al. [17]	$2^{O(\log^2(1/\epsilon \cdot \log n))}$	LOCAL	$1+\epsilon$	Det.
Ghaffari et al. [21]	$O((\log^{10} n \cdot \log^5 \Delta)/\epsilon^9)$	LOCAL	$1+\epsilon$	Det.
Harris [28]	$\tilde{O}((\log^6 n)/\epsilon^4)$	LOCAL	$1+\epsilon$	Det.
Harris [28]	$\tilde{O}((\log^3 n)/\epsilon^3)$	LOCAL	$1+\epsilon$	Rand.
new	$\tilde{O}((\log^2 n)/\epsilon^2)$	CONGEST	$1+\epsilon$	Det.

Table 1 summarizes previous algorithms for this problem in the LOCAL and CONGEST models. It is worthwhile to note that $\lceil D \rceil \leq a(G) \leq \lceil D \rceil + 1$ [37], where $a(G)$ is the arborcity of the graph, as several previous results were parameterized in terms of $a(G)$.

Barenboim and Elkin [6] introduced the H-partition algorithm that obtains a $(2+\epsilon)a(G)$-orientation as well as a $(2+\epsilon)a(G)$ forest decomposition. Ghaffari and Su [24] observed that the problem of computing a $(1+\epsilon)a(G)$-orientation reduces to computing of maximal independent sets (MIS) on the conflict graphs formed by augmenting paths. The MIS can be computed efficiently by simulating Luby's randomized MIS on the conflict graph. Fischer et al. [17] initiated the study of computing such MIS (i.e. the maximal matching in hypergraphs) deterministically. They gave a deterministic quasi-polynomial (in r) algorithm for computing the maximal matching in rank r hypergraphs, resulting in a $2^{O(\log^2(1/\epsilon \cdot \log n))}$ round algorithm for the orientation problem. Later, the dependency on r has been improved to polynomial by [21], which results in a $O((\log^{10} n \cdot \log^5 \Delta)/\epsilon^9)$ rounds algorithm for the orientation problem. Recently, the deterministic running time for the problem is further improved by Harris [28] to $\tilde{O}((\log^6 n)/\epsilon^4)$ as they developed a faster algorithm for the hypergraph maximal matching problem. It is unclear if the above approaches via maximal matching in hypergraphs can be implemented in the CONGEST model without significantly increase on the number of rounds. The low outdegree orientation problem has also been studied in the centralized context by [7, 10, 19, 27, 31, 32].

Our methods and contributions in a nutshell. Our first contribution is a simple yet powerful observation that dense subgraphs have low-diameter approximations. We first give a simple proof via low-diameter decomposition [33, 35] that there exists a subgraph with diameter $O((\log n)/\epsilon)$ that has density at least $(1-\epsilon)D$ (Lemma 5). Hence, if each vertex examines its local neighborhood up to a small radius, at least one vertex gets a good estimate of the densest subgraph. With appropriate bookkeeping, this leads to our *deterministic* $O((\log n)/\epsilon)$-round algorithm for detecting dense subgraphs in the LOCAL model. We complement this algorithm with a lower bound showing that $\Omega(1/\epsilon)$ rounds are necessary (Lemma 6).

In the CONGEST model, the starting point for both problems of detecting dense subgraphs and low outdegree orientation is the adaptation of the multiplicative weights update algorithm of Bahmani et al. [4]. Their algorithm solves the dual linear program for DenseSubgraph($\tilde{D}, \epsilon$) (which is the linear program for low outdegree orientation) using the multiplicative weights method. They also showed how to round the dual program's fractional solution to find a dense subgraph. While it appears that their algorithm can be implemented directly in the CONGEST model, there are a few issues that we need to resolve.

A naive implementation of the algorithm in [4] to solve the dual linear program uses $O(\text{diameter}(G))$ rounds per iteration. Furthermore, for the dense subgraph detection problem DenseSubgraph($\tilde{D}, \epsilon$), the rounding procedures in [4, 12] are inherently global. This is because they require checking whether certain subgraphs have high enough density. These subgraphs may have large diameters. Our contribution here is to remove the dependence on diameter(G) using ideas from Lemma 5 along with appropriate bookkeeping. This results in a randomized algorithm that runs in $O((\log^3 n)/\epsilon^3)$ rounds instead of $O(\text{diameter}(G) \cdot \text{poly}(\log n, 1/\epsilon))$ rounds. Our adaption also removes the explicit use of real-valued weights and only use integers, which can be transmitted easily in the CONGEST model.

For the low outdegree orientation problem, we still need to develop a procedure to round the dual fractional solution obtained from the multiplicative weights update method into an integral solution efficiently and deterministically in the CONGEST model. Our

contribution here is twofold. We use the idea of recent rounding-type algorithms for distributed matching [1,16], where we process the fractional solution bit-by-bit and round the solution at each bit scale. We show that each scale reduces to the directed splitting problem where there are known algorithms for the LOCAL model [22,24]. We will show how to modify these directed splitting algorithms to run in the CONGEST model. We provide an analysis showing that the total error incurred by multiple bit scales is small so that each outdegree is at most $(1+\epsilon)D$ at the end of the rounding procedure.

Organization. In Section 2, we present our deterministic algorithm for the dense subgraph problem in the LOCAL model. Section 3 presents a randomized algorithm for the dense subgraph problem in the CONGEST model. Section 4 exhibits a deterministic algorithm for the low outdegree orientation problem in the CONGEST model. Due to space constraint, some proofs are omitted; all missing proofs can be found in the full version [41].

2 Deterministic Dense Subgraph Detection in the LOCAL Model

In this section, we investigate the locality of the dense subgraph detection problem. In particular, we show how to solve DENSESUBGRAPH$(\tilde{D},\epsilon)$ *deterministically* in the LOCAL model. Combining with ideas in this section, we show that this problem can also be solved in the CONGEST model with randomization in the next section.

The locality of dense subgraphs. We present an algorithm to solve DENSESUBGRAPH$(\tilde{D},\epsilon)$ in $O((\log n)/\epsilon)$ rounds deterministically in the LOCAL model. We first give a structural lemma showing that for some sufficiently large constant K, there exists a subgraph with diameter at most $K/\epsilon \cdot \log n$ that has density at least $(1-\epsilon)D$.

▶ **Lemma 5** (Densest subgraph's locality). *For all simple graphs, there exists a subgraph with diameter at most $K/\epsilon \cdot \log n$ for some sufficiently large constant K that has density at least $(1-\epsilon)D$.*

Proof. It can be shown that for any simple graph G with n vertices and m edges, we can decompose G into disjoint components such that each component has diameter at most $K/\epsilon \cdot \log n$ for some sufficiently large constant K and furthermore the number of inter-component edges is at most ϵm [3,33,35] (see also Theorem 11). This is known as the *low-diameter decomposition*.

Consider the densest subgraph $H^* \subseteq G$ with n^* vertices and m^* edges. We apply the low-diameter decomposition to H^* and let the components be $H_1^*,\ldots,H_t^*$. Let n_i^* and m_i^* be the number of vertices and edges in H_i^* respectively. Suppose that $d(H_i^*) < (1-\epsilon)D$ for all i. Then, $m_i^*/n_i^* < (1-\epsilon)D$ which implies $m_i^* < (1-\epsilon)m^*n_i^*/n^*$. Thus,

$$\sum_{i=1}^{t} m_i^* < (1-\epsilon)\sum_{i=1}^{t}\frac{m^*n_i^*}{n^*} = (1-\epsilon)m^* \ .$$

This implies that the number of inter-component edges is more than ϵm^* which is a contradiction. Therefore, at least one component H_j^* must have density $d(H_j^*) \geq (1-\epsilon)D$. ◀

Using Lemma 5, we can design a LOCAL algorithm to solve DENSESUBGRAPH$(\tilde{D},\epsilon)$ in $O((\log n)/\epsilon)$ rounds. See Algorithm 1.

▶ **Theorem 1.** *There exists a deterministic algorithm for* DENSESUBGRAPH$(\tilde{D},\epsilon)$ *that runs in $O((\log n)/\epsilon)$ rounds in the* LOCAL *model.*

Algorithm 1 Distributed Algorithm for DenseSubgraph($\tilde{D}, \epsilon$) in the LOCAL model.

1: Initialize $h_v \leftarrow 0$ for all vertices v.
2: Let $r = K/\epsilon \cdot \log n$ for some sufficiently large constant K.
3: Each vertex v collects the subgraph induced by its r-neighborhood $N_r(v) = \{u : \text{dist}(v, u) \leq r\}$ and computes the densest subgraph $H(v)$ in $G[N_r(v)]$, i.e., the subgraph induced by $N_r(v)$.
4: **if** $d(H(v)) \geq (1-\epsilon)\tilde{D}$ **then**
5: $\quad$ v becomes an active vertex and collects its $2r$-neighborhood $N_{2r}(v)$.
6: **if** an active vertex v has the smallest ID among $N_{2r}(v)$ **then**
7: $\quad$ v becomes a black vertex.
8: Each black vertex v broadcasts $H(v)$ to $N_r(v)$ and set $h_u \leftarrow 1$ for all u in $V(H(v))$.

Proof. Consider Algorithm 1. The number of rounds is clearly $O(r) = O(1/\epsilon \cdot \log n)$, since it is dominated by the steps in Lines 3, 5, and 8 which is $O(r)$. Appealing to Lemma 5, if $\tilde{D} \leq D$, then there must be a subgraph C with diameter at most r whose density is at least $(1-\epsilon)D \geq (1-\epsilon)\tilde{D}$. Therefore, at least one vertex must be active. Among the active vertices, there must be a black vertex, i.e., the active vertex with the smallest ID. We therefore have a non-empty output.

The next observation is that if u and v are black vertices, then $H(v)$ and $H(u)$ are disjoint. Otherwise, there is a path of length at most $2r$ from u to v which leads to a contradiction. This is because u and v cannot both be black vertices if $\text{dist}(u, v) \leq 2r$. Furthermore, it is easy to see that if two subgraphs $G[A]$ and $G[B]$, where $A, B \subset V$ are disjoint, have density at least $(1-\epsilon)\tilde{D}$, then $d(G[A \cup B]) \geq (1-\epsilon)\tilde{D}$. To see this,

$$d(G[A \cup B]) \geq \frac{|E(A)| + |E(B)|}{|A| + |B|} \geq \frac{(1-\epsilon)\tilde{D}(|A| + |B|)}{|A| + |B|} = (1-\epsilon)\tilde{D} .$$

Let the set of black vertices be B. Since we argued that $H(v)$'s are disjoint for $v \in B$ and the output subgraph is $H = \cup_{v \in B} H(v)$, we deduce that $d(H) \geq (1-\epsilon)\tilde{D}$.

Finally, we need to argue that the algorithm is correct for when $\tilde{D} > D$. Note that if the output subgraph is non-empty, its density is at least $(1-\epsilon)\tilde{D}$. Hence, if $\tilde{D} > D$, the algorithm may output an empty subgraph or a subgraph with density $(1-\epsilon)\tilde{D}$ which are both acceptable. ◀

We show that Theorem 1 is tight in terms of ϵ up to a constant.

▶ **Lemma 6.** *Let $0 < \epsilon < 0.1$. Any algorithm that solves DenseSubgraph($\tilde{D}, \epsilon$) correctly with probability at least 0.51 requires more than $1/(10\epsilon)$ rounds.*

Proof. Consider deterministic algorithms for DenseSubgraph($\tilde{D}, \epsilon$) where $\tilde{D} = 1 - \epsilon$. Without loss of generality, assume $1/(10\epsilon)$ is an integer and let $\ell = 4/(10\epsilon) + 1$. Consider ℓ vertices $1, \ldots, \ell$ where $(i, i+1) \in E$ for $1 \leq i \leq \ell - 1$. We consider two cases. Case 1: $(\ell, 1) \in E$ in which the graph (called G_1) is a cycle. Case 2: $(\ell, 1) \notin E$ in which the graph (called G_2) is a chain of ℓ vertices. Let $v = \lfloor \ell/2 \rfloor + 1$. In both cases, the $(1/(10\epsilon))$-neighborhood of v is the same and therefore h_v must be the same regardless of whether the network is G_1 or G_2. We observe that a chain of t vertices has density $1 - 1/t$.

If $h_v = 0$, then let the underlying graph be G_1. Then $D = 1 > 1 - \epsilon = \tilde{D}$, and therefore the output must be non-empty. The only correct output is when every vertex outputs 1 since otherwise the output subgraph's density is at most $1 - 1/(4/(10\epsilon) + 1) < (1-\epsilon)^2 = (1-\epsilon)\tilde{D}$ for $\epsilon < 0.1$. Hence, the algorithm fails.

If $h_v = 1$, then let the underlying graph be G_2. Then $D = 1 - 1/(4/(10\epsilon) + 1) < (1-\epsilon)\tilde{D}$. Then, the only correct output is that every vertex outputs 0. Therefore, the algorithm fails.

For a randomized lower bound, we choose the above two inputs with probability 0.5 each and therefore the probability that a deterministic algorithm is correct is at most 0.5. By Yao's minimax principle, no randomized algorithm outputs correctly with probability more than 0.5. ◀

We remark that an interesting *open question* is whether the $\log n$ factor is necessary for either the lower bound or the upper bound. In our full version [41], we show that similar locality result also holds for directed densest subgraph.

3 Dense Subgraph Detection in the CONGEST Model

Relating the densest subgraph problem and the lowest outdegree orientation problem. We show how to find a) a dense subgraph and b) a low outdegree orientation (in Section 4) by first solving the same feasibility LP. Let us first consider the LP formulations in Figure 1.

(PRIMAL) Maximum Density Subgraph

$$\begin{aligned} \text{maximize} \quad & \sum_{e \in E} x_e \\ \text{subject to} \quad & \\ & \sum_{v \in V} y_v = 1 \\ \forall e = uv \in E, \quad & x_e \leq y_u \text{ and } x_e \leq y_v \\ \forall e \in E, u \in V, \quad & y_e, x_u \geq 0 \ . \end{aligned}$$

(DUAL) Lowest Outdegree Orientation

$$\begin{aligned} \text{minimize} \quad & z \\ \text{subject to} \quad & \\ \forall e = uv \in E, \quad & \alpha_{eu} + \alpha_{ev} \geq 1 \\ \forall e \in E, u \in V, \quad & \sum_{e \ni u} \alpha_{eu} \leq z \\ \forall e = uv \in E, \quad & \alpha_{eu}, \alpha_{ev} \geq 0 \ . \end{aligned}$$

Figure 1 Linear programs for densest subgraph and fractional lowest outdegree orientation.

Given a subgraph $H \subseteq G$ of size k, in the primal, we can set $y_v = 1/k$ for all $v \in V(H)$ and $x_e = 1/k$ for all edges $e \in E(H)$ while setting other variables to 0. Then, $\sum_{e \in E} x_e = |E(H)|/k = d(H)$. In fact, the optimal value of the LP is exactly the maximum subgraph density D. Charikar gave a rounding algorithm that recovers the densest subgraph [12].

We observe that the dual models the lowest outdegree orientation problem. In particular, if an edge $e = uv$ is oriented from u to v then we set $\alpha_{eu} = 1$ and $\alpha_{ev} = 0$. By duality, the dual is fractionally feasible if and only if $z \geq D$.

Now we consider the feasibility versions of the programs. We say a primal solution is a solution satisfying PRIMAL(z) if it is a feasible solution whose objective function is at least z. Similarly, we say a dual solution is a solution satisfying DUAL(z) if it is a feasible dual solution whose objective function is at most z.

Adapting the algorithm of Bahmani et al. [4]. Algorithm 2 and Algorithm 3 are adapted from the multiplicative weights update approach of [4] for solving DUAL and PRIMAL. We modify them in a way so that we only have to operate with integers instead of real-valued weights. This is more suitable for the CONGEST model because it may take $\omega(1)$ rounds to transfer a real-valued weight over an edge. Similar to the tree packing method for minimum cuts [42], this is another example of a combinatorial algorithm derived from multiplicative weights update method, where the weights are only used in the analysis but not in the algorithms. However, as mentioned in the first section, this adaptation has a dependence on diameter(G). We will subsequently show how to remove this dependence when solving DENSESUBGRAPH($\tilde{D}, \epsilon$).

Algorithm 2 and Algorithm 3 have the same structure except for the outputs. Both algorithms consist of $T = O((\log n)/\epsilon^2)$ iterations. Every edge maintains a load $\ell(e)$ throughout the algorithms. In each iteration t, each vertex u has z units budget. Each vertex u distributes the z units of budget to the $\alpha_{eu}^{(t)}$ variables of the incident edges with the lowest $\lceil z/2 \rceil$ load. It allocates 2 units of budget to the incident edges with the lowest $\lceil z/2 \rceil - 1$ load and the remaining budget to the other edge. Then the load of an edge is updated by adding the allocation from both endpoints. In Algorithm 2, we output the average of the allocations, $\sum_t \alpha_{eu}^{(t)}/T$, multiplied by $(1+2\epsilon)$. In Algorithm 3, in each iteration, the loads of the edges will be used as a guide to find a dense subgraph.

We can implement the algorithms by only sending integers across each link. This is because the value of each α_{eu} and $\ell(e)$ is a summation of an integer and an integer multiple of $z - 2(\lceil z/2 \rceil - 1)$. A pair of integers will be enough to express each value involved in the algorithms.

Algorithm 2 FRACTIONAL_DUAL(z, ϵ).

1: Let $T = K \cdot (1/\epsilon^2) \cdot \log n$ for some sufficiently large constant K.
2: Initialize the load $\ell(e) \leftarrow 0$ for each edge e.
3: **for** $t = 1, 2, \ldots, T$ **do**
4:     **for** each vertex u **do**
5:         Let $e_1, e_2, \ldots, e_{\deg(u)}$ be the edges adjacent to u where $\ell(e_1) \le \ell(e_2) \ldots \le \ell(e_{\deg(u)})$.
6:         Set $\alpha_{e_i u}^{(t)} \leftarrow 2$ for $i = 1, \ldots, \min(\lceil z/2 \rceil - 1, \deg(u))$.
7:         Set $\alpha_{e_{\lceil z/2 \rceil} u}^{(t)} \leftarrow z - 2 \cdot (\lceil z/2 \rceil - 1)$ if $\deg(u) \ge \lceil z/2 \rceil$.
8:     **for** each edge $e = uv$ **do**
9:         Set $\ell(e) \leftarrow \ell(e) + \alpha_{eu}^{(t)} + \alpha_{ev}^{(t)}$.
10: Return $\alpha_{eu} = \dfrac{\sum_{t=1}^{T} \alpha_{eu}^{(t)}}{T} \cdot (1 + 2\epsilon)$.

Algorithm 3 INTEGRAL_PRIMAL(z, ϵ).

1: Let $T = K \cdot (1/\epsilon^2) \cdot \log n$ for some sufficiently large constant K.
2: Initialize the load $\ell(e) \leftarrow 0$ for all e.
3: **for** $t = 1, 2, \ldots, T$ **do**
4:     **for** each vertex u **do**
5:         Let $e_1, e_2, \ldots, e_{\deg(u)}$ be the edges adjacent to u where $\ell(e_1) \le \ell(e_2) \ldots \le \ell(e_{\deg(u)})$.
6:         Set $\alpha_{e_i u}^{(t)} \leftarrow 2$ for $i = 1, \ldots, \min(\lceil z/2 \rceil - 1, \deg(u))$.
7:         Set $\alpha_{e_{\lceil z/2 \rceil} u}^{(t)} \leftarrow z - 2 \cdot (\lceil z/2 \rceil - 1)$ if $\deg(u) \ge \lceil z/2 \rceil$.
8:     Let $\ell_{\min} = \min_{e \in E} \lfloor \ell(e) \rfloor$ and $\ell_{\max} = \ell_{\min} + \lceil \frac{1}{\epsilon} \log \frac{2m}{\epsilon} \rceil$.
9:     **for** each integer $\ell \in [\ell_{\min}, \ell_{\max}]$ **do**
10:         Let V'_ℓ be the set of vertices with at least $\lceil z/2 \rceil$ incident edges e with $\lceil \ell(e) \rceil \le \ell$.
11:         Test if $G[V'_\ell]$ is a graph of density at least $(1 - 3\epsilon)z$.
12:         If yes, output $G[V'_\ell]$ and terminate.
13:     **for** each edge $e = uv$ **do**
14:         Set $\ell(e) \leftarrow \ell(e) + \alpha_{eu}^{(t)} + \alpha_{ev}^{(t)}$.

We will show that if we run the algorithms on the same input with the same parameters z and ϵ, then at least one of the following must be true:

- Algorithm 2 returns a solution satisfying DUAL($(1 + 2\epsilon)z$).
- Algorithm 3 outputs a subgraph of density at least $(1 - 3\epsilon)z$, i.e., an integral solution for PRIMAL($(1 - 3\epsilon)z$).

Define $w_e = (1-\epsilon)^{\ell(e)}$ to be the weight associated with the edge e. We will use $\ell^{(t)}(e)$ and $w_e^{(t)}$ if we are referring to the load or the weight of an edge e at the beginning of iteration t. We often use u and v to denote the endpoints of an edge e when the context is clear.

If it is the case that $\sum_{e\in E} w_e^{(t)}(\alpha_{eu}^{(t)} + \alpha_{ev}^{(t)}) \geq \sum_{e\in E} w_e^{(t)}$ for every iteration t, then we show that Algorithm 2 returns a feasible solution for $\text{DUAL}((1+2\epsilon)z)$. The following lemma is a standard derivation of the multiplicative weights update method [2, 43].

▶ **Lemma 7.** *Let $0 < \epsilon < 1/4$. Suppose that $\sum_{e\in E} w_e^{(t)}(\alpha_{eu}^{(t)} + \alpha_{ev}^{(t)}) \geq \sum_{e\in E} w_e^{(t)}$ for every iteration t in Algorithm 2. Then the dual variables $\{\alpha_{eu}\}$ returned by Algorithm 2 must be a feasible solution satisfying $\text{DUAL}((1+2\epsilon)z)$.*

Note that if we provide the algorithms with $z \geq D$, the condition $\sum_{e\in E} w_e^{(t)}(\alpha_{eu}^{(t)} + \alpha_{ev}^{(t)}) \geq \sum_{e\in E} w_e^{(t)}$ holds for every iteration t. This is because if $z \geq D$ then we know there exists a feasible solution such that $(\alpha_{eu} + \alpha_{ev}) \geq 1$ for every edge $e = uv$. This implies $\sum_{e\in E} w_e^{(t)}(\alpha_{eu} + \alpha_{ev}) \geq \sum_{e\in E} w_e^{(t)}$. Note that

$$\sum_{e\in E} w_e^{(t)}(\alpha_{eu}^{(t)} + \alpha_{ev}^{(t)}) = \sum_{u\in V}\sum_{e\ni u} \alpha_{eu}^{(t)} w_e^{(t)} = \sum_{u\in V}\sum_{e\ni u} \alpha_{eu}^{(t)} (1-\epsilon)^{\ell^{(t)}(e)} .$$

Hence, the way we assign $\{\alpha_{eu}^{(t)}\}$ maximizes $\sum_{e\in E} w_e^{(t)}(\alpha_{eu}^{(t)} + \alpha_{ev}^{(t)})$ over the set of feasible solutions $S(z)$, where

$$S(z) = \{\alpha : \sum_{e\ni u} \alpha_{eu} \leq z \text{ for all } u \text{ and } 0 \leq \alpha_{eu} \leq 2 \text{ for all } \alpha_{eu}\} .$$

Therefore, $\sum_{e\in E} w_e^{(t)}(\alpha_{eu}^{(t)} + \alpha_{ev}^{(t)}) \geq \sum_{e\in E} w_e^{(t)}(\alpha_{eu} + \alpha_{ev}) \geq \sum_{e\in E} w_e^{(t)}$. From the above lemma, this implies Algorithm 2 will output a fractional solution satisfying $\text{DUAL}(z)$.

Next, we need to show that if the opposite holds, i.e., $\sum_{e\in E} w_e^{(t)}(\alpha_{eu}^{(t)} + \alpha_{ev}^{(t)}) < \sum_{e\in E} w_e^{(t)}$ for some iteration t, then Algorithm 3 outputs a subgraph of density at least $(1-3\epsilon)z$. First, we rely on the following lemma which is a paraphrase of [4].

▶ **Lemma 8** (Paraphrase of [4])**.** *Let $1/2 < F < 1$. Suppose there exists a set of (non-negative) weights $\{w'_e\}$ such that $\sum_{e\in E} w'_e > F \cdot \max_{\alpha\in S(z)} \left(\sum_{e\in E} w'_e(\alpha_{eu} + \alpha_{ev})\right)$. Let V''_λ denote the set of vertices that are incident to at least $\lceil z/2 \rceil$ edges e with $w'_e \geq \lambda$. Then, there exists λ such that $d(G[V''_\lambda]) > F \cdot z$.*

Hence, if there is an iteration t where $\sum_{e\in E} w_e^{(t)}(\alpha_{eu}^{(t)} + \alpha_{ev}^{(t)}) < \sum_{e\in E} w_e^{(t)}$, we can apply the above lemma with $F = 1$ and $w'_e = w_e^{(t)}$ to find the desired dense subgraph. However, there may be $\Omega(m)$ potential values for λ to check. We circumvent this by discretizing the weights. Since we are only testing the densities of $G[V'_\ell]$ for integer loads ℓ, we are effectively discretizing the weights. We will show that this discretization only introduces a small error on the inequality so that we can apply Lemma 8 with $F = 1 - O(\epsilon)$. This, in turn, gives us the following.

▶ **Lemma 9.** *Suppose that there exists an iteration t in Algorithm 3 where $\sum_{e\in E} w_e^{(t)}(\alpha_{eu}^{(t)} + \alpha_{ev}^{(t)}) < \sum_{e\in E} w_e^{(t)}$. Then for $0 < \epsilon < 1/4$, Algorithm 3 will output a subgraph of density at least $(1-3\epsilon)z$ in that iteration.*

▶ **Lemma 10.** *Suppose $0 < \epsilon < 1/4$ and $\tilde{D} \leq D$. Then, there exists a deterministic algorithm that solves $\text{DENSESUBGRAPH}(\tilde{D}, \epsilon)$ in $O(\text{diameter}(G) + 1/\epsilon^3 \cdot \log^2 n)$ rounds in the* CONGEST *model.*

Proof. Assume that $\tilde{D} \leq D$. Let $z = (1 - \epsilon/2)\tilde{D}$ and $\epsilon' = \epsilon/8$. If we were to run Algorithm 2 with parameters z and ϵ', then it cannot possibly output a feasible solution for $\textsc{Dual}(z(1 + 2\epsilon'))$, since $\textsc{Dual}(z(1 + 2\epsilon'))$ is infeasible. This is because $z(1 + 2\epsilon') = (1 - \epsilon/2)(1 + \epsilon/4)\tilde{D} < \tilde{D} \leq D$. This implies that Algorithm 3 has to output a subgraph with density of at least $(1 - 3\epsilon')z \geq (1 - \epsilon/2)(1 - \epsilon/2)\tilde{D} \geq (1 - \epsilon)\tilde{D}$.

We now argue that Algorithm 3 can be implemented in $O(\text{diameter}(G) + 1/\epsilon^3 \cdot \log^2 n)$ rounds in the CONGEST model. Each step in the main loop of Algorithm 3 uses $O(1)$ rounds except for Line 8 and the inner loop (Line 9 – Line 12). Line 8 can be done in $O(\text{diameter}(G))$ rounds. The inner loop tests the density of $O((1/\epsilon)\log n)$ subgraphs. In total, there are $O((1/\epsilon)\log n) \cdot O((1/\epsilon^2)\log n)$ subgraphs to be tested. Each subgraph and its density can be computed in $O(\text{diameter}(G))$ rounds. These steps can be pipelined to run in $O(\text{diameter}(G) + (\log^2 n)/\epsilon^3)$ rounds. Finally, we argue that each $\alpha_{eu}^{(t)}$ can be represented using $O(\log\log n + \log(1/\epsilon))$ bits (Lemma 12) and therefore in each iteration, Line 14 in Algorithm 3 (and Line 9 in Algorithm 2) can be done in one CONGEST round. ◀

Using the above lemma, we now present the following results:

- There exists an algorithm that solves $\textsc{DenseSubgraph}(\tilde{D}, \epsilon)$ in $O(\text{poly}(\log n, 1/\epsilon))$ rounds w.h.p. In particular, we are able to avoid diameter(G) rounds in Lemma 10.
- There exists an algorithm that finds a $1 - \epsilon$ approximation of the densest subgraph (instead of solving the parameterized version) in $O(\text{diameter}(G) + \text{poly}(\log n, 1/\epsilon))$ rounds w.h.p.

Solving DenseSubgraph$(\tilde{D}, \epsilon)$. Our first goal is to avoid $O(\text{diameter}(G))$ rounds to solve the parameterized version of the densest subgraph problem. The main idea is to apply the low diameter decomposition and solve the problem in each component. First, we recall that the low-diameter decomposition can be implemented efficiently in the CONGEST model. See also [11].

▶ **Lemma 11** ([35]). *There exists an algorithm that decomposes the graph into disjoint components such that: 1) Each edge is inter-component with probability at most ϵ, 2) Each component has diameter $O(1/\epsilon \cdot \log n)$ w.h.p. and 3) Runs in $O(1/\epsilon \cdot \log n)$ rounds in the* CONGEST *model w.h.p.*

In the decomposition given by Miller et al. [35], the rough idea is that each vertex v draws δ_v from the exponential distribution $Exp(\epsilon)$. Let $\delta = K/\epsilon \cdot \log n$ for some sufficiently large constant K. At time step $\lfloor \delta - \delta_v \rfloor$, v wakes up and starts a breadth first search (BFS) if it has not been covered by another vertex's BFS. At the end of the decomposition, $v \in \text{cluster}(u)$ if $u = \min_{y \in V}(\text{dist}(y, v) + \lfloor \delta - \delta_y \rfloor)$. This algorithm can be simulated in the CONGEST model in $O(1/\epsilon \cdot \log n)$ rounds. Now, we are ready to prove our next main result.

▶ **Theorem 2.** *There exists a randomized algorithm that solves $\textsc{DenseSubgraph}(\tilde{D}, \epsilon)$ w.h.p. and runs in $O((\log^3 n)/\epsilon^3)$ rounds in the* CONGEST *model.*

Proof. We apply the decomposition above with parameter $\epsilon/2$ to the graph to obtain low-diameter components $C_1, \ldots, C_t$. Let H^* be the densest subgraph with n^* vertices and m^* edges. We condition on the event ξ that at most $\epsilon|E(H^*)|$ edges in $E(H^*)$ are inter-component. This happens with probability at least 1/2 according to Markov's inequality.

First, consider the case $\tilde{D} \leq D$. Using a similar argument as in the proof of Lemma 5, we can show that the densest subgraph of at least one component has density at least $(1 - \epsilon)D \geq (1 - \epsilon)\tilde{D}$. Specifically, let $H_i^* = H^* \cap C_i$ and let $C_i^* \subseteq C_i$ be the densest subgraph in C_i. Furthermore, let $|V(H_i^*)| = n_i^*$ and $|E(H_i^*)| = m_i^*$.

If $d(C_i^*) < (1-\epsilon)D$ for all $1 \le i \le t$. Then, $d(H_i^*) \le d(C_i^*) < (1-\epsilon)D$. This implies $m_i^* < (1-\epsilon)m^* n_i^*/n^*$. Therefore, $\sum_{i=1}^t m_i^* < (1-\epsilon)m^*$ which means that more than ϵm^* edges in $E(H^*)$ are inter-component. This is a contradiction since we condition on ξ.

For each low-diameter component C_i, using the algorithm in Lemma 10, we can solve DENSESUBGRAPH$(\tilde{D}, \epsilon)$ in C_i. As argued above, since $\tilde{D} \le D$, we must have a non-empty output in some component i. Observe that since the diameter of each C_i is $O(1/\epsilon \cdot \log n)$, this takes $O(1/\epsilon^3 \cdot \log^2 n)$ rounds.

We ensure ξ happen w.h.p. by repeating $O(\log n)$ trials. The total number of rounds becomes $O(1/\epsilon^3 \cdot \log^3 n)$. However, there is a catch regarding the consistency of the output. Recall that we want the subgraph induced by the marked vertices $\{v : h_v = 1\}$ has density at least $(1-\epsilon)\tilde{D}$. It is possible that for different trials j and j', v might be marked and unmarked respectively. We need to address the issue of how to decide the final output.

This can be done with proper bookkeeping. Originally, all vertices are unmarked. In each trial, compute a low-diameter decomposition $C_1, C_2, \ldots, C_t$. For each low-diameter component C_i, if it does not contain a marked vertex, check if there exists a subgraph $H_i \subseteq C_i$ with density at least $(1-\epsilon)\tilde{D}$ using the algorithm in Lemma 10. If there exists such subgraph, mark every vertex in H_i. In the end, return the subgraph induced by the marked vertices. The output must be non-empty w.h.p. by considering the first time the event ξ occurs. Since all the output subgraphs among the trials are disjoint and they have density at least $(1-\epsilon)\tilde{D}$, their union must have density at least $(1-\epsilon)\tilde{D}$ as argued in the proof of Theorem 1.

In the case $\tilde{D} > D$, the algorithm either returns a subgraph with density at least $(1-\epsilon)\tilde{D}$ or an empty subgraph which are both acceptable. ◀

Approximating the densest subgraph. We can also apply the ideas above to find a $1-\epsilon$ approximation of the densest subgraph. This is done by simply running the algorithm in Theorem 2 on different guesses for the maximum subgraph density D.

▶ **Corollary 12.** *There exists a randomized algorithm that finds a $(1-\epsilon)$-approximation to the maximum density subgraph w.h.p. and runs in $O(\text{diameter}(G) + (\log^4 n/\epsilon^4))$ in the* CONGEST *model.*

Proof. For each $\tilde{D} = 1, (1+\epsilon), (1+\epsilon)^2, \ldots, (1+\epsilon)^{\lceil \log_{1+\epsilon} n \rceil}$, we run the algorithm in Theorem 2. This requires $O(1/\epsilon^4 \cdot \log^4 n)$ rounds. We then identify the largest $\tilde{D}$ in which we have a non-empty output. In particular, we refer to when $\tilde{D} = (1+\epsilon)^i$ as phase i. Each vertex v sets $\psi(v) = i$ where i is the largest phase in which it is marked. In the end, we can broadcast $j = \max_{v \in V} \psi(v)$ to all vertices v in $O(\text{diameter}(G))$ rounds. Then, for every vertex v, if v is marked in phase j, set $h_v = 1$. ◀

We include the following lemma, which will be useful in bounding the running time for rounding the fractional dual solution into an integer dual solution in the next section.

▶ **Lemma 12.** *Suppose that we run Algorithm 2 with parameters z and ϵ, where z is an integer and ϵ is a (negative) power of 2. Moreover, assume, $T = \Theta((\log n)/\epsilon^2)$, in the main loop of Algorithm 2, is a power of 2. Each α_{eu} in the solution returned by Algorithm 2 contains at most $O(\log\log n + \log(1/\epsilon))$ bits.*

4 Deterministic Algorithms for Low Outdegree Orientation

Recall that in the low outdegree orientation problem, we are given an integer parameter $\tilde{D} \ge D$. The goal is to find an orientation of the edges such that for every vertex, the number of outgoing edges is upper bounded by $(1+\epsilon)\tilde{D}$.

In Section 3, we showed that Algorithm 2 can be used to obtain a solution for the fractional version of the low outdegree orientation problem. In this section, we will show how to round a fractional solution to an integral solution deterministically in the CONGEST model. We first present the framework and then describe the subroutine for the directed splitting procedure adapted from [22] in Section 4.1.

Let $\epsilon_1, \epsilon_2 \in \Theta(\epsilon)$ be error control parameters which will be determined later. First, we will run Algorithm 2 with parameters $\tilde{D}$ and $\epsilon_1/2$ to obtain a fractional solution for $\text{Dual}((1+\epsilon_1)\tilde{D})$. Let $\{\alpha'_{eu}\}_{e\in E, u\in V}$ be the output fractional solution. They satisfy the conditions that $\alpha'_{eu} + \alpha'_{ev} \geq 1$ for every edge $e = uv$ and $\sum_{e\ni u} \alpha'_{eu} \leq (1+\epsilon_1)\tilde{D}$ for every vertex u. We show how to round the α'-values to $\{0,1\}$ bit-by-bit deterministically and incur bounded errors on the constraints.

We may also assume without loss of generality that both $1/\epsilon_1$ and the number of iterations, $T = \Theta((\log n)/\epsilon_1^2)$, in the main loop of Algorithm 2, is a power of 2. We can apply Lemma 12 to obtain an upper bound, $B = O(\log\log n + \log(1/\epsilon_1))$, on the number of bits needed to store each α'_{eu}. In the case where the maximum degree Δ is very small, we can even truncate the bits without creating much error. Let $t = \min(B, \lceil \log_2(\Delta/\epsilon_2) \rceil)$. We round the α'-values up to the t'th bit after the decimal point. In other words, we set $\alpha_{eu}^{(0)} = \lceil 2^t \cdot \alpha'_{eu} \rceil / 2^t$ for every variable α_{eu}. If $t = B$, we are just setting $\alpha_{eu}^{(0)} = \alpha'_{eu}$.

Note that in the case $t = \lceil \log_2(\Delta/\epsilon_2) \rceil$, because $\tilde{D} \geq 1$, we have

$$\sum_{e\ni u} \alpha_{eu}^{(0)} \leq \sum_{e\ni u} \alpha'_{eu} + \deg(u) \cdot 2^{-\lceil \log_2(\Delta/\epsilon_2) \rceil} \leq (1+\epsilon)\tilde{D} + \epsilon_2 \leq (1+\epsilon_1)(1+\epsilon_2)\tilde{D} \ .$$

The algorithm (Algorithm 4) consists of t iterations. It processes the α-values bit-by-bit, from the t'th bit to the first bit after the decimal point. When it is processing the k'th bit, for each α_{eu}, we will round its k'th bit either up or down. Therefore, after we have processed the first bit in the last iteration, all the α-values are integers. Let $\alpha_{eu}(i)$ be the $(t-i+1)$'th bit of α_{eu} after the decimal point (i.e. the i'th rightmost bit after the initial rounding).

Let $\alpha_{eu}^{(k)}$ denote the value α_{eu} at the end of iteration k. During iteration k, if $\alpha_{eu}^{(k-1)}(k) = 1$, we will either need to round it up (set $\alpha_{eu}^{(k)} = \alpha_{eu}^{(k-1)} + 2^{-(t-k+1)}$) or round it down (set $\alpha_{eu}^{(k)} = \alpha_{eu}^{(k-1)} - 2^{-(t-k+1)}$) so that $\alpha_{eu}^{(k)}(k) = 0$.

Consider an edge $e = uv$. If $\alpha_{eu}^{(k-1)}(k) = 0$ or $\alpha_{ev}^{(k-1)}(k) = 0$, we will round both of their k'th bit down so that $\alpha_{eu}^{(k)}(k) = 0$ and $\alpha_{ev}^{(k)}(k) = 0$. All the remaining edges must be contained in the graph $G_k = (V, E_k)$, where $E_k = \{uv \mid \alpha_{eu}^{(k-1)}(k) = 1 \text{ and } \alpha_{ev}^{(k-1)}(k) = 1\}$. We will perform a deterministic directed splitting algorithm on G_k which we adapt from [22] to the CONGEST model. Let $\deg_k(u)$ denote the degree of v in G_k. The outcome of the algorithm is an orientation of the edges in E_k such that for each vertex u, $|\text{outdeg}_k(u) - \text{indeg}_k(u)| \leq \epsilon_3 \deg_k(u) + 12$.

Suppose that $e = uv$ is oriented from u to v. We will round $\alpha_{eu}^{(k-1)}$ up and round $\alpha_{ev}^{(k-1)}$ down. We do the opposite if it is oriented from v to u.

▶ **Lemma 13.** *Suppose that $\epsilon_3 \leq 1/4$. The $\{\alpha_{ev}\}$ values produced by Algorithm 4 satisfy the following properties. (1) $\alpha_{eu} \in \{0,1\}$ (2) For every edge $e = uv$, $\alpha_{eu} + \alpha_{ev} \geq 1$. (3) For every vertex u, $\sum_{e\ni u} \alpha_{eu} \leq (1+\epsilon_1)(1+\epsilon_2)(1+\epsilon_3)^t \tilde{D} + 16$.*

Proof. For (1), since we either round $\alpha_{ev}^{(k-1)}(k)$ up or down during iteration k, we have $\alpha_{ev}^{(k)}(k) = 0$ at the end of iteration k. Moreover, once $\alpha_{ev}^{(k)}(k)$ becomes 0, $\alpha_{ev}^{(k')}(k)$ remains 0 for $k' \geq k$. Therefore, at the end of iteration t, we have $\alpha_{ev}^{(t)}(i) = 0$ for $1 \leq i \leq t$. This implies $\alpha_{ev}^{(t)}$ is a non-negative integer. Since the final output, α_{ev}, is the minimum of 1 and $\alpha_{ev}^{(t)}$, we have $\alpha_{ev} \in \{0,1\}$.

Algorithm 4 Deterministic Rounding for Low Outdegree Orientation.

1: Obtain a feasible $\{\alpha'_{eu}\}_{eu}$ for $\textsc{Dual}((1+\epsilon_1)\tilde{D})$.
2: Set $\alpha^{(0)}_{eu} = \lceil 2^t \cdot \alpha'_{eu} \rceil / 2^t$ for every α_{eu}.
3: **for** $k = 1 \ldots t$ **do**
4: **for** every edge $e = uv$ s.t. $\alpha^{(k-1)}_{eu}(k) = 0$ or $\alpha^{(k-1)}_{ev}(k) = 0$ **do**
5: Set $\alpha^{(k)}_{eu} = \alpha^{(k-1)}_{eu}$ and then set $\alpha^{(k)}_{eu}(k) = 0$. ▷ round the k'th bit down
6: Set $\alpha^{(k)}_{ev} = \alpha^{(k-1)}_{ev}$ and then set $\alpha^{(k)}_{ev}(k) = 0$.
7: Let $G_k = (V, E_k)$, where $E_k = \{uv \mid \alpha^{(k-1)}_{eu}(k) = 1 \text{ and } \alpha^{(k-1)}_{ev}(k) = 1\}$.
8: Obtain a directed splitting of G_k whose discrepancy is at most $\epsilon_3 \deg_k(u) + 12$ for each vertex u.
9: **for** every edge $e = uv$ where u is oriented toward v **do**
10: Set $\alpha^{(k)}_{eu} = \alpha^{(k-1)}_{eu} + 2^{-(t-k+1)}$. ▷ round up
11: Set $\alpha^{(k)}_{ev} = \alpha^{(k-1)}_{ev} - 2^{-(t-k+1)}$. ▷ round down
12: **for** each $e = uv$ **do**
13: Set $\alpha_{eu} = \min(\alpha^{(t)}_{eu}, 1)$.

We show (2) inductively. In the beginning, since $\alpha^{(0)}_{eu} \geq \alpha'_{eu}$, we must have $\alpha^{(0)}_{eu} + \alpha^{(0)}_{ev} \geq 1$. Suppose that $\alpha^{(k-1)}_{eu} + \alpha^{(k-1)}_{ev} \geq 1$. During iteration k, if $\alpha^{(k-1)}_{eu}(k) = 0$ and $\alpha^{(k-1)}_{ev}(k) = 0$, then $\alpha^{(k)}_{eu} = \alpha^{(k-1)}_{eu}$ and $\alpha^{(k)}_{ev} = \alpha^{(k-1)}_{ev}$ and so $\alpha^{(k)}_{eu} + \alpha^{(k)}_{ev} \geq 1$.

If $\alpha^{(k-1)}_{eu}(k) = 1$ and $\alpha^{(k-1)}_{ev}(k) = 0$, then it must be the case that $\alpha^{(k-1)}_{eu} + \alpha^{(k-1)}_{ev} \geq 1 + 2^{-(t-k+1)}$. After rounding $\alpha^{(k-1)}_{eu}$ down, we still have $\alpha^{(k)}_{eu} + \alpha^{(k)}_{ev} \geq 1$. The case for $\alpha^{(k-1)}_{eu}(k) = 0$ and $\alpha^{(k-1)}_{ev}(k) = 1$ is symmetric.

The remaining case is when $\alpha^{(k-1)}_{eu}(k) = 1$ and $\alpha^{(k-1)}_{ev}(k) = 1$. In this case, $e \in G_k$. We must have $\alpha^{(k)}_{eu} + \alpha^{(k)}_{ev} = \alpha^{(k-1)}_{eu} + \alpha^{(k-1)}_{ev} \geq 1$, since one of them is rounded up and the other is rounded down.

For (3), let $D_0 = (1+\epsilon_1)(1+\epsilon_2)\tilde{D}$ and $D_k = (1+\epsilon_3)D_{k-1} + 12 \cdot 2^{-(t-k+1)}$ for $k \geq 1$. We will show by induction that $\sum_{e \ni u} \alpha^{(k)}_{eu} \leq D_k$. For the base case, initially, we argued that $\sum_{e \ni u} \alpha^{(0)}_{eu} \leq (1+\epsilon_1)(1+\epsilon_2)\tilde{D}$. For $k \geq 1$, note that the increase on the quantity $\sum_{e \ni u} \alpha_{eu}$ during iteration k is at most $2^{-(t-k+1)} \cdot (\epsilon_3 \cdot \deg_k(u) + 12)$. Therefore,

$$\begin{aligned}\sum_{e \ni u} \alpha^{(k)}_{eu} &\leq 2^{-(t-k+1)} \cdot (\epsilon_3 \cdot \deg_k(u) + 12) + \sum_{e \ni u} \alpha^{(k-1)}_{eu} \\ &\leq \epsilon_3 D_{k-1} + 12 \cdot 2^{-(t-k+1)} + D_{k-1} \leq (1+\epsilon_3)D_{k-1} + 12 \cdot 2^{-(t-k+1)} = D_k \ .\end{aligned}$$

The second inequality follows since $2^{-(t-k+1)} \deg_k(u) \leq \sum_{e \ni u} \alpha^{(k-1)}_{eu} \leq D_{k-1}$. This completes the induction. Since $\epsilon_3 \leq 1/4$, in the end, we have

$$\begin{aligned}\sum_{e \ni u} \alpha_{eu} \leq \sum_{e \ni u} \alpha^{(t)}_{eu} \leq D_t &\leq (1+\epsilon_1)(1+\epsilon_2)(1+\epsilon_3)^t \tilde{D} + 12 \cdot (1/2) \cdot \sum_{k=0}^{t-1} \left(\frac{1+\epsilon_3}{2} \right)^k \\ &\leq (1+\epsilon_1)(1+\epsilon_2)(1+\epsilon_3)^t \tilde{D} + 16 \ .\end{aligned}$$

◀

▶ **Theorem 4.** *Given an integer $\tilde{D} \geq D$, for any $32/\tilde{D} \leq \epsilon \leq 1/4$, there exists a deterministic algorithm in the CONGEST model that computes a $(1+\epsilon)\tilde{D}$-orientation and runs in*

$$O\left(\frac{\log n}{\epsilon^2} + (\min(\log\log n, \log\Delta) + \log(1/\epsilon))^{2.71} \cdot (1/\epsilon)^{1.71} \cdot \log^2 n \right) \leq \tilde{O}((\log^2 n)/\epsilon^2) \textit{ rounds.}$$

Proof. We set parameters $\epsilon_1 = \epsilon_2 = \epsilon/8$ and $\epsilon_3 = \epsilon/(4t)$. Run Algorithm 4 to obtain integral $\{\alpha_{eu}\}$ that satisfy $\alpha_{eu} + \alpha_{ev} \geq 1$ for every $e = uv$ and $\sum_{e \ni u} \alpha_{eu} \leq (1+\epsilon_1)(1+\epsilon_2)(1+\epsilon_3)^t \tilde{D} + 16$ for every u. For each edge $e = uv$, if $\alpha_{eu} = 1$ then we orient e from u to v. Otherwise, we

orient e from v to u. Since $16 \leq \epsilon\tilde{D}/2$, the out-degree of each vertex is upper bounded by:

$$\sum_{e \ni u} \alpha_{eu} \leq (1+\epsilon_1)(1+\epsilon_2)(1+\epsilon_3)^t \tilde{D} + 16 \leq (1+\epsilon/2)\tilde{D} + 16 \leq (1+\epsilon)\tilde{D} \ .$$

The running time for Algorithm 4 consists of the following. The number of rounds to obtain a fractional solution is $O((\log n)/\epsilon_1^2) = O((\log n)/\epsilon^2)$. Then, it consists of $t = \min(\log(\Delta/\epsilon_2), B) = O(\min(\log\log n, \log\Delta) + \log(1/\epsilon))$ iterations. Each iteration invokes a directed splitting procedure that runs in $O((1/\epsilon_3)^{1.71} \cdot \log^2 n)$ rounds by Theorem 16. Recall that $\epsilon_3 = \epsilon/(4t)$, the total number of rounds is therefore:

$$O\left(\frac{\log n}{\epsilon^2} + t \cdot (1/\epsilon_3)^{1.71} \cdot \log^2 n\right) = O\left(\frac{\log n}{\epsilon^2} + t^{2.71} \cdot (1/\epsilon)^{1.71} \cdot \log^2 n\right) \ . \qquad ◀$$

4.1 Distributed Splitting in the CONGEST Model

Given a graph $G = (V, E)$, a *weak $f(v)$-orientation* of G is an orientation of the edges in G such that there are at least $f(v)$ outgoing edges for each $v \in V$. In order to adapt the algorithm of [22] for directed splitting, we need an algorithm for weak $\lfloor \deg(v)/3 \rfloor$-orientation in the CONGEST model. The previous algorithms [22, 24] for weak $\lfloor \deg(v)/3 \rfloor$-orientation requires finding short cycles for containing each edge. They are not adaptable to the CONGEST model. We use an augmenting path approach for finding a weak $\lfloor \deg(v)/3 \rfloor$-orientation instead.

▶ **Lemma 14.** *There exists a deterministic distributed algorithm that computes a weak $\lfloor \deg(v)/3 \rfloor$-orientation in $O(\log^2 n)$ rounds in the* CONGEST *model.*

Proof. First we construct a new graph G' as follows: Split every vertex v into $\lceil \deg(v)/3 \rceil$ copies. Attach evenly the edges to each copy of the vertex so every copy except possibly the last gets 3 edges and the last copy gets $\deg(v) - 3 \cdot (\lceil \deg(v)/3 \rceil - 1)$ edges. Given an orientation of G', we call a vertex v a *sink* **if it has exactly 3 incoming edges**. Clearly, a sinkless orientation (i.e. an orientation where there are no sinks) in G' corresponds to a weak $\lfloor \deg(v)/3 \rfloor$-orientation of G. Moreover, one round in G' can be emulated in G by using one round.

Now we start with an arbitrary orientation of G'. Some vertices might be sinks. We will use an augmenting path approach to eliminate sinks.

Divide the vertices into the following three types. Type I vertices are the sinks. Type II vertices are those u such that $\deg(u) = 3$ and $\mathrm{indeg}(u) = 2$. Type III vertices are those with $\mathrm{indeg}(u) \leq 1$ or $\deg(u) < 3$.

An augmenting path is a path $P = (u_1, \ldots, u_l)$ such that:

1. u_1 is a Type I vertex. u_l is a Type III vertex. u_i is a Type II vertex for $1 < i < l$.
2. u_{i+1} is oriented towards u_i for $1 \leq i < l$.

If P is an augmenting path then flipping P will make u_1 no longer a sink. Moreover, it will not create any new sink.

Consider a Type I vertex u. An augmenting path of length $O(\log n)$ starting from u can be found as follows. Let $L_0 = \{u\}$. Given L_{i-1}, let L_i be the set of incoming neighbors of every vertex in L_{i-1}. If L_i contains a Type III vertex, then an augmenting path is found. Otherwise, it must be the case that L_i contains all Type II vertices. Since there are at least $2 \cdot |L_{i-1}|$ incoming edges from L_i to L_{i-1} ($\mathrm{indeg}(u) = 2$ for $u \in L_{i-1}$) and Type II vertices have out-degree 1, we have $|L_i| \geq 2 \cdot |L_{i-1}|$. Hence, this process can only continue for at most $O(\log n)$ times.

Every Type I vertex would be able to find an augmenting paths of length $O(\log n)$ this way. Moreover, note that these augmenting paths can only overlap at their ending vertex, since the intermediate Type II vertices have out-degree 1. Each ending vertex is a Type

III vertex, which can only be the ending vertex of at most 3 augmenting paths since it has at most 3 outgoing edges. It selects an arbitrary augmenting path to accept. Therefore, at least 1/3 fraction of augmenting paths will be accepted. We flip along the accepted augmenting paths to fix Type I vertices so that we eliminate at least 1/3 fraction of the sinks. Therefore, it takes $O(\log n)$ repetitions to eliminate all the sinks. The total number of rounds is $O(\log n \cdot \log n) = O(\log^2 n)$. The process can be easily implemented in the CONGEST model. ◀

Before we describe how to adapt the splitting algorithm of [22], we need to introduce the following definition. Given a function $\delta : V \to \mathbb{R}_{\geq 0}$ and $\lambda \in \mathbb{Z}_{\geq 0}$. A (δ, λ)-*path decomposition* $\mathcal{P}$ is a partition of E into paths $P_1, \ldots, P_\rho$ such that

1. Each vertex v is an endpoint of at most $\delta(v)$ paths.
2. Each path P_i is of length at most λ.

Given a (δ, λ)-path decomposition $\mathcal{P}$. The virtual graph $G_{\mathcal{P}} = (V, E_{\mathcal{P}})$ consists of exactly ρ edges, where each path $P_i = (v_{i,start}, \ldots, v_{i,end})$ corresponds to an edge $(v_{i,start}, v_{i,end})$ in $E_{\mathcal{P}}$. Lemma 15 is the adaption of [22, Lemma 2.11] to the CONGEST model.

▶ **Lemma 15.** *Assume that $T(n, \Delta) \geq \log n$ is the running time of an algorithm $\mathcal{A}$ that finds a weak $\lfloor \deg(v)/3 \rfloor$-orientation in the* CONGEST *model. Then for any positive integer i, there is a deterministic distributed algorithm $\mathcal{A}$ that finds a $((\frac{2}{3})^i \cdot \deg(v) + 12, 2^i)$-path decomposition $\mathcal{P}$ in time $O(2^i \cdot T(n, \Delta))$ in the* CONGEST *model.*

The reason of why such an adaptation works is due to the fact that all the paths in $\mathcal{P}$ are disjoint. Therefore, one round in $G_{\mathcal{P}}$ can be simulated using $O(\lambda)$ rounds in G in the CONGEST model, given $\mathcal{P}$ is a (δ, λ)-path decomposition.

For completeness, we explain how the path-decomposition in Lemma 15 can be obtained. Let $\mathcal{P}_0$ denote the initial path decomposition where each path is a single edge in E. Given $\mathcal{P}_{i-1}$, $\mathcal{P}_i$ can be built as follows: Obtain a $\lfloor \deg(v)/3 \rfloor$-orientation on $G_{\mathcal{P}_{i-1}}$. For each vertex u, group the outgoing edges into at least $\lfloor \lfloor \deg(u)/3 \rfloor / 2 \rfloor$ pairs. For each such edge pair (u, x) and (u, w), we reverse the path that corresponds to (u, x) and append it with (u, w). The new degree of u becomes at most $\deg(u) - 2 \cdot \lfloor \lfloor \deg(u)/3 \rfloor / 2 \rfloor \leq \frac{2}{3} \deg(u) + 4$. Therefore, if $\mathcal{P}_{i-1}$ is a $(\delta, \lambda(v))$-decomposition then $\mathcal{P}_i$ is a $(\frac{2}{3} \cdot \delta(v) + 4, 2\lambda(v))$ decomposition.

Since $\mathcal{P}_0$ is a $(\deg(v), 1)$-path decomposition, $\mathcal{P}_i$ a $(z_i(v), 2^i)$-path decomposition, where

$$z_i(v) = \left(\frac{2}{3}\right)^i \cdot \deg(v) + 4 \sum_{k=0}^{i-1} \left(\frac{2}{3}\right)^k \leq \left(\frac{2}{3}\right)^i \cdot \deg(v) + 12 \ .$$

The running time of the j'th iteration is $O(2^j \cdot T(n, \Delta))$. Therefore, the total running time is $\sum_{j=1}^{i} O(2^j \cdot T(n, \Delta)) = O(2^i \cdot T(n, \Delta))$. By setting $i = \log_{3/2}(1/\epsilon)$ and $T(n, \Delta) = \log^2 n$ from Lemma 14, we get a $(\epsilon \cdot \deg(v) + 12, (1/\epsilon)^{\log_{3/2} 2})$-path decomposition in $O((1/\epsilon)^{\log_{3/2} 2} \cdot \log^2 n)$ rounds.

Suppose $\mathcal{P}$ is a (δ, λ)-path decomposition. If we orient the edges on each path in $\mathcal{P}$ in consistent with the direction of the path, then we must have $|\operatorname{outdeg}(v) - \operatorname{indeg}(v)| \leq \delta(v)$. Therefore, we obtain the following theorem.

▶ **Theorem 16.** *For $\epsilon > 0$, there exists a $O((1/\epsilon)^{1.71} \cdot \log^2 n)$ rounds deterministic algorithm in the* CONGEST *model that computes an orientation such that for each vertex v, $|\operatorname{outdeg}(v) - \operatorname{indeg}(v)| \leq \epsilon \deg(v) + 12$.*

References

1. Mohamad Ahmadi, Fabian Kuhn, and Rotem Oshman. Distributed approximate maximum matching in the CONGEST model. In *Proc. 32nd International Symposium on Distributed Computing (DISC)*, pages 6:1–6:17, 2018.
2. Sanjeev Arora, Elad Hazan, and Satyen Kale. The multiplicative weights update method: a meta-algorithm and applications. *Theory of Computing*, 8(6):121–164, 2012. `doi:10.4086/toc.2012.v008a006`.
3. Baruch Awerbuch. Complexity of network synchronization. *J. ACM*, 32(4):804–823, 1985.
4. Bahman Bahmani, Ashish Goel, and Kamesh Munagala. Efficient primal-dual graph algorithms for mapreduce. In *WAW*, volume 8882 of *Lecture Notes in Computer Science*, pages 59–78. Springer, 2014.
5. Bahman Bahmani, Ravi Kumar, and Sergei Vassilvitskii. Densest subgraph in streaming and mapreduce. *PVLDB*, 5(5):454–465, 2012.
6. Leonid Barenboim and Michael Elkin. Sublogarithmic distributed MIS algorithm for sparse graphs using nash-williams decomposition. *Distributed Computing*, 22(5-6):363–379, 2010.
7. Edvin Berglin and Gerth Stølting Brodal. A simple greedy algorithm for dynamic graph orientation. *Algorithmica*, 82(2):245–259, 2020.
8. Alex Beutel, Wanhong Xu, Venkatesan Guruswami, Christopher Palow, and Christos Faloutsos. Copycatch: Stopping group attacks by spotting lockstep behavior in social networks. In *Proceedings of the 22Nd International Conference on World Wide Web (WWW)*, pages 119–130, 2013.
9. Sayan Bhattacharya, Monika Henzinger, Danupon Nanongkai, and Charalampos E. Tsourakakis. Space- and time-efficient algorithm for maintaining dense subgraphs on one-pass dynamic streams. In *Proc. 47th ACM Symposium on Theory of Computing (STOC)*, pages 173–182, 2015.
10. Gerth Stølting Brodal and Rolf Fagerberg. Dynamic representations of sparse graphs. In *Proc. 6th International Workshop on Algorithms and Data Structures (WADS)*, pages 342–351, 1999.
11. Yi-Jun Chang and Thatchaphol Saranurak. Improved distributed expander decomposition and nearly optimal triangle enumeration. *CoRR*, abs/1904.08037, 2019.
12. Moses Charikar. Greedy approximation algorithms for finding dense components in a graph. In *APPROX*, volume 1913 of *Lecture Notes in Computer Science*, pages 84–95. Springer, 2000.
13. Jie Chen and Yousef Saad. Dense subgraph extraction with application to community detection. *IEEE Trans. on Knowl. and Data Eng.*, 24(7):1216–1230, 2012.
14. Yon Dourisboure, Filippo Geraci, and Marco Pellegrini. Extraction and classification of dense communities in the web. In *Proc. 16th International Conference on World Wide Web (WWW)*, pages 461–470, 2007.
15. Hossein Esfandiari, MohammadTaghi Hajiaghayi, and David P. Woodruff. Applications of uniform sampling: Densest subgraph and beyond. *CoRR*, abs/1506.04505, 2015.
16. Manuela Fischer. Improved deterministic distributed matching via rounding. In *31st International Symposium on Distributed Computing (DISC)*, pages 17:1–17:15, 2017.
17. Manuela Fischer, Mohsen Ghaffari, and Fabian Kuhn. Deterministic distributed edge-coloring via hypergraph maximal matching. In *Proc. 58th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 180–191, 2017.
18. Eugene Fratkin, Brian T. Naughton, Douglas Brutlag, and Serafim Batzoglou. Motifcut: Regulatory motifs finding with maximum density subgraphs. *Bioinformatics (Oxford, England)*, 22:e150–7, August 2006.
19. Harold N. Gabow and Herbert H. Westermann. Forests, frames, and games: Algorithms for matroid sums and applications. *Algorithmica*, 7(1):465, June 1992.
20. G. Gallo, M. D. Grigoriadis, and R. E. Tarjan. A fast parametric maximum flow algorithm and applications. *SIAM J. Comput.*, 18(1):30–55, 1989.

21 Mohsen Ghaffari, David G. Harris, and Fabian Kuhn. On derandomizing local distributed algorithms. In *Proc. 59th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 662–673, 2018.

22 Mohsen Ghaffari, Juho Hirvonen, Fabian Kuhn, Yannic Maus, Jukka Suomela, and Jara Uitto. Improved distributed degree splitting and edge coloring. In *DISC*, pages 19:1–19:15, 2017.

23 Mohsen Ghaffari, Silvio Lattanzi, and Slobodan Mitrović. Improved parallel algorithms for density-based network clustering. In *Proc. 36th International Conference on Machine Learning (ICML)*, volume 97, pages 2201–2210, 2019.

24 Mohsen Ghaffari and Hsin-Hao Su. Distributed degree splitting, edge coloring, and orientations. In *Proc. of 28th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2505–2523, 2017.

25 David Gibson, Ravi Kumar, and Andrew Tomkins. Discovering large dense subgraphs in massive graphs. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB)*, pages 721–732, 2005.

26 A. V. Goldberg. Finding a maximum density subgraph. Technical Report UCB/CSD-84-171, EECS Department, University of California, Berkeley, 1984.

27 Anupam Gupta, Amit Kumar, and Cliff Stein. Maintaining assignments online: Matching, scheduling, and flows. In *Proc. 25th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, page 468–479, 2014.

28 David G. Harris. Distributed approximation algorithms for maximum matching in graphs and hypergraphs. In *Proc. 60th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 700–724, 2019.

29 R. Kannan and V. Vinay. Analyzing the structure of large graphs. *Technical report*, 1999.

30 Samir Khuller and Barna Saha. On finding dense subgraphs. In *Proc. 36th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 597–608, 2009.

31 Tsvi Kopelowitz, Robert Krauthgamer, Ely Porat, and Shay Solomon. Orienting fully dynamic graphs with worst-case time bounds. In Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias, editors, *Proc. 41st International Colloquium on Automata, Languages and Programming (ICALP)*, pages 532–543, 2014.

32 Łukasz Kowalik. Approximation scheme for lowest outdegree orientation and graph density measures. In *Proc. of the 17th International Conference on Algorithms and Computation (ISAAC)*, pages 557–566, 2006.

33 Nathan Linial and Michael E. Saks. Low diameter graph decompositions. *Combinatorica*, 13(4):441–454, 1993.

34 Andrew McGregor, David Tench, Sofya Vorotnikova, and Hoa T. Vu. Densest subgraph in dynamic graph streams. In *MFCS (2)*, volume 9235 of *Lecture Notes in Computer Science*, pages 472–482. Springer, 2015.

35 Gary L. Miller, Richard Peng, and Shen Chen Xu. Parallel graph decompositions using random shifts. In *SPAA*, pages 196–203. ACM, 2013.

36 C. St.J. A. Nash-Williams. Decomposition of finite graphs into forests. *Journal of the London Mathematical Society*, s1-39(1):12–12, 1964.

37 Jean-Claude Picard and Maurice Queyranne. A network flow solution to some nonlinear 0-1 programming problems, with applications to graph theory. *Networks*, 12(2):141–159, 1982.

38 Barna Saha, Allison Hoch, Samir Khuller, Louiqa Raschid, and Xiao-Ning Zhang. Dense subgraphs with restrictions and applications to gene annotation graphs. In *Research in Computational Molecular Biology*, pages 456–472, 2010.

39 Atish Das Sarma, Ashwin Lall, Danupon Nanongkai, and Amitabh Trehan. Dense subgraphs on dynamic networks. In *DISC*, volume 7611 of *Lecture Notes in Computer Science*, pages 151–165. Springer, 2012.

40 Saurabh Sawlani and Junxing Wang. Near-optimal fully dynamic densest subgraph. In *Proc. 52th ACM Symposium on Theory of Computing (STOC)*, 2020. To appear.

41 Hsin-Hao Su and Hoa T. Vu. Distributed dense subgraph detection and low outdegree orientation. *CoRR*, abs/1907.12443, 2019.

42 M. Thorup. Fully-dynamic min-cut. *Combinatorica*, 27(1):91–127, 2007. doi:10.1007/s00493-007-0045-2.

43 Neal E. Young. Randomized rounding without solving the linear program. In *SODA*, pages 170–178. ACM/SIAM, 1995.

Local Conflict Coloring Revisited: Linial for Lists

Yannic Maus
Technion – Israel Institute of Technology, Haifa, Israel
yannic.maus@cs.technion.ac.il

Tigran Tonoyan
Technion – Israel Institute of Technology, Haifa, Israel
ttonoyan@gmail.com

Abstract

Linial's famous color reduction algorithm reduces a given m-coloring of a graph with maximum degree Δ to a $O(\Delta^2 \log m)$-coloring, in a single round in the LOCAL model. We show a similar result when nodes are restricted to choose their color from a list of allowed colors: given an m-coloring in a directed graph of maximum outdegree β, if every node has a list of size $\Omega(\beta^2(\log \beta + \log\log m + \log\log |\mathcal{C}|))$ from a color space $\mathcal{C}$ then they can select a color in two rounds in the LOCAL model. Moreover, the communication of a node essentially consists of sending its list to the neighbors. This is obtained as part of a framework that also contains Linial's color reduction (with an alternative proof) as a special case. Our result also leads to a *defective list coloring* algorithm. As a corollary, we improve the state-of-the-art *truly local* $(deg + 1)$-list coloring algorithm from Barenboim et al. [PODC'18] by slightly reducing the runtime to $O(\sqrt{\Delta \log \Delta}) + \log^* n$ and significantly reducing the message size (from huge to roughly Δ). Our techniques are inspired by the *local conflict coloring* framework of Fraigniaud et al. [FOCS'16].

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases distributed graph coloring, list coloring, low intersecting set families

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.16

Related Version A full version of the paper is available at `https://arxiv.org/abs/2007.15251`.

Funding This project was supported by the European Union's Horizon 2020 Research and Innovation Programme under grant agreement no. 755839.

1 Introduction

Symmetry breaking problems are a cornerstone of distributed graph algorithms in the LOCAL model.[1] A central question in the area asks: *How fast can these problems be solved in terms of the maximum degree Δ, when the dependence on n is as mild as $O(\log^* n)$?* [12]. That is, we are looking for *truly local* algorithms, with complexity of the form $f(\Delta) + O(\log^* n)$.[2] The $O(\log^* n)$ term is unavoidable due to the seminal lower bound by Linial [41] that, via simple reductions, applies to most typical symmetry breaking problems. $(\Delta + 1)$-*vertex coloring* and *maximal independent set (MIS)* are the key symmetry breaking problems. Both can be solved by simple centralized greedy algorithms (in particular they are always solvable), and even more importantly in a distributed context, any partial solution (e.g. a partial coloring) can be extended to a complete solution. The complexity of MIS is settled up to constant factors with $f(\Delta) = \Theta(\Delta)$, by the algorithm from [14] and a recent breakthrough lower bound by

[1] In the LOCAL model [41, 46] a communication network is abstracted as an n-node graph $G = (V, E)$ with unique $O(\log n)$-bit identifiers. Communications happen in synchronous rounds. Per round, each node can send one (unbounded size) message to each of its neighbors. At the end, each node should know its own part of the output, e.g., its own color. In the CONGEST model, there is a limitation of $O(\log n)$ bits per message.

[2] Not all symmetry breaking problems admit such a runtime, e.g., Δ-coloring and sinkless orientation [17].

© Yannic Maus and Tigran Tonoyan;
licensed under Creative Commons License CC-BY
34th International Symposium on Distributed Computing (DISC 2020).
Editor: Hagit Attiya; Article No. 16; pp. 16:1–16:18
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Balliu et al. [3]. In contrast, the complexity of vertex coloring, despite being among the most studied distributed graph problems [12], remains widely open, with the current best upper bound $f(\Delta) = \tilde{O}(\sqrt{\Delta})$ and lower bound $f(\Delta) = \Omega(1)$. While most work has focused on the $(\Delta + 1)$-coloring problem, recent algorithms, e.g., [8, 23, 13, 37], rely on the more general *list coloring problem* as a subroutine, where each vertex v of a graph G has a list $L_v \subseteq \mathcal{C}$ of colors from a colorspace $\mathcal{C}$, and the objective is to compute a proper vertex coloring, but each vertex has to select a color from its list. Again, a natural case is the always solvable $(deg + 1)$-*list coloring problem*, where the list of each vertex is larger than its degree. Our paper contributes to the study of list coloring problems. To set the stage for our results, let us start with an overview of truly local coloring algorithms.

In [41], besides the mentioned lower bound, Linial also showed that $O(\Delta^2)$-coloring can be done in $O(\log^* n)$ rounds. Szegedy and Vishwanathan [49] improved the runtime to $\frac{1}{2}\log^* n + O(1)$ rounds, and showed that in additional $O(\Delta \cdot \log \Delta)$ rounds, the $O(\Delta^2)$-coloring could be reduced to $(\Delta+1)$-coloring. The latter result was rediscovered by Kuhn and Wattenhofer [39]. Barenboim, Elkin and Kuhn used *defective coloring* for partitioning the graph into low degree subgraphs and coloring in a divide-and-conquer fashion, and brought the complexity of $(\Delta + 1)$-coloring down to $O(\Delta + \log^* n)$ [14]. A simpler algorithm with the same runtime but without using defective coloring was obtained recently in [13]. All of these results also hold for $(deg + 1)$-list coloring, since given a $O(\Delta)$-coloring, one can use it as a "schedule" for computing a $(deg + 1)$-list coloring in $O(\Delta)$ additional rounds. Meantime, two sub-linear in Δ algorithms were already published [8, 23]. They both used *low outdegree colorings*, or *arb-defective colorings*, introduced by Barenboim and Elkin in [10], for graph partitioning purposes. The basic idea here is similar to the defective coloring approach, with the difference that the graph is partitioned into directed subgraphs with low *maximum outdegree.* In [13] they also improved and simplified the computation of low outdegree colorings, which led to improved runtime and simplification of that component in the mentioned sublinear algorithms. As a result, the currently fastest algorithm in CONGEST needs $O(\Delta^{3/4} + \log^* n)$ rounds ([8]+[13]),[3] while the fastest one in LOCAL needs $O(\sqrt{\Delta \log \Delta} \log^* \Delta + \log^* n)$ rounds ([23]+[13]).

Let us take a better look at the latter result, which is the closest to our paper. The algorithm consists of two main ingredients: (1) an algorithm that partitions a given graph into $p = O(\Delta/\beta)$ subgraphs, each equipped with a $\beta = O(\sqrt{\Delta/\log \Delta})$-outdegree orientation, in $O(p) + \frac{1}{2}\log^* n$ rounds [13], (2) a list coloring subroutine that gives rise to the following [23]:

▶ **Theorem 1.1** ([23]). *In a directed graph with max. degree Δ, max. outdegree β, and an input m-coloring, list coloring with lists L_v of size $|L_v| \geq 10\beta^2 \ln \Delta$ from any color space can be solved in $O(\log^*(\Delta + m))$ rounds in LOCAL.*

The two ingredients are combined to give a $(deg + 1)$-list coloring of the input graph G [23]. First, partition G using (1), iterate through the p directed subgraphs, and for each of them, let uncolored nodes refine their lists by removing colors taken by a neighbor, and use (2) to color nodes that still have sufficiently large lists ($\geq 10\beta^2 \ln \Delta$). With a fine grained choice of β it is ensured that every node v with (refined) list size greater than $\Delta/2$ is colored. Thus, after iterating through all p subgraphs, all uncolored nodes have list size at most $\Delta/2$, and because each vertex always has *more colors in its list than uncolored neighbors*, the max degree of the graph induced by uncolored nodes is at most half of that of G. Thus, we can apply the partition-then-list-color paradigm recursively to the subgraph induced by uncolored nodes to complete the coloring. The runtime is dominated by the first level of recursion.

[3] For more colors, i.e., $(1 + \varepsilon)\Delta$-coloring, [8] gives runtime $O(\sqrt{\Delta} + \log^* n)$ in CONGEST.

The strength of the partitioning algorithm above is that it is conceptually simple and works with small messages. In contrast, the algorithm from Thm. 1.1 is conceptually complicated and uses gigantic messages. This complication might be due to the generality of the addressed setting as, in fact, [23] studies the more general local conflict coloring problem, and Thm. 1.1 is only a special case. In *local conflict coloring*, each edge of the given graph is equipped with an arbitrary conflict relation between colors and this relation may vary across different edges. This framework is also leveraged to achieve the colorspace size independence of Thm. 1.1 (see the technical overview below). It was not clear prior to our work whether the situation simplifies significantly if one restricts to ordinary list coloring because, even if the input to the algorithm in Thm. 1.1 is a list coloring problem, the intermediate stages of the algorithm fall back to the more general local conflict coloring.

The overarching goal of our paper is providing deeper understanding of the remarkable framework of [23], better connecting it with classic works in the area, and obtaining a simpler list coloring algorithm that also uses smaller messages, by moderately sacrificing generality.

1.1 Our Contribution

Our main result is a simple algorithm that yields the following theorem.

▶ **Theorem 1.2** (Linial for Lists). *In a directed graph with max. degree Δ, max. outdegree β, and an input m-coloring, list coloring with lists L_v from a color space $\mathcal{C}$ and of size $|L_v| \geq l_0 = 4e\beta^2(4\log\beta + \log\log|\mathcal{C}| + \log\log m + 8)$ can be solved in 2 rounds in LOCAL. Each node sends $l_0 + 1$ colors in the first round, and a l_0/β^2-bit message in the second.*

The name “Linial for Lists” stems from the fact that Thm. 1.2 is a “list version” of one of the cornerstones of distributed graph coloring, Linial’s color reduction, which says that an m-coloring can be reduced to a $(5\Delta^2\log m)$-coloring in a single round [41]. Moreover, our framework is itself a *natural generalization of Linial’s approach of cover-free set families.* Applied to equal lists, it yields an alternative proof of Linial’s color reduction, in the form of a *greedy construction of cover-free families* (Linial proved their existence using the probabilistic method [41]; he also used an alternative construction from [21] via polynomials over finite fields, which however yields a weaker color reduction for $m \gg \Delta$) (see Sec. 2.3 and Sec. 5).

Compared with Thm. 1.1, we lose colorspace independence, and our algorithm does not extend to general local conflict coloring (although we use a kind of conflict coloring in the process). In exchange, we eliminate Δ from the bound on the list size, reduce the runtime to exactly 2 rounds, and dramatically reduce message size. This is achieved by a non-trivial paradigm shift in the local conflict coloring framework (see the technical overview below). The runtime cannot be reduced to 1 round, due to a lower bound of [49] (see Sec. 5).

Combining Thm. 1.2 with the partitioning algorithm of [13] as outlined above gives us a $(deg + 1)$-list coloring algorithm. Note that any improvement in the list size bound in Thm. 1.2 (with little increase in runtime) would yield a faster $(deg+1)$-list coloring algorithm.

▶ **Theorem 1.3** ($(deg + 1)$-List Coloring). *In a graph with max. degree Δ, $(deg + 1)$-list coloring with lists $L_v \subseteq \mathcal{C}$ from a color space of size $|\mathcal{C}| = 2^{\text{poly}(\Delta)}$ [4] can be solved in $O(\sqrt{\Delta\log\Delta}) + \frac{1}{2}\cdot\log^* n$ rounds in LOCAL. Furthermore, each node only needs to broadcast to its neighbors a single non-CONGEST message consisting of a subset of its list.*

[4] We use the notation $\text{poly}(X) = O(X^c)$, for an absolute constant c, and $\tilde{O}(X) = X\cdot\text{poly}(\log X)$.

The bound on the color space size stems from the color space dependence in Theorem 1.2. As discussed in Sec. 5, it is possible to trade color space dependence with runtime in Theorem 1.2, which could improve or suppress the bound in Theorem 1.3. That, however, comes with the cost of having huge messages.

Theorem 1.3 immediately provides the fastest known truly local $(\Delta+1)$-coloring algorithm in LOCAL. Below we list further implications of our framework.

- **CONGEST (see Cor. 4.2):** We obtain an improved $(\Delta+1)$-coloring algorithm in a *low degree* regime in CONGEST. In particular, if $\Delta = \tilde{O}(\log n)$ then $(\Delta+1)$-coloring (more generally, $(deg+1)$-list coloring with colorspace of size $|\mathcal{C}| = \text{poly}(\Delta)$) can be solved in $\tilde{O}(\sqrt{\Delta}) + \frac{1}{2} \cdot \log^* n$ rounds in CONGEST. Generally, if one allows messages of size B, this runtime holds for degree up to $\Delta = \tilde{O}(B)$. On the other hand, if $\Delta = \Omega(\log^{2+\varepsilon} n)$, for an arbitrarily small constant $\varepsilon > 0$, an algorithm from recent work [37] achieves runtime $O(\sqrt{\Delta})$ in CONGEST (if one recasts their dependency on n as a Δ-dependency). Thus, only for the regime of $\Delta \in \Omega(\log^{1+\epsilon} n) \cap O(\log^{2+\epsilon} n)$ we do not have an algorithm with runtime $\tilde{O}(\sqrt{\Delta})$ in CONGEST (with the current best being $O(\Delta^{3/4} + \log^* n)$ due to [8]).
- **Defective list coloring:** Our framework extends to *d-defective list coloring*, that is, list coloring where each node v can have at most d neighbors with the same colors as v: If lists are of size $\Omega\big((\Delta/(d+1))^2 \cdot (\log\Delta + \log\log|\mathcal{C}| + \log\log m)\big)$ we can compute a d-defective list coloring in 2 rounds in LOCAL. The result can be seen as the "list variant" of a defective coloring result in [36]. While we are not aware of an immediate application, defective list coloring with a better "colors vs. defect" tradeoff (d vs. $O(\Delta/d)$) for *line graphs* has recently been used to obtain a *edge-coloring* algorithm with complexity $\text{quasi poly} \log\Delta + O(\log^* n)$ [6]. See the full paper [42] for the formal statement and proof.
- **Δ-coloring:** The improvements obtained in Thm. 1.3 also imply respective improvements for several Δ*-coloring* algorithms that use $(\deg+1)$-list coloring as a subroutine [27].

1.2 Technical Overview

At their core, the proofs of Theorems 1.1 and 1.2 are based on three important concepts: *conflict coloring*, *problem amplification* and *0-round solvability.* A *conflict coloring problem* is a list coloring problem where two colors can conflict even if they are not equal. The associated *conflict degree* is the maximum number of conflicts per color a node can have. *Problem amplification* transforms one conflict coloring problem instance into another, as follows: given an input to a problem A, each node computes its input to another problem B (perhaps by exchanging information along the way), with the property that, 1. having a solution to B, a simple one round algorithm computes a solution to A, and 2. the list-size-to-conflict-degree (l/d) ratio of B is larger than that of A. Note that the first property essentially determines the conflicts in B, and usually a color in B is a set of colors in A. The importance of the second property stems from the concept of *0-round solvability*: an instance of a problem B with large enough l/d ratio can be solved in 0 rounds, i.e., with no communication.

From here, the plan is simple: take problem P_0, which is the list coloring problem, recast it as a conflict coloring problem, and amplify it into problems $P_1, \ldots, P_t$, so that P_t is 0-round solvable. Then we can cascade down to a solution of problem P_0, in t rounds. Crucially, in order to do the above, we need P_0 to have sufficiently large l/d ratio to begin with (which explains the particular list size requirements in our theorems). The input m-coloring is used for tie-breaking in the 0-round solution of P_t.

In [23], *local conflict coloring* is the main problem type, where the conflict between two colors depends on who the colors belong to, i.e., two colors can conflict along one edge of the graph and not conflict on another one. Their framework allows solving any local conflict

coloring problem, and by re-modeling a problem with an arbitrary colorspace via mapping each list to an interval $[1, l]$ of natural numbers, one can redefine local conflicts and "forget" about the real size of the colorspace (hence colorspace independence). When computing the input of P_i (given P_{i-1}), in order to maintain manageable conflict degree, *nodes exchange messages* to filter out colors in P_i that cause too much conflict with any neighboring node. These messages are huge (recall that a color in P_i is a set of colors in P_{i-1}). Thus, the input to P_i is usually the topology, P_0-lists and conflicts in the i-hop neighborhood of a node. The goal towards 0-round solvability is then to find a problem P_t whose l/d ratio is larger than the number of all t-hop neighborhood patterns (i.e., inputs). The complicated nature of the input to P_t also makes the 0-round solvability proof rather conceptually involved. The number t of problems required is about $3 \log^*(m + \Delta)$.

Our framework, on the other hand, is based on special *global conflict coloring* instances, where the conflict relation of two colors does not depend on the edge across which they are. This limits us to solving only ordinary list problems P_0. Our key insight (see Section 3.3), which sets Theorems 1.1 and 1.2 apart, is that in our setting *nodes do not need to communicate* for computing the input to problems $P_1, \ldots, P_t$. To achieve this, we show that when forming the lists for P_i from the input to P_{i-1}, it suffices to drop "universally bad" colors (sets of colors in P_{i-1}), whose absence is enough to ensure moderate conflict degree towards any (!) other node. We achieve this by crucially exploiting the symmetry of the particular conflict coloring problems arising from ordinary list coloring.

Thus, the input of a node in P_t is just its input in P_0. This makes the 0-round solution (of P_t) particularly simple. The only communication happens when we cascade down from a solution of P_t to that of P_0. With $t = 2$, we get our main theorem. Since here we have only two problems, the message size is limited (the first round is needed to *learn the P_0-lists* of neighbors, while the second one consists of a *small auxiliary message*). Taking larger t would reduce the requirement on the initial list size but increase message size (see Section 5). Since $t = 2$ is sufficient for our applications, we limit our exposition to that case. Setting $t = 1$ does not give anything non-trivial for list coloring, since the l/d ratio is not large enough, but when all P_0-lists are equal, it gives *an alternative proof of Linial's color reduction* (Section 2.3). In fact, P_1 is essentially the problem of finding a *low intersecting set family*, which Linial's algorithm is based on, while P_2 is a "higher-dimensional" variant of it. Thus, at the core of our result there is an (offline) construction of certain set families over the given color space: given those, the algorithm is easy. This way, we believe our paper also provides a deeper insight into the framework of Thm. 1.1. Our result can also be seen as a bridge between the results of [23] and the recently popular concept of speedup (see Section 5).

1.3 Further Related Work

Most results on distributed graph coloring until roughly 2013 are covered in the excellent monograph by Barenboim and Elkin [12]. An overview of more recent work can be found in [37]. Due to the large volume of published work on distributed graph coloring, we limit this section to an informative treatment of a selected subset. While we have covered most literature on truly local vertex coloring algorithms, there are many known algorithms that trade the high Δ-dependence in the runtime with lower n-dependence. All **deterministic** algorithms in this category (for general input graphs) involve a $\Omega(\log n)$ factor. From the early 90s until very recently, the complexity of $(\deg +1)$-list coloring (and $(\Delta+1)$-coloring) in terms of n was $2^{O(\sqrt{\log n})}$ [1, 45], with algorithms based on *network decomposition* (into small diameter components). A recent breakthrough in network decomposition algorithms [47]

brought the runtime of $(deg+1)$-list coloring down to $\text{poly}\log n$ in LOCAL (it also applies to many other symmetry breaking problems; see [47, 29, 26]). A little later, [7] found a $\text{poly}\log n$ round CONGEST algorithm.

Historically, decompositions into subgraphs that are equipped with low outdegree orientations as used in our results, in [23], and in [8] are closely related to the notion of arboricity. To the best of our knowledge, [9] was the first paper to introduce low out-degree orientations as a tool for distributed graph coloring. First, they showed that one can compute $O(a)$-outdegree orientations in graphs with arboricity a in $O(\log n)$ rounds, and used it to devise several algorithms to color graphs with bounded arboricity. [9] is also the first paper to notice that the degree bound of Δ in Linial's color reduction can be replaced with a bound on the outdegree. Then, [10] devised methods to recursively partition into graphs with small arboricity yielding an $O(\log\Delta\log n)$-round algorithm for $O(\Delta^{1+\varepsilon})$-coloring and an $O(\Delta^{\varepsilon}\log n)$-round algorithm for $O(\Delta)$ coloring. Recently, this recursive technique was extended to $(\deg+1)$-list coloring, giving a $(2^{O(\sqrt{\log\Delta})}\log n)$-round algorithm [37]; the runtime of [37] has a hidden dependence on the color space. While [9, 10, 37] have an inherent $O(\log n)$-factor in their runtime, [8] showed that one can decompose a graph into small arboricity subgraphs (equipped with a small outdegree orientation) without inferring a $O(\log n)$ factor, yielding the first sublinear in Δ algorithm for $\Delta+1$ coloring. In the aftermath, [13] improved the runtime for computing the underlying decompositions (and also simplified the algorithm). Thus, the best forms of our results, [23] and [8] are obtained by using [13] to compute decompositions into subgraphs of small arboricity (equipped with small outdegree orientations).

Note that our results, [23] and [8] only require a bound on the outdegree of the subgraphs' orientations and are oblivious to their arboricity. While bounded outdegree in a graph with a given orientation implies bounded arboricity, computing a bounded outdegree orientation in a graph with bounded arboricity requires $\Omega(\log n)$ rounds, as shown in [9].

Recent **randomized** coloring algorithms rely on the *graph shattering* technique [15]. In the *shattering phase*, a randomized algorithm computes a partial coloring of the graph, after which every uncolored connected component of the graph has small size (say, $\text{poly}\log n$). Then, in the *post-shattering* phase, deterministic $(\deg+1)$-list coloring is applied on all uncolored components in parallel. The runtime of the shattering phase has progressed from $O(\log\Delta)$ [15], over $O(\sqrt{\log\Delta})$ [33] to $O(\log^*\Delta)$ [20]. Combined with the $\text{poly}\log n$-round list coloring algorithm of [47], this gives the current best runtime $\text{poly}\log\log n$, for $(\Delta+1)$-coloring [20], and $O(\log\Delta)+\text{poly}\log\log n$, for $(deg+1)$-list coloring [15].

While special graph classes are out of the scope of this paper, we mention the extensively studied case of distributed **edge coloring**. Here, $\text{poly}\log n$-round algorithms were designed for progressively improving number of colors, from $(2+\varepsilon)\Delta$ [31, 28] to $(2\Delta-1)$ [22, 32], then to $(1+\varepsilon)\Delta$ [30, 32, 48]. The truly local complexity of $(2\Delta-1)$-edge coloring has improved from $O(\Delta)$ [44] to $2^{O(\sqrt{\log\Delta})}$) [37] then to $\text{quasi}\,\text{poly}\log\Delta$ [6] (in addition to $O(\log^* n)$). $O(\Delta^{1+\varepsilon})$-edge colorings can be computed in $O(\log\Delta+\log^* n)$ rounds [11].

Little is known on coloring **lower bounds** (in contrast to other symmetry breaking problems, e.g., maximal matching, MIS or ruling sets [38, 3, 4]). Linial's $\Omega(\log^* n)$ lower bound is extended to randomized algorithms in [43]. The deterministic bound has recently been re-proven in a topological framework [24]. A $\Omega(\Delta^{1/3})$ lower bound for $O(\Delta)$-coloring holds in a weak variant of the LOCAL model [34]. Several works characterized coloring algorithms which can only spend a single communication round [49, 39, 34]. None of these results gives anything non-trivial for two rounds. Also, the *speedup* technique (e.g., [16, 17, 3, 18, 4, 2]), which proved very successful for MIS lower bounds, is poorly understood for graph coloring. We briefly discuss the technique and its relation to our result in Sec. 5. There are lower

bounds for more restricted variants of coloring. There is a $\Omega(\log n)$ ($\Omega(\log\log n)$) lower bound for deterministic [17] (randomized [19]) Δ-coloring, as well as for $(\Delta-2)$-defective 2-coloring [5]. Further, [25] provides a $\Omega(\log n/\log\log n)$ lower bound for greedy coloring. Similar bounds hold for coloring trees and bounded arboricity graphs with significantly fewer than Δ colors [41, 9].

1.4 Roadmap

Section 2.1 introduces our version of conflict coloring together with the 0-round solvability lemma. Section 2.2 defines the problems P_0 and P_1 and provides further notation. Section 2.3 contains the first result of our framework: an alternative proof of Linial's algorithm. Theorem 1.2 (Linial for Lists) is proved in Section 3. Theorem 1.3 ($(deg+1)$-list coloring) is proved in Section 4. We conclude with a discussion of the results and open problems in Section 5.

2 Basic Setup and Linial's Color Reduction

In this section, we first introduce the conflict coloring framework that is the basis of our algorithm, then we show how it quickly implies an alternative variant of Linial's color reduction algorithm. For a set S and an integer $k\geq 0$, let $\mathcal{P}(S)$ and $\binom{S}{k}$ denote the set of all subsets and all size-k subsets of S, respectively. For a map f we use $f^{(i)}$ to denote the i-fold application of f, e.g., $\mathcal{P}^{(2)}(S)=\mathcal{P}(\mathcal{P}(S))$.

2.1 Global Conflict Coloring

A *list family* $\mathcal{F}\subseteq\mathcal{P}(\mathcal{C})$ is a set of subsets of a color space $\mathcal{C}$. Given a *symmetric conflict relation* $\mathcal{R}\subseteq\{\{c,c'\}\mid c,c'\in\mathcal{C}\}$, the *conflict degree* of a family $\mathcal{F}$ in $\mathcal{R}$ is the maximum number of colors in a list L that conflict with a single color in a list L' (possibly same as L), i.e. , $d_{\mathcal{R}}(\mathcal{F})=\max_{L,L'\in\mathcal{F},c\in L}|\{c'\in L'\mid\{c,c'\}\in\mathcal{R}\}|$. An instance $\mathfrak{P}=(\mathcal{C},\mathcal{R},\mathcal{F},\mathcal{L})$ of the *global conflict coloring problem* on the graph G is given[5] by a color space $\mathcal{C}$, a symmetric conflict relation $\mathcal{R}$ on $\mathcal{C}$, a list family $\mathcal{F}$, and an assignment $\mathcal{L}:V\to\mathcal{F}$ of lists $\mathcal{L}(v)\in\mathcal{F}$ of colors to each vertex v. The goal is to assign each vertex a color from its list such that no pair of neighboring vertices get conflicting colors $\{c,c'\}\in\mathcal{R}$. The *conflict degree* of $\mathfrak{P}$ is $d_{\mathcal{R}}(\mathcal{F})$. Note that the conflict degree *does not depend on G or $\mathcal{L}$.*

▶ **Lemma 2.1** (Zero Round Solution). *An instance $(\mathcal{C},\mathcal{R},\mathcal{F},\mathcal{L})$ of the conflict coloring problem on a graph G can be solved without communication if G is m-colored, $m,\mathcal{R},\mathcal{F}$ are globally known, and every list in $\mathcal{F}$ has size at least $l>m\cdot|\mathcal{F}|\cdot d_{\mathcal{R}}(\mathcal{F})$.*

Proof. Every vertex v has a *type* $(\psi_v,\mathcal{L}(v))\in[m]\times\mathcal{F}$, which is uniquely determined by its input color ψ_v and list $\mathcal{L}(v)$. Note that adjacent vertices have distinct types, and there are $t=m\cdot|\mathcal{F}|$ (globally known) possible types. Below, we show how to greedily assign each type a color from its list s.t. different types get non-conflicting colors. The conflict coloring problem is then solved by running this algorithm locally and consistently by all vertices, where each vertex gets the color assigned to its type.

[5] Formally, G is also part of the problem, but we omit it since it is always clear from the context. These definitions crucially differ from <u>local</u> conflict coloring in [23], where a pair of colors can conflict along one edge and not conflict along another.

Let $\{T_i = (m_i, L_i)\}_{i=1}^t$ be a fixed ordering of $[m] \times \mathcal{F}$. Assign T_1 a color $\phi(T_1) \in L_1$ arbitrarily. For any $i \geq 1$, given the colors $\phi(T_1), \ldots, \phi(T_i)$ of preceding types, assign T_{i+1} a color from L_{i+1} that does not conflict with $\phi(T_1), \ldots, \phi(T_i)$. This can be done since each of the i fixed colors conflicts with at most $d_{\mathcal{R}}(\mathcal{F})$ colors in L_{i+1}, i.e., there are at most $i \cdot d_{\mathcal{R}}(\mathcal{F}) \leq m \cdot |\mathcal{F}| \cdot d_{\mathcal{R}}(\mathcal{F})$ colors that T_{i+1} cannot take, and this is less than the size of L_{i+1}, as assumed. ◀

2.2 Basic Problems: P_0 and P_1

Let $\mathcal{C}$ be a fixed and globally known color space (which may depend on the graph G). An *i-list* is a subset $L \subseteq \mathcal{P}^{(i)}(\mathcal{C})$; e.g., the initial color list $L_v \subseteq \mathcal{C}$ of a vertex v is a 0-list. Below, we introduce two problems. Problem P_0 is the standard list coloring problem, which we would like to solve via Lemma 2.1. However, the Lemma may not apply, if the lists L_v are not large enough. We then introduce problem P_1, with parameters $0 < \tau \leq k$, which is a *low intersecting sublist selection* problem. On the one hand, P_1 can be reformulated as a conflict coloring problem with larger lists and color space (hence could be solvable via Lemma 2.1), and on the other hand, a solution to P_1 can be used to solve P_0. The input of a node v in both problems contains its list L_v. Formally, we have, for parameters τ and k,

- **P_0 (list coloring):** Node v has to output a color $c(v) \in L_v$ such that adjacent nodes' colors do not conflict, i.e., they are not equal.
- **P_1 (low intersecting sublists):** Node v has to output a 0-list $C_v \subseteq L_v$ such that $|C_v| = k$ and adjacent nodes' 0-lists do not τ-conflict.
  Two 0-lists $C, C' \subseteq \mathcal{C}$ do *τ-conflict* if $|C \cap C'| \geq \tau$.

Note that problems P_0 and P_1 are not conflict coloring problems in the formal sense defined above (e.g., we do not define a list family $\mathcal{F}$ or a list assignment $\mathcal{L} : V \to \mathcal{F}$). The aim with such definitions is to have a higher level and more intuitive (but still formal) problem statement. As the name suggests, in P_1 each node needs to compute a subset of its list such that the outputs form a low intersecting set family. P_1 can be reduced to a formal conflict coloring problem $\mathfrak{P}_1$ whose solution immediately solves the P_1 instance (see Thm. 2.2).

2.3 Warmup: Linial's Color Reduction (without Lists)

As a demonstration, we use the introduced framework to re-prove Linial's color reduction theorem [41, Thm. 4.1] (which was extended to directed graphs in [9]). An *r-cover-free family* of size k over a set U is a collection of k subsets $C_1, \ldots, C_k \subseteq U$ such that no set C_i is a subset of the union of r others. The obtained algorithm is essentially a *greedy construction* (via Lemma 2.1) of an r-cover-free family (with appropriate parameters) whose existence was proved via the probabilistic method in [41]. This greedy construction was first obtained in [35] but, to our knowledge, remained unnoticed in the distributed computing community.

While our aim is to make the proof below reusable for the later sections (hence the general statement in list coloring terms), we note that similar ideas can be used to obtain a less technical proof of Linial's color reduction (see the full version [42]).

▶ **Theorem 2.2** ([41, 9]). *Let the graph G be m-colored and oriented, with max outdegree β. All nodes have an identical color list $L_v = \mathcal{C}$ from a color space $\mathcal{C}$. Further, $\mathcal{C}$, m, and β are globally known. If $|L_v| = l_0 \geq 2e \cdot \beta^2 \cdot \lceil \log m \rceil$ holds then in 1 round in LOCAL, each node can output a color from its list s.t. adjacent nodes output distinct colors.*

Proof. Our goal is to solve P_0 with the given lists. To this end, it suffices to solve P_1 with parameters $\tau = \lceil \log m \rceil > 1$ and $k = \beta \cdot \tau$, without communication. Indeed, having selected the sublists $(C_v)_{v \in V}$, we need only one round to solve P_0: Node v learns sublists

of its outneighbors and outputs any color $c(v) \in C_v \setminus \cup_{u \in N_{out}(v)} C_u$. This can be done since for any outneighbor u, C_v and C_u do not τ-conflict ($|C_v \cap C_u| \leq \tau - 1$) and hence $|C_v \setminus \cup_{u \in N_{out}(v)} C_u| \geq k - (\tau - 1)\beta > 0$.

We recast the given P_1 instance with (identical) input lists $(L_v)_{v \in V}$ as a conflict coloring problem $\mathfrak{P}_1 = (\mathcal{C}_1, \mathcal{R}_1, \mathcal{F}_1, \mathcal{L}_1)$ with color space $\mathcal{C}_1 = \mathcal{P}(\mathcal{C})$ and the τ-conflict relation as $\mathcal{R}_1$. The list of a node $\mathcal{L}_1(v) = \binom{L_v}{k}$ in $\mathfrak{P}_1$ consists of all k-sized subsets of its input list L_v. As each L_v is identical to $\mathcal{C}$, we have that $\mathcal{L}_1$ maps each node v to the same list $\binom{L_v}{k} = \binom{\mathcal{C}}{k}$ and the list family $\mathcal{F}_1 = \{\binom{\mathcal{C}}{k}\}$ consists of that singleton. A solution to $\mathfrak{P}_1$ immediately solves P_1.

▷ Claim 2.3. The conflict degree of $\mathfrak{P}_1$ is upper bounded by $d_1 = \binom{k}{\tau} \cdot \binom{l_0 - \tau}{k - \tau}$.

Proof. Consider two arbitrary lists $L, L' \in \mathcal{F}_1$ and some 0-list $C \in L$ (that is of size k). Each 0-list $C' \in L'$ that τ-conflicts with C can be constructed by first choosing τ elements of C, and then adding $k - \tau$ elements from the rest of $\mathcal{C}$ which is of size $l_0 - \tau$. This can be done in at most d_1 many ways. ◁

▷ Claim 2.4. Let $l_1 = \binom{l_0}{k}$. For any $k \geq \tau > 1$, if $l_0 \geq 2ek^2/\tau$ then $l_1/d_1 > 2^\tau$.

Proof. We have

$$\frac{l_1}{d_1} = \frac{\binom{l_0}{k}}{\binom{l_0-\tau}{k-\tau}\binom{k}{\tau}} > \left(\frac{l_0}{k}\right)^\tau \cdot \left(\frac{\tau}{ek}\right)^\tau = \left(\frac{l_0 \tau}{ek^2}\right)^\tau \geq 2^\tau \ , \tag{1}$$

where in the first inequality, we used the well-known approximation $\binom{k}{\tau} \leq (ek/\tau)^\tau$, and the following inequality, applied to $\binom{l_0}{k}/\binom{l_0-\tau}{k-\tau}$: for integers $L > K > x > 0$,

$$\frac{\binom{L}{K}}{\binom{L-x}{K-x}} = \frac{L!(K-x)!(L-K)!}{K!(L-K)!(L-x)!} = \frac{L(L-1)\dots(L-x+1)}{K(K-1)\dots(K-x+1)} > \left(\frac{L}{K}\right)^x \ , \tag{2}$$

which follows as $(L-i)/(K-i) > L/K$ holds for $0 < i \leq x$. ◁

Since $\tau \geq \lceil \log m \rceil$ and $|\mathcal{F}_1| = 1$ the last claim implies $|\mathcal{L}_1(v)| = l_1 > md_1 \geq m|\mathcal{F}_1| d_{\mathcal{R}_1}(\mathcal{F}_1)$, hence we can solve $\mathfrak{P}_1$ (and thus also P_1) without communication, using Lemma 2.1. ◀

What we did above is *greedily* forming a Δ-*cover-free family* $C_1, \dots, C_m \subseteq [O(\Delta^2 \log m)]$ of *size* m. The same was done in [41], using the probabilistic method. Having such a family globally known, every vertex of input color x picks, in 0 rounds, the set C_x as its candidate output colors (neighboring vertices get distinct sets). Then, every vertex of color x learns the sets C of its neighbors, and based on the Δ-cover-free property and the fact that there are at most Δ neighbors, can select a color $c \in C_x$ that is not a candidate for any neighbor.

3 Linial for Lists

The goal of this section is to prove the following theorem.

▶ **Theorem 1.2** (Linial for Lists). *In a directed graph with max. degree Δ, max. outdegree β, and an input m-coloring, list coloring with lists L_v from a color space $\mathcal{C}$ and of size $|L_v| \geq l_0 = 4e\beta^2(4 \log \beta + \log \log |\mathcal{C}| + \log \log m + 8)$ can be solved in 2 rounds in LOCAL. Each node sends $l_0 + 1$ colors in the first round, and a l_0/β^2-bit message in the second.*

Assumption. Throughout this section, we assume that *the list of each node is exactly of size l_0*; if a node's list is larger it can select an arbitrary subset of size l_0.

3.1 The Problem P_2 (Low Intersecting Sublist Systems)

Recall that when applying our framework to the case where all lists were equal (Thm. 2.2), we essentially constructed a Δ-cover-free family over the color space, and this was sufficient because we only needed a family of size m: one set for each possible input pair (x, L) of a color x and list L, with *the same L for all nodes.* In order to replicate the construction for list coloring, we would need to construct a cover-free family with a set $C_{x,L}$ for every combination of color x and list L, and such that $C_{x,L} \subseteq L$. It is not hard to see that such a family does not exist. Instead, we introduce problem P_2, whose goal is to assign every input (x, L) a *collection of candidate subsets* $K_{x,L} = \{C_{x,L,1}, C_{x,L,2}, \ldots\}$, where each $C_{x,L,i} \subseteq L$. Further, we need that for every pair of distinct collections, there are not many pairs of subsets from the two collections that intersect much (in a sense formally defined below). This ensures that having such $K_{x,L}$, the nodes can compute the desired Δ-cover free family with one communication round, and use it to choose a color in another round.

Problem P_2 depends on parameters $0 < \tau \leq k \leq l_0$ and $0 < \tau' \leq k'$, and each node has a list $L_v \subseteq \mathcal{C}$ in its input. Instead of a color (as in P_0) or a sublist (as in P_1), each vertex v now needs to output a collection $K_v = \{C_1, C_2, \ldots\}$ of sublists of L_v, each of size k.

P_2 (low intersecting sublist systems): Node v has to output a 1-list $K_v \subseteq \mathcal{P}(L_v)$ s.t. adjacent nodes' 1-lists do not (τ', τ)-conflict and $|K_v| = k'$ and $|C| = k$ for all $C \in K_v$. Two 1-lists $K, K' \subseteq \mathcal{P}(\mathcal{C})$ do (τ', τ)-*conflict* if there are two sequences $C_1, \ldots, C_{\tau'} \in K$ and $C'_1, \ldots, C'_{\tau'} \in K'$, where at least one of the sequences has τ' distinct elements and for every $1 \leq i \leq \tau'$, C_i and C'_i τ-conflict.

We prove in Lemma 3.1 that with a suitable choice of the parameters a solution of $P_2(\tau, k, \tau', k')$ yields a solution of $P_1(\tau, k)$ and P_0, where we implicitly impose (and throughout this section assume) that P_0, P_1 and P_2 receive the same input lists $(L_v)_{v \in V}$.

3.2 Algorithm

Under the assumptions of Thm. 1.2, fix the following (globally known) four parameters: $\tau = \lceil 8 \log \beta + 2 \log \log |\mathcal{C}| + 2 \log \log m \rceil + 14$, $\tau' = 2^{\tau - \lceil \log(2e\beta^2) \rceil}$, $k = \beta \cdot \tau$ and $k' = \beta \cdot \tau'$. Note that, τ', k, k' are determined by τ and β. [6] We have the bound $l_0 \geq 2ek^2/\tau$ on list size.

The algorithm consists of two phases. In the first phase, nodes **locally** and without any communication compute a solution $(K_v)_{v \in V}$ of P_2 consisting of 1-lists (see Lemma 3.2). The second phase has two rounds of communication. In the **first round**, each node v learns the solution K_u to P_2 of each outneighbor $u \in N_{out}(v)$, and selects a 0-list $C_v \in K_v$ that does not conflict with the 0-lists in K_u, for $u \in N_{out}(v)$, and thus is a solution to P_1 (Lemma 3.1). In the **second round**, node v learns the lists C_u of outneighbors, and selects a color $c(v) \in C_v$ that does not appear in C_u, for $u \in N_{out}(v)$ (Lemma 3.1). This solves P_0.

▶ **Lemma 3.1** ($P_2 \to P_1 \to P_0$). *Given a solution $(K_v)_{v \in V}$ of P_2 (a solution $(C_v)_{v \in V}$ of P_1), a solution of P_1 (of P_0, resp.) can be computed in one round.*

Proof. **$P_2 \to P_1$:** As K_v and K_u do not (τ', τ)-conflict for any $u \in N_{out}(v)$, there are at most $\tau' - 1$ 0-lists $C \in K_v$ that τ-conflict with a 0-list in K_u. By removing all C from K_v that τ-conflict with any $C' \in K_u$ for any outneighbor $u \in N_{out}(v)$ at least $|K_v| - \beta \cdot (\tau' - 1) = k' - \beta \cdot (\tau' - 1) \geq 1$ outputs remain; let C_v be any such 0-list. As the conflict relation is symmetric, P_1 is solved.

[6] It may also be helpful to note the similarity between this parameter setting and that in Thm. 2.2.

***$P_1 \to P_0$*:** Since C_v, C_u do not τ-conflict, removing from C_v all the colors from the 0-lists of the outneighbors leaves at least $k - \beta \cdot (\tau - 1) \geq 1$ colors that v can select as $c(v)$. ◀

3.3 Zero Round Solution to P_2

The results in this section hold for parameters τ, τ', k' fixed as in Section 3.2, and for any $\tau \leq k \leq \beta\tau$. While we set $k = \beta\tau$ for solving P_0, we will use another value of k for our defective coloring result (see [42]). Note that we still have the bound $l_0 \geq 2ek^2/\tau$ on list size, for any such k. The goal of this section is to prove the following lemma.

▶ **Lemma 3.2** (P_2 in zero rounds). *Under the assumptions of Thm. 1.2, the problem $P_2(\tau, k, \tau', k')$ can be solved in zero rounds.*

To prove Lemma 3.2, we reduce (without communication) an instance of P_2 to a conflict coloring instance $\mathfrak{P}_2$ that can be solved in zero rounds with Lemma 2.1.

Reducing P_2 to a conflict coloring instance $\mathfrak{P}_2$ (without communication): Given input lists $(L_v)_{v \in V}$ and parameters $0 < \tau \leq k \leq l_0$ and $0 < \tau' \leq k$, the conflict coloring instance $\mathfrak{P}_2$ is given by the colorspace $\mathcal{P}^{(2)}(\mathcal{C})$, the (τ', τ)-conflict relation $\mathcal{R}_2$ on 1-lists, the list family $\mathcal{F}_2 = \mathrm{Im}(L_2) = \{L_2(S) \mid S \in \binom{\mathcal{C}}{l_0}\}$ and list $\mathcal{L}(v) = L_2(L_v)$ for node v, where $L_2 : \binom{\mathcal{C}}{l_0} \to \mathcal{P}^{(3)}(\mathcal{C})$ maps l_0-sized subsets of $\mathcal{C}$ to 2-lists and is defined below. The map L_2, the colorspace, the conflict relation and the set family $\mathcal{F}_2$ are global knowledge and no communication is needed to compute the list $\mathcal{L}(v)$ of a node in $\mathfrak{P}_2$.

To define the map L_2 we need another definition. For an integer $t \geq 0$ and a 2-list T, a 1-list $K \in T$ is (T, t, τ', τ)-*good* if there are less than t 1-lists $K' \in T$ such that K and K' do (τ', τ)-conflict. We define maps L_1, $\bar{L}_2$ and L_2, as follows. For $S \in \binom{\mathcal{C}}{l_0}$,

$$L_1(S) = \binom{S}{k} \qquad \text{(elements } C \text{ are 0-lists)}$$

$$\bar{L}_2(S) = \binom{L_1(S)}{k'} \qquad \text{(elements } K \text{ are 1-lists)}$$

$$L_2(S) = \{K \in \bar{L}_2(S) \mid K \text{ is } (\bar{L}_2(S), d_2, \tau', \tau)\text{-good}\} \qquad \text{(elements } K \text{ are 1-lists),}$$

where d_2 is chosen as in Lemma 3.4.[7] Due to the definition of the (τ', τ)-conflict relation and the map L_2, solving $\mathfrak{P}_2$ immediately solves P_2.

The sizes of $L_1(S), \bar{L}_2(S)$ and $L_2(S)$ do not depend on S. Let $l_1 = |L_1(S)| = \binom{l_0}{k}$, and $l_2 = |\bar{L}_2(S)|/2 = \binom{l_1}{k'}/2$. We will later show that $|L_2(S)| \geq l_2$. Let $\mathcal{F}_1 = \{L_1(S) \mid S \in \binom{\mathcal{C}}{l_0}\}$.

Some intuition: In the conflict coloring instance $\mathfrak{P}_2$, every node v has a list $\{K_1, K_2, \ldots\}$ of 1-lists, each a collection of subsets of its input list L_v. To ensure small conflict degree, but still large list size, it is enough that v only takes Ks that are "good", as defined above. Since being "good" only depends on L_v, node v can also compute its $\mathfrak{P}_2$-list locally.

In Lemmas 3.3 to 3.5, we show that lists $L_2(L_v)$ are large and that $\mathfrak{P}_2$ has small conflict degree. Before that, let us see how these lemmas imply 0-round solvability of P_2 (Lemma 3.2).

Proof of Lemma 3.2. To solve an instance of P_2 on input lists $(L_v)_{v \in V}$, nodes locally set up the conflict coloring instance $\mathfrak{P}_2$. Lemmas 3.3 and 3.4 show that the conflict degree of $\mathfrak{P}_2$ is bounded by $d_{\mathcal{R}_2}(\mathcal{F}_2) \leq d_2$, and that every list in $\mathcal{F}_2$ has size at least l_2. Note that $\mathcal{F}_2$ is globally known and $|\mathcal{F}_2| = \binom{|\mathcal{C}|}{l_0} < |\mathcal{C}|^{l_0}$, since each element in $\mathcal{F}$ can be written as $L_2(S)$

[7] The precise value is not important to understand how L_2 is formed.

for some $S \in \binom{\mathcal{C}}{l_0}$. Using Lemma 3.5 we obtain $l_2/d_2 \geq \frac{1}{8}2^{2^{\tau-\log(4e\beta^2)}} \geq m \cdot |\mathcal{C}|^{l_0} > m \cdot |\mathcal{F}_2|$, where the second inequality follows by a routine calculation using the definition of τ and l_0 (see [42]). Thus, Lemma 2.1 holds, and $\mathfrak{P}_2$ and P_2 can be solved in zero rounds. ◀

We continue with proving Lemmas 3.3 to 3.5. First, we bound the conflict degree of $\mathfrak{P}_2$. Recall that it is a property of the list family $\mathcal{F}_2$ and the conflict relation $\mathcal{R}_2$, and is independent of the graph and list assignment. The proof involves establishing an isomorphism between $L_2(S)$ and $L_2(S')$, for any S, S', which preserves their common elements. For this, it is crucial to have $|S| = |S'|$. This is why we need all input lists to have same size $|L_v| = l_0$.

▶ **Lemma 3.3** (Conflict Degrees). *Let $X, Y \in \binom{\mathcal{C}}{l_0}$ be 0-lists. Let $d_1 = \binom{k}{\tau} \cdot \binom{l_0-\tau}{k-\tau}$.*

1. *For any 0-list $C \in L_1(X)$, there are at most d_1 0-lists in $L_1(Y)$ that τ-conflict with C.*
2. *For any 1-list $K \in L_2(X)$, there are at most d_2 1-lists in $L_2(Y)$ that (τ', τ)-conflict with K. In particular, $d_{\mathcal{R}_2}(\mathcal{F}_2) \leq d_2$, and this holds irrespective of the value of d_2.*

Proof. The proof of the first claim is along the same lines as the proof of Claim 2.3, so we only prove the second claim here. Let $X_1 = L_1(X), X_2 = L_2(X)$ and $\bar{X}_2 = \bar{L}_2(X)$, and define $Y_1, Y_2, \bar{Y}_2$ similarly. As $|X| = |Y|$, there is a bijection $\alpha : X \to Y$ that is the identity on $X \cap Y$: if $c \in X \cap Y$ then $\alpha(c) = c$. Further, since $X_1 = \binom{X}{k}$ and $Y_1 = \binom{Y}{k}$, we have the bijection $\beta : X_1 \to Y_1$ given by $\beta(\{c_1, \ldots, c_k\}) = \{\alpha(c_1), \ldots, \alpha(c_k)\}$, and since $\bar{X}_2 = \binom{X_1}{k'}$ and $\bar{Y}_2 = \binom{Y_1}{k'}$, we have the bijection $\gamma : \bar{X}_2 \to \bar{Y}_2$, where $\gamma(\{C_1, \ldots, C_{k'}\}) = \{\beta(C_1), \ldots, \beta(C_{k'})\}$.

We show that the claim holds for any $t \geq 0$ and for any $K \in \bar{X}_2$ that is $(\bar{X}_2, t, \tau', \tau)$-good (which demonstrates that the actual value of d_2 is irrelevant). As $Y_2 \subseteq \bar{Y}_2$, it suffices to show that K does (τ', τ)-conflict with at most t 1-lists in $\bar{Y}_2$. Towards a contradiction, let $K \in \bar{X}_2$ (τ', τ)-conflict with each of t distinct 1-lists $K'_1, K'_2, \ldots, K'_t \in \bar{Y}_2$ and define $K_i = \gamma^{-1}(K'_i) \in \bar{X}_2$. We show that K also (τ', τ)-conflicts with each of the distinct (γ is a bijection) $K_1, \ldots, K_t \in \bar{X}_2$, which is a contradiction to K being $(\bar{X}_2, t, \tau', \tau)$-good: To ease notation, let us focus on K and K_1. Assume there are τ' distinct (case 2: not necessarily distinct) 0-lists $C'_1, C'_2, \ldots, C'_{\tau'}$ in K'_1, and τ' not necessarily distinct (case 2: distinct) 0-lists $C_1, C_2, \ldots, C_{\tau'}$ in K, such that C_i and C'_i τ-conflict. Then $\beta^{-1}(C'_i)$ and C_i τ-conflict, since α is the identity on $C_i \cap C'_i$, $\beta^{-1}(C'_i)$ are all distinct (since β is a bijection) and belong to K_1, therefore K and K_1 (τ', τ)-conflict. ◀

Next, we show that at most half of the elements $K \in \bar{L}_2$ fail to be good; this lemma crucially depends on the value of d_2. Below, we use the conflict degree d_1 from Lemma 3.3.

▶ **Lemma 3.4** (L_2 is large). *Let $d_2 = 4\binom{k'd_1}{\tau'} \cdot \binom{l_1-\tau'}{k'-\tau'}$. For any $S \in \binom{\mathcal{C}}{l_0}$, we have $|L_2(S)| \geq l_2$.*

Proof. Fix $S \in \binom{\mathcal{C}}{l_0}$ and consider the digraph $H = (V_H, E_H)$ over the vertex set $V_H = \bar{L}_2(S)$, where $(K, K') \in E_H$ iff K contains at least τ' lists, each in τ-conflict with a list in K' (in particular, for every K, $(K, K) \in E_H$). Note that a 1-list K is $(\bar{L}_2(S), d_2, \tau', \tau)$-good iff its undirected degree in H is at most d_2.

▷ Claim. The maximum outdegree of a node $K \in V_H$ is at most $d_2/4$.

Proof. Consider a fixed $K \in V_H$. Let $X \subseteq K$ be the set of 0-lists in $L_1(S)$ that τ-conflict with a 0-list in K. By Lemma 3.3 part 1, every $C \in K$ τ-conflicts with at most d_1 of 0-lists, hence $|X| \leq |K| \cdot d_1 = kd_1$. Every 1-list K', such that there are at least τ' 0-lists in K that are in τ-conflict with a 0-list in K', can be obtained by first choosing τ' 0-lists from X, and adding an arbitrary subset of $k' - \tau'$ other 0-lists. Clearly, this can be done in at most $\binom{k'd_1}{\tau'} \cdot \binom{l_1-\tau'}{k'-\tau'} = d_2/4$ many ways. ◁

The Claim implies that $|E_H| \leq |V_H| \cdot d_2/4$, hence the undirected average degree of a node in H is at most $2|E_H|/|V_H| \leq d_2/2$, and by Markov's inequality, at most half of the nodes have degree greater than d_2. Since $L_2(S)$ is the set of nodes of degree at most d_2, we conclude that $|L_2(S)| \geq |V_H|/2 = |\bar{L}_2(S)|/2 = l_2$. ◀

Finally, we bound the ratio l_2/d_2 based on the values of the remaining parameters.

▶ **Lemma 3.5** (l/d Ratio). *If $k \geq \tau \geq \lceil\log(2e\beta^2)\rceil$, $l_0 \geq 2ek^2/\tau$, $\tau' = 2^{\tau-\lceil\log(2e\beta^2)\rceil}$, and $k' = \beta\tau'$, then $l_2/d_2 > 2^{2^{\tau-\log(4e\beta^2)}}/8$.*

Proof. First, we get $l_1/d_1 \geq 2^\tau$, as in Eq. (1). Then, with $\binom{k'd_1}{\tau'} \leq (\frac{ekd_1}{\tau'})^{\tau'}$, and (2) applied to $\binom{l_1}{k'}/\binom{l_1-\tau'}{k'-\tau'}$, we lower bound l_2/d_2 as

$$\frac{l_2}{d_2} = \frac{1}{8}\frac{\binom{l_1}{k'}}{\binom{l_1-\tau'}{k'-\tau'}\binom{k'd_1}{\tau'}} > \frac{1}{8}\left(\frac{l_1}{k'}\cdot\frac{\tau'}{e(k'd_1)}\right)^{\tau'} \geq \frac{1}{8}\left(\frac{2^\tau}{e\beta^2\tau'}\right)^{\tau'} \geq \frac{2^{\tau'}}{8} \geq \frac{2^{2^{\tau-\log(4e\beta^2)}}}{8},$$

where the third and fourth inequalities hold since $\tau' \leq \frac{2^\tau}{2e\beta^2} \leq 2\tau'$. ◀

3.4 Proof of the Main Theorem

Proof of Thm. 1.2. Nodes solve P_2 in zero rounds (Lemma 3.2), and then use two rounds of communication to solve the input list coloring problem P_0 (see algorithm description and Lemma 3.1). We bound the messages sent by a node v during the algorithm. In the first round, v needs to send K_v to its neighbors. Note that K_v is uniquely determined by the list L_v and the input color ψ_v (see the proof of Lemma 2.1) , so it suffices to send (ψ_v, L_v), which can be encoded in $l_0\lceil\log|\mathcal{C}|\rceil + \lceil\log m\rceil$ bits. In the second round, v needs to send C_v. Since $C_v \in K_v$, and the neighbors know K_v, it suffices to send the index of C_v in K_v (in a fixed ordering). Recall that $|K_v| = k' < 2^\tau$, so v only needs to send $\tau \leq l/4e\beta^2$ bits. ◀

▶ Remark 3.6. Note that in both communication rounds of Theorem 1.2 each node only needs to send messages to its in-neighbors. In contrast, the results in Section 4 and the defective coloring results require bi-directional communication.

4 Application: $(\Delta + 1)$-Coloring and $(deg + 1)$-List Coloring

▶ **Theorem 1.3** ($(deg+1)$-List Coloring). *In a graph with max. degree Δ, $(deg+1)$-list coloring with lists $L_v \subseteq \mathcal{C}$ from a color space of size $|\mathcal{C}| = 2^{\text{poly}(\Delta)}$ [8] can be solved in $O(\sqrt{\Delta\log\Delta}) + \frac{1}{2}\cdot\log^* n$ rounds in LOCAL. Furthermore, each node only needs to broadcast to its neighbors a single non-CONGEST message consisting of a subset of its list.*

The proof combines Thm. 1.2 with the graph partitioning provided by [13], following the high level description in Sec. 1. A variant of this framework was also used in [37, 6]. We nevertheless present a proof for completeness, and also due to subtle but important differences from [23] (we have an additional finishing phase that is not present there).

The graph partitioning given by [13] aims at *arbdefective colorings*, as introduced in [10], but the main technical object provided by [13] (and which is all we need here) is a *low outdegree partition* of a graph. For a graph $H = (V, E)$, the collection $H_1, \ldots, H_k$ of *directed* graphs $H_i = (V_i, E_i)$ is a *β-outdegree partition* of H if their vertices span V, i.e.,

[8] We use the notation $\text{poly}(X) = O(X^c)$, for an absolute constant c, and $\tilde{O}(X) = X \cdot \text{poly}(\log X)$.

$V = V_1 \cup \cdots \cup V_k$, the underlying undirected graph of H_i is the induced subgraph $H[V_i]$ (so it is indeed a partition), and the max. outdegree of a node in H_i is at most β, for all i.

▶ **Lemma 4.1** (Lemmas 6.1-6.3, [13]). *There are constants $c, c' > 0$, s.t. for every $\beta \geq c$, given a graph H with an m-coloring, there is a deterministic algorithm that computes a β-outdegree partition $H_1, \ldots, H_k$ with $k = c'\Delta/\beta$ in $O(k + \log^* m)$ rounds in CONGEST.*

Proof of Thm. 1.3. We begin with computing an $m = O(\Delta^2)$-coloring in $\frac{1}{2}\log^* n + O(1)$ rounds [41, 49]. The main algorithm consists of $t = \log_2(\Delta/\Delta^{1/4})$ phases. After phase j, we have colored a subset of vertices, s.t. the maximum degree Δ_j of the graph $G[U_j]$ induced by uncolored vertices is upper bounded as $\Delta_j \leq \Delta/2^j$. Before describing a phase j, let us show how we finish the coloring after the phase t, in a **final phase**. Consider the graph $G[U_t]$ at the end of phase t. Note that it has maximum degree $\Delta_t = O(\Delta^{1/4})$. We compute an m'-coloring of $G[U_t]$ with $m' = O(\Delta_t^2) = O(\sqrt{\Delta})$ from the initial m-coloring in $O(1)$ rounds [41]. In each of the final m' rounds $i = 1, \ldots, m'$, vertices with color i pick a color from their list not picked by a neighbor (can be done since $|L_v| > \Delta$ and no two neighbors pick simultaneously). The runtime of the final phase is $O(\sqrt{\Delta})$.

The following happens in **phase** $j = 1, \ldots, t$. At the beginning of the phase, we have the set U_{j-1} of uncolored vertices, where $U_0 = V(G)$. Let $X = 4e \cdot (4\log\Delta + \log\log|\mathcal{C}| + \log\log m + 8) = O(\log\Delta)$ and for $j = 0, \ldots, t-1$ let $\beta_j = \sqrt{\Delta_j/(2X)}$ and $k_j = c' \cdot \Delta_j/\beta_j$, where c' is the constant in Lemma 4.1.[9] We partition $G[U_{j-1}]$ into β_j-outdegree subgraphs $H_1, H_2, \ldots, H_{k_j}$, using Lemma 4.1. The phase consists of k_j stages $i = 1, \ldots, k_j$, each consisting of 3 rounds. In **stage** i, we partially color H_i, as follows. For every uncolored vertex $v \in H_i$, let $L_{v,j,i}$ be the set of colors in L_v that have not been taken by a neighbor of v. Let $W_i = \{v \in H_i : |L_{v,j,i}| \geq \beta_j^2 X\}$. Color the graph $H_i[W_i]$ using Linial for Lists (Thm. 1.2) with color space $\mathcal{C}$, the β_j-outdegree orientation and the m-coloring. This is a valid application of the theorem, by the definition of X, β_j and W_i. In the third round of the stage, all nodes in W_i send their color to their neighbors. This completes the algorithm description. Clearly, phase j takes $3k_j$ rounds.

It remains to show that $\Delta_j \leq \Delta/2^j$. We do this by induction, with base $j = 0$, $\Delta_0 = \Delta$. Assume $\Delta_j \leq \Delta/2^j$ holds for some $j \geq 0$. Let $v \in U_j$ be a node that is uncolored at the end of phase j. We know that $|L_{v,j,i}| < \beta_j^2 X = \Delta_j/2$, in a stage i. Recall that $L_{v,j,i}$ is the set of colors in L_v not taken by a neighbor of v. Since $|L_v|$ is larger than the number of neighbors of v, $|L_{v,j,i}|$ is larger than the number of *uncolored* neighbors of v. Therefore v has at most $|L_{v,j,i}| < \Delta_j/2 \leq \Delta/2^{j+1}$ neighbors in U_j, which proves the induction: $\Delta_{j+1} \leq \Delta/2^{j+1}$.

Recall that $X = O(\log\Delta)$ and bound the runtime as follows:

$$\frac{1}{2}\log^* n + O(1) + \sum_{j=1}^{t} 3k_j + m' = \frac{1}{2}\log^* n + \sum_{j=1}^{t} 3c'\sqrt{\frac{X\Delta}{2^{j-1}}} + O(\sqrt{\Delta}) = \frac{1}{2}\log^* n + O(\sqrt{\Delta\log\Delta})\,.$$

The second claim easily follows, recalling the message complexity of Linial for Lists. ◀

Note that the final phase in the algorithm above is necessary as otherwise, if the recursion continued until the maximum degree of uncolored nodes was, say, $O(\log^{(3)}\Delta)$, their reduced list size would be similarly small, and we could no longer apply Theorem 1.2, which requires lists of size $\Omega(\log\log|\mathcal{C}|) = \Omega(\log\log\Delta)$, as the color space does not change in the recursion.

The (simple) proof of the following corollary is deferred to the full version [42].

---

[9] In order to apply the lemma, we need $\beta_j \geq c$. Since $\beta_j \geq \beta_t = \Omega(\sqrt{\sqrt{\Delta}/\log\Delta})$, $\beta_j \geq c$ holds if Δ is large enough. For $\Delta = O(1)$, Thm. 1.3 holds via a $O(\Delta) + 1/2\log^* n$ round algorithm (see e.g. [13]).

▶ **Corollary 4.2.** *In a graph with max. degree* $\Delta = \tilde{O}(\log n)$*,* $(deg + 1)$*-list coloring with lists* $L_v \subseteq \mathcal{C}$ *from a color space of size* $|\mathcal{C}| = \text{poly}(\Delta)$ *can be solved in* $\tilde{O}(\sqrt{\Delta}) + \frac{1}{2} \cdot \log^* n$ *rounds in CONGEST.*

5 Discussion

We conclude with several observations on our results, as well as open problems.

1. It is possible to define problems $P_3, \ldots, P_t$ for any t, as we defined P_1 and P_2. Lemma 3.3 extends naturally to these problems, so the input of a node v in P_i is again only its initial list L_v. We need t rounds, instead of 2, to derive a solution of P_0 from a solution of P_t (which also implies larger messages). On the other hand, we have somewhat smaller list size requirement: $c\beta^2(\log\beta + \log^{(t)}|\mathcal{C}| + \log^{(t)} m)$, for a constant $c > 0$. In particular, one can list color in $O(\log^* \max\{|\mathcal{C}|, m\})$ rounds if lists are at least $c\beta^2 \log\beta$ for a sufficiently large constant $c > 0$.
2. Unlike in [23], our bound on the list size does not depend on Δ. This result implies that, e.g., given a graph with a β-outdegree orientation and an input coloring with $2^{\text{poly}(\beta)}$ colors and list sizes of at least $c\beta^2 \cdot \log\beta$ from a color space of size $2^{\text{poly}(\beta)}$, for a constant $c > 0$, it is possible to list-color the graph in 2 rounds. By the remark above, one can have even larger color space, by increasing the runtime accordingly.
3. A lower bound in [49] suggests that the coloring in Theorem 1.2 cannot be done in a single round. In particular, if one is willing to keep the doubly-logarithmic dependence on m in the list size, then one has to pay a factor exponential in β. On the other hand, we do not know how to eliminate the $\log\beta$ term, even if we use more communication.
4. The recently popular *speedup* technique has mostly been used to prove lower bounds, e.g., [16, 17, 3, 18, 4, 2]. Here, a problem P_0 is mechanically (and without communication!) transformed into a problem P_1 whose complexity is exactly one round less. Then, if P_1 cannot be solved locally one deduces that P_0 cannot be solved in 1 round. By iterating this process, one can derive larger lower bounds. However, the description complexity of derived problems grows exponentially, and it is very important to be able to simplify the problem description, in order to iterate the process. If P_0 is the $(\Delta + 1)$-vertex coloring problem, this process has only been understood in the special case of $\Delta = 2$, which corresponds to Linial's $\Omega(\log^* n)$ lower bound [41, 40]. While [23] also performs a similar transformation, it is different from the speedup technique, since the transformation is not mechanical, requiring nodes to communicate for building the new problems. It may rather be seen as a transformation of *problem instances* (that depend on the graph) than problems. In contrast, our transformations are mechanical, and the input and output labels live in the same universe as it is the case for mechanical speedup.
5. While our present treatment of the proof of Thm. 1.2 in terms of conflict coloring problems and problems P_0, P_1 and P_2 has the aim of connecting to the framework of [23] as well as to the speedup framework, we note that the proof can be stated entirely in terms of set systems, just like the proof of Linial's color reduction.

Open Problem: Remove the $\log\beta$ term in Thm. 1.2 while keeping the runtime $o(\sqrt{\log\beta})$. This question is particularly of interest because $\log\beta$ is the source of the $\sqrt{\log\Delta}$ factor in in Thm. 1.3 (note that the terms depending on $m, |\mathcal{C}|$ can be reduced, by the remarks above). The non-list $O(\Delta^2)$-coloring by Linial uses, in addition to his main color reduction, a $O(\Delta^3)$-to-$O(\Delta^2)$ color reduction, using polynomials over finite fields [41]. With a more sophisticated use of polynomials [8] constructs a cover-free family for list coloring but it requires a much smaller outdegree. It is not clear if polynomials help with our question.

- **Open Problem:** More generally, prove or rule out a truly local $(\Delta+1)$-coloring algorithm with Δ-dependence $f(\Delta) = o(\sqrt{\Delta})$.

References

1. Baruch Awerbuch, Andrew V. Goldberg, Michael Luby, and Serge A. Plotkin. Network decomposition and locality in distributed computation. In *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*, pages 364–369, 1989.
2. Alkida Balliu, Sebastian Brandt, Yuval Efron, Juho Hirvonen, Yannic Maus, Dennis Olivetti, and Jukka Suomela. Classification of distributed binary labeling problems. In *Proceedings of the International Symposium on Distributed Computing (DISC)*, 2020.
3. Alkida Balliu, Sebastian Brandt, Juho Hirvonen, Dennis Olivetti, Mikaël Rabie, and Jukka Suomela. Lower bounds for maximal matchings and maximal independent sets. In *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*, pages 481–497, 2019.
4. Alkida Balliu, Sebastian Brandt, and Dennis Olivetti. Distributed lower bounds for ruling sets. In *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*, 2020.
5. Alkida Balliu, Juho Hirvonen, Christoph Lenzen, Dennis Olivetti, and Jukka Suomela. Locality of not-so-weak coloring. In *Proceedings of the International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, pages 37–51, 2019.
6. Alkida Balliu, Fabian Kuhn, and Dennis Olivetti. Distributed edge coloring in time quasi-polylogarithmic in delta. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, 2020.
7. Philipp Bamberger, Fabian Kuhn, and Yannic Maus. Efficient deterministic distributed coloring with small bandwidth. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, 2020.
8. Leonid Barenboim. Deterministic $(\Delta + 1)$-coloring in sublinear (in Δ) time in static, dynamic, and faulty networks. *Journal of the ACM*, 63(5):47:1–47:22, 2016.
9. Leonid Barenboim and Michael Elkin. Sublogarithmic distributed MIS algorithm for sparse graphs using Nash-Williams decomposition. *Distributed Comput.*, 22(5-6):363–379, 2010.
10. Leonid Barenboim and Michael Elkin. Deterministic distributed vertex coloring in polylogarithmic time. *Journal of the ACM*, 58(5):23:1–23:25, 2011.
11. Leonid Barenboim and Michael Elkin. Distributed deterministic edge coloring using bounded neighborhood independence. In *Proceedings of the 30th Annual ACM Symposium on Principles of Distributed Computing, PODC 2011, San Jose, CA, USA, June 6-8, 2011*, pages 129–138, 2011. doi:10.1145/1993806.1993825.
12. Leonid Barenboim and Michael Elkin. *Distributed Graph Coloring: Fundamentals and Recent Developments*. Morgan & Claypool Publishers, 2013.
13. Leonid Barenboim, Michael Elkin, and Uri Goldenberg. Locally-Iterative Distributed $(\Delta+1)$-Coloring below Szegedy-Vishwanathan Barrier, and Applications to Self-Stabilization and to Restricted-Bandwidth Models. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 437–446, 2018.
14. Leonid Barenboim, Michael Elkin, and Fabian Kuhn. Distributed (Delta+1)-Coloring in Linear (in Delta) Time. *SIAM J. Comput.*, 43(1):72–95, 2014.
15. Leonid Barenboim, Michael Elkin, Seth Pettie, and Johannes Schneider. The locality of distributed symmetry breaking. *Journal of the ACM*, 63(3):20:1–20:45, 2016.
16. Sebastian Brandt. An automatic speedup theorem for distributed problems. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 379–388, 2019.
17. 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 *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 479–488, 2016.

18 Sebastian Brandt and Dennis Olivetti. Truly tight-in-δ bounds for bipartite maximal matching and variants. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, 2020.

19 Yi-Jun Chang, Tsvi Kopelowitz, and Seth Pettie. An exponential separation between randomized and deterministic complexity in the LOCAL model. *SIAM J. Comput.*, 48(1):122–143, 2019.

20 Yi-Jun Chang, Wenzheng Li, and Seth Pettie. An optimal distributed (Δ+1)-coloring algorithm? In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 445–456, 2018.

21 Paul Erdös, Peter Frankl, and Zóltan Füredi. Families of finite sets in which no set is covered by the union of r others. *Israel Journal of Mathematics*, 51:79–89, 1985.

22 Manuela Fischer, Mohsen Ghaffari, and Fabian Kuhn. Deterministic distributed edge-coloring via hypergraph maximal matching. In *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*, pages 180–191, 2017.

23 Pierre Fraigniaud, Marc Heinrich, and Adrian Kosowski. Local conflict coloring. In *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*, pages 625–634, 2016.

24 Pierre Fraigniaud and Ami Paz. The topology of local computing in networks. In *Proceedings of the International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 128:1–128:18, 2020. doi:10.4230/LIPIcs.ICALP.2020.128.

25 Cyril Gavoille, Ralf Klasing, Adrian Kosowski, Lukasz Kuszner, and Alfredo Navarra. On the complexity of distributed graph coloring with local minimality constraints. *Networks*, 54(1):12–19, 2009.

26 Mohsen Ghaffari, David G. Harris, and Fabian Kuhn. On derandomizing local distributed algorithms. In *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*, pages 662–673, 2018.

27 Mohsen Ghaffari, Juho Hirvonen, Fabian Kuhn, and Yannic Maus. Improved distributed delta-coloring. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 427–436, 2018.

28 Mohsen Ghaffari, Juho Hirvonen, Fabian Kuhn, Yannic Maus, Jukka Suomela, and Jara Uitto. Improved distributed degree splitting and edge coloring. In *Proceedings of the International Symposium on Distributed Computing (DISC)*, pages 19:1–19:15, 2017.

29 Mohsen Ghaffari, Fabian Kuhn, and Yannic Maus. On the complexity of local distributed graph problems. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 784–797, 2017.

30 Mohsen Ghaffari, Fabian Kuhn, Yannic Maus, and Jara Uitto. Deterministic distributed edge-coloring with fewer colors. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 418–430, 2018.

31 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.

32 David G. Harris. Distributed local approximation algorithms for maximum matching in graphs and hypergraphs. In *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*, pages 700–724, 2019.

33 David G. Harris, Johannes Schneider, and Hsin-Hao Su. Distributed (Δ +1)-coloring in sublogarithmic rounds. *Journal of the ACM*, 65(4):19:1–19:21, 2018.

34 Dan Hefetz, Fabian Kuhn, Yannic Maus, and Angelika Steger. Polynomial lower bound for distributed graph coloring in a weak LOCAL model. In *Proceedings of the International Symposium on Distributed Computing (DISC)*, pages 99–113, 2016.

35 Frank K. Hwang and Vera T. Sós. Non-adaptive hypergeometric group testing. *Studia scient. Math. Hungaria*, 22:257–263, 1987.

36 Fabian Kuhn. Weak graph colorings: distributed algorithms and applications. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architecture (SPAA)*, pages 138–144, 2009.

37 Fabian Kuhn. Faster deterministic distributed coloring through recursive list coloring. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1244–1259, 2020.

38 Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. Local computation: Lower and upper bounds. *Journal of the ACM*, 63(2), 2016.

39 Fabian Kuhn and Roger Wattenhofer. On the complexity of distributed graph coloring. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 7–15, 2006.

40 Juhana Laurinharju and Jukka Suomela. Brief announcement: Linial's lower bound made easy. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 377–378, 2014.

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

42 Yannic Maus and Tigran Tonoyan. Local conflict coloring revisited: Linial for lists, 2020. `arXiv:2007.15251`.

43 Moni Naor. A lower bound on probabilistic algorithms for distributive ring coloring. *SIAM J. Discret. Math.*, 4(3):409–412, 1991.

44 Alessandro Panconesi and Romeo Rizzi. Some simple distributed algorithms for sparse networks. *Distributed Comput.*, 14(2):97–100, 2001.

45 Alessandro Panconesi and Aravind Srinivasan. Improved distributed algorithms for coloring and network decomposition problems. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 581–592, 1992.

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

47 Václav Rozhon and Mohsen Ghaffari. Polylogarithmic-time deterministic network decomposition and distributed derandomization. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 350–363, 2020.

48 Hsin-Hao Su and Hoa T. Vu. Towards the locality of vizing's theorem. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 355–364, 2019.

49 Mario Szegedy and Sundar Vishwanathan. Locality based graph coloring. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 201–207, 1993.

Classification of Distributed Binary Labeling Problems

Alkida Balliu
Freiburg University, Germany
alkida.balliu@cs.uni-freiburg.de

Sebastian Brandt
ETH Zürich, Switzerland
brandts@ethz.ch

Yuval Efron
Technion – Israel Institute of Technology, Haifa, Israel
efronyuv@gmail.com

Juho Hirvonen
Aalto University, Finland
juho.hirvonen@aalto.fi

Yannic Maus
Technion – Israel Institute of Technology, Haifa, Israel
yannic.maus@cs.technion.ac.il

Dennis Olivetti
Freiburg University, Germany
dennis.olivetti@cs.uni-freiburg.de

Jukka Suomela
Aalto University, Finland
jukka.suomela@aalto.fi

Abstract

We present a complete classification of the deterministic distributed time complexity for a family of graph problems: *binary labeling problems* in trees. These are locally checkable problems that can be encoded with an alphabet of size two in the edge labeling formalism. Examples of binary labeling problems include sinkless orientation, sinkless and sourceless orientation, 2-vertex coloring, perfect matching, and the task of coloring edges red and blue such that all nodes are incident to at least one red and at least one blue edge. More generally, we can encode e.g. any cardinality constraints on indegrees and outdegrees.

We study the deterministic time complexity of solving a given binary labeling problem in trees, in the usual LOCAL model of distributed computing. We show that the complexity of any such problem is in one of the following classes: $O(1)$, $\Theta(\log n)$, $\Theta(n)$, or unsolvable. In particular, a problem that can be represented in the binary labeling formalism cannot have time complexity $\Theta(\log^* n)$, and hence we know that e.g. any encoding of maximal matchings has to use at least three labels (which is tight).

Furthermore, given the description of any binary labeling problem, we can easily determine in which of the four classes it is and what is an asymptotically optimal algorithm for solving it. Hence the distributed time complexity of binary labeling problems is *decidable*, not only in principle, but also *in practice*: there is a simple and efficient algorithm that takes the description of a binary labeling problem and outputs its distributed time complexity.

2012 ACM Subject Classification Theory of computation → Distributed computing models; Theory of computation → Complexity classes

Keywords and phrases LOCAL model, graph problems, locally checkable labeling problems, distributed computational complexity

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.17

Related Version The full version of this work is available at `https://arxiv.org/abs/1911.13294`.

Funding This work was supported in part by the Academy of Finland, Grant 314888 (Juho Hirvonen), and by the European Union's Horizon 2020 Research And Innovation Programme under grant agreement no. 755839 (Yuval Efron, Yannic Maus).

Acknowledgements We thank Jan Studený and anonymous reviewers for helpful comments on earlier versions of this work.

© Alkida Balliu, Sebastian Brandt, Yuval Efron, Juho Hirvonen, Yannic Maus, Dennis Olivetti, and Jukka Suomela;
licensed under Creative Commons License CC-BY
34th International Symposium on Distributed Computing (DISC 2020).
Editor: Hagit Attiya; Article No. 17; pp. 17:1–17:17
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

This work presents a complete classification of the deterministic distributed time complexity for a family of distributed graph problems: *binary labeling problems* in trees. These are a special case of widely-studied locally checkable labeling problems [27]. The defining property of a binary labeling problem is that it can be encoded with an *alphabet of size two* in the *edge labeling formalism*, which is a modern representation for locally checkable graph problems [3, 9, 28]; we will give the precise definition in Section 2.

Contributions. In this work, we focus on *deterministic* distributed algorithms in the LOCAL model of distributed computing [24,29], and we study the computational complexity of solving a binary labeling problem in *trees*. It is easy to see that there are binary labeling problems that fall in each of the following classes:

- Trivial problems, solvable in $O(1)$ rounds.
- Problems similar to sinkless orientation, solvable in $\Theta(\log n)$ rounds [10, 15, 22].
- Global problems, requiring $\Theta(n)$ rounds.
- Unsolvable problems.

We show that this is a *complete* list of all possible complexities. In particular, there are no binary labeling problems of complexities such as $\Theta(\log^* n)$ or $\Theta(\sqrt{n})$. For example, maximal matching is a problem very similar in spirit to binary labeling problems, it has a complexity $\Theta(\log^* n)$ in bounded-degree graphs [17, 24], and it can be encoded in the edge labeling formalism using an alphabet of size three [3] – our work shows that three labels are also necessary for all problems in this complexity class.

Moreover, using our results one can easily determine the complexity class of any given binary labeling problem. We give a simple, concise characterization of all binary labeling problems for classes $O(1)$, $\Theta(n)$, and unsolvable, and we show that all other problems belong to class $\Theta(\log n)$. Hence the deterministic distributed time complexity of a binary labeling problem is *decidable*, not only in theory but also *in practice*: given the description of any binary labeling problem, a human being or a computer can easily find out the distributed computational complexity of the problem, as well as an asymptotically optimal algorithm for solving the problem. Our classification of all binary labeling problems is presented in Table 1, and given any binary labeling problem Π, one can simply do mechanical pattern matching to find its complexity class in this table.

Our work also sheds new light on the *automatic round elimination technique* [9, 28]. Previously, it was known that sinkless orientation is a nontrivial *fixed point* for round elimination [10] – such fixed points are very helpful for lower bound proofs, but little was known about the existence of other nontrivial fixed points. Our classification of binary labeling problems in this work led to the discovery of new nontrivial fixed points – this will hopefully pave the way for the development of a theoretical framework that enables us to understand when round elimination leads to fixed points and why.

This work will also make it *easier to prove lower bounds in the future.* Before this work, in essence the only known way to prove a nontrivial $\Omega(\log n)$ lower bound for some locally checkable problem Π was to come up with a reduction showing that Π is at least as hard as sinkless orientation. Our systematic exploration of binary labeling problems led to the discovery of an entire family of simple graph problems that are not directly comparable with sinkless orientation, but for which we can prove tight $\Omega(\log n)$ lower bounds directly with round elimination. In the future, whenever we encounter a graph problem Π of an unknown complexity, we can try to relate it not only to sinkless orientation but also to one of the new problems for which we now have new lower bounds.

Open questions. The main open question that we leave for future work is extending the characterization to randomized distributed algorithms: some binary labeling problems can be solved in $\Theta(\log \log n)$ rounds with randomized algorithms, but it is not yet known exactly which binary labeling problems belong to this class. Our work takes the first steps towards developing such a classification.

Structure. We start with the model of computation in Section 1.1 and a brief discussion of the general landscape of distributed computational complexity in Section 1.2, and then give the formal definitions of binary labeling problems in Section 2 that are needed to present our results in a concise manner, which we do in Section 3 – our main contribution is the characterization of all binary labeling problems in Table 1. Section 4 discusses the expressive power of binary labeling problems and collects examples of interesting binary labeling problems. In Section 5, we explain our proof techniques, and, more importantly, the key ideas of the technically most involved results. We conclude with Section 6, in which we discuss additional connections between the present work and related work – in particular, we highlight the role of binary labeling problems in the recent developments in the distributed computational complexity theory – and we will also introduce new directions for future research. All algorithms and lower bound proofs related to deterministic complexity and a discussion of randomized complexity are presented in the full version [2].

1.1 Model of computing

Deterministic LOCAL model. In this work, we use the standard LOCAL model of distributed computing [24, 29]. In this model, the input graph G represents a communication network, where each node is a computer and each edge is a communication link. If there are n nodes in the graph, each node is labeled with a *unique identifier* from $\{1, 2, \ldots, \text{poly}(n)\}$. This is part of the local input, and a distributed algorithm can use it when it initializes the local state of a node. Computation proceeds in synchronous rounds, and in each round each node:

- sends a message to each neighbor,
- receives a message from each neighbor,
- performs local computation and updates its local state.

After each round, a node can choose to stop and produce its *local output*.

Let Π be a graph problem. We say that $\mathcal{A}$ is a deterministic algorithm that solves problem Π in $T(n)$ rounds if, for any input graph G with n nodes and for any assignment of unique identifiers, each node stops after at most $T(n)$ rounds and the local outputs of the nodes form a feasible solution of Π in G; if the task is to label edges we require that both endpoints agree on a label for the edge. We say that Π has complexity T in the LOCAL model if T is the pointwise minimum of all functions T' such that there exists an algorithm $\mathcal{A}$ that solves Π in time T'.

Note that in this model time is equivalent to distance: in T synchronous communication rounds all nodes can gather their radius-T neighborhoods (and nothing more). Hence we can interchangeably refer to *locality*, *distributed time complexity*, and the *number of communication rounds*.

Randomized LOCAL model. Our main focus is on deterministic distributed algorithms, but we will also discuss randomized distributed algorithms. In the randomized LOCAL model, in addition to having unique identifiers, nodes are labeled with an unbounded stream of random bits. Hence, we can let state transitions be probabilistic, and we arrive at randomized algorithms. In this work, randomized algorithms are Monte Carlo algorithms that are correct w.h.p. in the size of the graph.

1.2 Background and related work

Distributed complexity theory and LCL problems. The study of distributed graph algorithms has traditionally focused on specific graph problems – for example, investigating exactly what is the locality of finding a maximal independent set [3, 4]. However, in the recent years we have seen more focus on the development of a distributed complexity theory with which we can reason about entire *families of graph problems* [1, 5–7, 10, 11, 15, 16, 18, 19, 21, 23, 30, 31].

The key example is the family of *locally checkable labeling* problems (LCLs), introduced by [27]. Informally, a problem is locally checkable if the feasibility of a solution can be verified by looking at all constant-radius neighborhoods. For example, maximal independent sets are locally checkable, as we can verify both independence and maximality by looking at radius-1 neighborhoods.

In this line of research, among the most intriguing results are various *gap theorems*: For example, there are LCL problems solvable in $\Theta(\log^* n)$ rounds, while some LCL problems require $\Theta(\log n)$ rounds. However, between these two classes there is a gap: there are no LCL problems whose deterministic complexity in bounded-degree graphs is between $\omega(\log^* n)$ and $o(\log n)$ [15].

Decidability of distributed computational complexity. The existence of such a gap immediately suggests a follow-up question: given the description of an LCL problem, can we *decide* on which side of the gap it lies? And if so, can we automatically construct an asymptotically optimal algorithm for solving the problem?

As soon as we look at a family of graphs that contains e.g. 2-dimensional grids, questions related to the distributed complexity of a given LCL problem become undecidable [11, 27]. However, if we look at the case of paths and cycles, we can at least in principle write a computer program that determines the computational complexity of a given LCL problem [1, 11, 27], and some questions related to the complexity of LCLs in trees are also decidable [16] – unfortunately, we run into PSPACE-hardness already in the case of paths and cycles [1].

We conjecture that *all* questions about the distributed complexity of LCL problems in trees are decidable. Proving (or disproving) the conjecture is a major research program, but in this work we take one step towards proving the conjecture and we bring plenty of good news: we introduce a family of LCL problems, so-called *binary labeling problems*, and we show that we can completely characterize the deterministic distributed complexity of every binary labeling problem in trees. In particular, all questions about the deterministic distributed complexity of these problems are decidable not only in principle but also in practice – using our results, a human being or a computer can easily find an optimal algorithm for solving any given binary labeling problem.

2 Binary labeling problems

We will now give the formal definition of the family of binary labeling problems. LCL problems have been traditionally specified by listing a *collection of permitted local neighborhoods* [27]. However, we will use the more recent *edge labeling* formalism [3, 9, 28], which is equally expressive in the case of trees, and it has the additional benefit that it makes it very convenient to apply the automatic round elimination technique [9, 28], which is very helpful to prove lower bounds:

- we have a bipartite graph and the nodes are colored with two colors, white and black,
- the task is to *label edges* with symbols from some alphabet Σ,
- there are *both white and black constraints* that define the graph problem.

We emphasize that despite its bipartite appearance, the edge labeling formalism can easily be used to describe problems on general graphs by interpreting edges as "black nodes". In fact, as explained in detail in Section 4, problems on general graphs are a special case in the edge labeling formalism – in general the formalism describes problems on hypergraphs.

2.1 General form

If we set $|\Sigma| = 2$ in the edge labeling formalism, we arrive at the definition of binary labeling problems. Formally, a *binary labeling problem* is a tuple $\Pi = (d, \delta, W, B)$, where

- $d \in \{2, 3, \dots\}$ is the *white degree*,
- $\delta \in \{2, 3, \dots\}$ is the *black degree*,
- $W \subseteq \{0, 1, \dots, d\}$ is the *white constraint*, and
- $B \subseteq \{0, 1, \dots, \delta\}$ is the *black constraint*.

An *instance* of problem Π is a pair (G, f), where

- $G = (V, E)$ is a simple graph,
- $f : V \to \{\text{black}, \text{white}\}$ is a proper 2-coloring of G, i.e., $f(u) \neq f(v)$ for all $\{u, v\} \in E$.

In the distributed setting, we will assume that the color $f(v)$ is part of the local input of node $v \in V$.

Let $X \subseteq E$ be a subset of edges. For each node $v \in V$, its X-degree $\deg_X(v)$ is the number of edges in X that are incident to v. We say that $X \subseteq E$ is a *solution* to binary labeling problem Π if it satisfies the following constraints for all $v \in V$:

- If $f(v) = \text{white}$ and $\deg(v) = d$, then $\deg_X(v) \in W$.
- If $f(v) = \text{black}$ and $\deg(v) = \delta$, then $\deg_X(v) \in B$.

We use the term *relevant nodes* to refer to white nodes of degree d and black nodes of degree δ. The interpretation is that problem Π is interesting in regular neighborhoods in which all white nodes have degree d and all black nodes have degree δ; any irregularities make the problem easier to solve, as all other nodes are unconstrained.

▶ **Example 2.1** (bipartite splitting)**.** Let $d = \delta = 4$ and $W = B = \{1, 2, 3\}$. We can interpret a solution $X \subseteq E$ as a coloring: edges in X are colored red and all other edges are colored blue. Now Π is equivalent to the following graph problem on bipartite graphs: color all edges red or blue such that all degree-4 nodes are incident to at least one blue edge and at least one red edge.

In order to be able to state our results in a concise manner, we will introduce some necessary notation and terminology in the following.

2.2 Equivalence, restrictions, and relaxations

If a is a natural number and A is a set of natural numbers, we will define $a - A = \{a - x : x \in A\}$.

▶ **Definition 2.2** (Equivalence)**.** *For any d, δ, W, B we call the following four problems* equivalent

$$\Pi_{00} = (d, \delta, W, B), \qquad \Pi_{01} = (\delta, d, B, W),$$
$$\Pi_{10} = (d, \delta, d - W, \delta - B), \qquad \Pi_{11} = (\delta, d, \delta - B, d - W).$$

Definition 2.2 partitions binary labeling problems in equivalence classes, each of them with at most four distinct problems. We use notation $\Pi \sim \Pi'$ for this equivalence relation, and say that Π and Π' are *equivalent*.

▶ **Observation 2.3.** *If $\Pi \sim \Pi'$ then Π and Π' have the same distributed complexity up to ± 1 round.*

Proof. Given an algorithm for Π, we get an algorithm for Π' by either exchanging the roles of black and white nodes or by replacing solution X with its complement $E \setminus X$. As incident nodes need to agree on the output on edges the complexity can differ by 1 round. ◀

The following definition is helpful to show that our classification is complete.

▶ **Definition 2.4.** *Given two problems $\Pi = (d, \delta, W, B)$ and $\Pi' = (d, \delta, W', B')$, we say that Π' is a* restriction *of Π and Π is a* relaxation *of Π' if $W' \subseteq W$, and $B' \subseteq B$.*

We use notation $\Pi' \subseteq \Pi$ to denote that Π' is a restriction of Π.

▶ **Observation 2.5.** *If $\Pi' \subseteq \Pi$, then any feasible solution for Π' is also a feasible solution for Π. In particular, if Π' can be solved in T rounds, then Π can also be solved in T rounds.*

2.3 Vector notation

It is convenient to interpret set W as a bit vector $w_0 w_1 \dots w_d$ with $d + 1$ bits, so that bit $w_i = 1$ if $i \in W$ and $w_i = 0$ if $i \notin W$. Similarly, B can be interpreted as a bit vector with $\delta + 1$ bits.

▶ **Example 2.6.** Using this notation, the problem of Example 2.1 can be represented with $W = 01110$ and $B = 01110$, or, in brief, $\Pi = (01110, 01110)$.

Note that when we use vector notation, vectors W and B fully determine problem Π; therefore we do not need to specify d and δ separately and we can simply write $\Pi = (W, B)$.

We will use shorthand notation such as 1^x for a vector of x 1s and 1^+ for a vector of one or more 1s, and we will use $*$ to refer to a bit of any value. For example, $W = 0{*}{*}0$ is a shorthand for $W \in \{0000, 0010, 0100, 0110\}$ and $W = 01^+0$ is a shorthand for $W \in \{010, 0110, 01110, \dots\}$.

3 Our contributions

3.1 Main results: deterministic complexity

Our main contribution is a complete classification of the deterministic distributed complexity of binary labeling problems in trees – this is presented in Table 1, and our results hold in the standard LOCAL model of distributed computing [24, 29].

Next, we discuss the classification in more detail: One can partition all binary labeling problems in 15 *families of problems* based on the structure of their white and black constraints such that, given a binary labeling problem Π in the vector notation, one can simply do pattern matching in Table 1 to first find its problem family and this way also find its deterministic complexity. For easier reference, we have listed all problem families explicitly, but if we combine families whose problems are equivalent (recall Observation 2.3) we can partition problems in seven *types*, labeled with **I**, **II**, ..., **VII** in the table. Types also reflect the structure of our proofs. The mapping from types (and families) to distributed complexity is also shown in Table 1. It then directly follows that the distributed complexity of any given binary labeling problem is decidable, and efficiently computable.

Table 1 The deterministic distributed complexity of binary labeling problems in trees; 1^+ signifies one or more 1s, 0^+ signifies one or more 0s, and $*$ signifies either 0 or 1. Non-empty means that the constraint is not 0^+. Note that e.g. all problem families of type I are equivalent to each other in the following sense: for any problem Π of type I, there is exactly one problem equivalent to Π in each of the four families I.a, I.b, I.c, and I.d.

Type	Problem family	White constraint	Black constraint	**Deterministic complexity**
I	I.a	100^+	$0{*}{*}^+$	**unsolvable**
	I.b	00^+1	$*{*}^+0$	
	I.c	$0{*}{*}^+$	100^+	
	I.d	$*{*}^+0$	00^+1	
II	II.a	000^+	$*{*}{*}^+$	
	II.b	$*{*}{*}^+$	000^+	
III	III.a	non-empty	111^+	$\boldsymbol{O(1)}$
	III.b	111^+	non-empty	
IV	IV.a	$1{*}{*}^+$	$1{*}{*}^+$	
	IV.b	$*{*}^+1$	$*{*}^+1$	
V	V.a	10^+1	010	$\boldsymbol{\Theta(n)}$
	V.b	010	10^+1	
VI	VI.a	0^+1*	$*10^+$	
	VI.b	$*10^+$	0^+1*	
VII	VII.a	all other cases		$\boldsymbol{\Theta(\log n)}$

Table 2 Examples of binary labeling problems and problem families and their deterministic complexities.

Deterministic complexity	Type	Examples of problems: W	B	reference	description
unsolvable	I	011^+	100	Example 4.8	contradiction
$O(1)$	III	0010^+	111	Example 4.9	trivial
$\Theta(n)$	V	10^+1	010	Example 4.5	two-coloring
$\Theta(\log n)$	VII	01110	01110	Example 2.1	bipartite splitting
		1010	010	Example 4.4	even orientation
		111^+0	010	Example 4.1	sinkless orientation
		011^+0	010	Example 4.3	sinkless & sourceless orientation
		0100^+	101	Example 4.6	regular matching
		011^+0	101	Example 4.7	splitting
		0100^+	0100^+	Section 5	bipartite matching
		0100^+	10^+1	Section 5	hypergraph matching
		0100^+	110	Section 5	edge grabbing
$\Omega(\log n)$, $O(n)$	V or VII	1^+01^+	01^+0	Section 5	forbidden degree
	VI or VII	11^+0	011^+	Section 5	bipartite sinkless orientation

We next detail on the different types. Problems of types **I** and **II** are unsolvable, problems of type **III** and **IV** have complexity $O(1)$, and problems of types **V** an **VI** have complexity $\Theta(n)$. Now by definition all problems that are not of type **I**–**VI** are of type **VII**; we prove that all of them are solvable in $O(\log n)$ rounds and we also prove a matching lower bound of $\Omega(\log n)$. This completes the proof that the classification of Table 1 is correct.

Finding problems that are unsolvable or need $O(1)$ rounds is not difficult:

- There are two types of problems that are unsolvable for trivial reasons. For example, if all white nodes must have X-degree 0 and all black nodes must have a non-zero X-degree, no solution exists as long as the tree is large enough.
- There are two types of problems that are solvable in $O(1)$ time for trivial reasons. For example, if black nodes are happy with any X-degree, then white nodes can simply pick some number $x \in W$ and choose arbitrarily x adjacent edges.

However, what is not obvious is that this list is exhaustive: no other problems are unsolvable, and no other problem is solvable in $O(1)$ time.

Furthermore, as we can see in Table 1, there are only very few binary labeling problems that are solvable but inherently global, i.e., they require $\Theta(n)$ rounds to solve. What is perhaps the biggest surprise is that all other problems can be solved in $O(\log n)$ rounds, and they also require $\Omega(\log n)$ rounds. In particular:

▶ **Theorem 3.1.** *There are no binary labeling problems with deterministic distributed complexity in the following ranges:*

- *between* $\omega(1)$ *and* $o(\log n)$*,*
- *between* $\omega(\log n)$ *and* $o(n)$*.*

Table 2 shows examples of problems in each complexity class.

3.2 Additional results: randomized complexity

While our focus in this work is on deterministic complexity, we will also explore the randomized complexity of binary labeling problems. Many of our theorems from the deterministic parts have direct implications on randomized complexity. By prior work, it is known that classes $O(1)$ and $\Theta(n)$ remain the same also for randomized complexity – this follows from [19] and (unpublished) extensions of the result in [5]. However, class $\Theta(\log n)$ is more interesting, as there are binary labeling problems of the following types:

- Deterministic complexity $\Theta(\log n)$ and randomized complexity $\Theta(\log \log n)$, e.g. sinkless orientation, Example 4.1.
- Deterministic complexity $\Theta(\log n)$ and randomized complexity $\Theta(\log n)$, e.g. even orientation, Example 4.4.

That is, randomness helps with some binary labeling problems but not all. While we present a partial classification of the randomized complexity of binary labeling problems, the main open question for the future work is coming up with a complete characterization of exactly which binary labeling problems can be solved in $O(\log \log n)$ rounds with randomized algorithms.

4 Expressive power of binary labeling problems

At first the bipartite setting in the definition of binary labeling problems in Section 2.1 may seem restrictive – indeed, why would we care about graphs that are properly 2-colored. However, we can take any graph and interpret edges as “black nodes” and this way many graph problems of interest can be represented in the binary labeling formalism. More precisely, let $G_0 = (V_0, E_0)$ be a graph. Subdivide (i.e., add a vertex in the middle of each edge) all edges of G_0 to construct a new graph $G = (V, E)$, and assign the color white to all original nodes in V_0 and color black to the new nodes in $V \setminus V_0$.

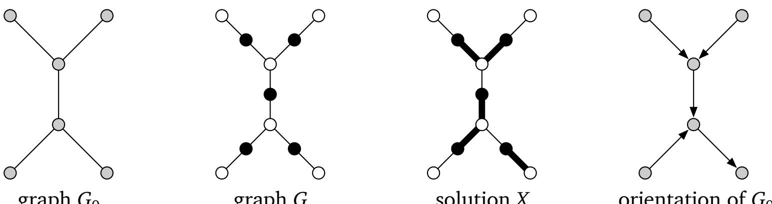


Figure 1 Binary labeling problems with $B = 010$ are orientation problems.

To simulate an algorithm $\mathcal{A}$ for the (virtual) network G in the communication network G_0 each vertex of G has to be simulated by a vertex of G_0 and we use node identifiers (or randomness) to choose which endpoint simulates the black node of an edge. Finally, for the purposes of lower bounds we can also go in the other direction and take an algorithm for G_0 and simulate it in graph G. This increases the round complexity by a factor of two, which we can ignore as we are only interested in asymptotics.

Consider a binary labeling problem Π with $B = 010$. Assume that we have a solution X to Π in G. We can now interpret X as an *orientation* of the original graph G_0: If an edge $e = \{u, v\} \in E_0$ was subdivided in two edges, $e_1 = \{u, x\}$, and $e_2 = \{x, v\}$, we will have exactly one of these edges in set X. If we have $e_1 \in X$, we can interpret it so that edge $\{u, v\}$ is oriented from v to u, and otherwise it is oriented from u to v; see Figure 1. In essence, Π is now equivalent to the following problem: Find an orientation of G_0 such that all nodes with $\deg(v) = d$ have $\text{indegree}(v) \in W$.

▶ **Example 4.1** (sinkless orientation)**.** Let $d > 2$, $\delta = 2$, $W = 111^+0$, and $B = 010$. Now all edges must be properly oriented: there is exactly one head. Furthermore, for all nodes of degree d, they must have indegree in $\{0, 1, \ldots, d-1\}$. Put otherwise, degree-d nodes must have outdegree at least one. Hence a feasible solution represents an orientation in which none of degree-d nodes are sinks.

▶ **Example 4.2.** Let $W = 0111^+$ and $B = 010$. By Observation 2.3, this is equivalent to Example 4.1. In essence, we have merely reversed the roles of heads and tails – now the task is to find a sourceless orientation.

▶ **Example 4.3** (sinkless and sourceless orientation)**.** Let $W = 011^+0$ and $B = 010$. Now in d-regular graphs the task is to find an orientation such that all nodes have at least one incoming and at least one outgoing edge.

Note that this problem is a restriction of Example 4.1; recall Definition 2.4.

▶ **Example 4.4** (even orientation)**.** Let $W = 1010$ and $B = 010$. In 3-regular graphs the task is to find an orientation such that all nodes have an even number of outgoing edges.

▶ **Example 4.5** (two-coloring)**.** Let $W = 10^+1$ and $B = 010$. In d-regular graphs this is a 2-coloring problem: each node v is either “red” (indegree 0) or “blue” (indegree d), and all edges between such nodes are properly colored (they are always oriented from red to blue).

Another interesting special case is $B = 101$. Now for each original edge of G_0 we will select either both of the half-edges or none of the half-edges, and hence a solution $X \subseteq E$ can be interpreted in a natural way as a *subset of edges* $X_0 \subseteq E_0$ in the original graph; see Figure 2. This is a splitting problem: partition E_0 in two classes, X_0 and $E_0 \setminus X_0$, and W determines how many incident edges in each class we can have.

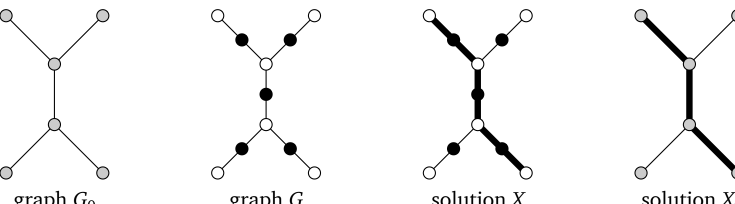


Figure 2 Binary labeling problems with $B = 101$ are splitting problems.

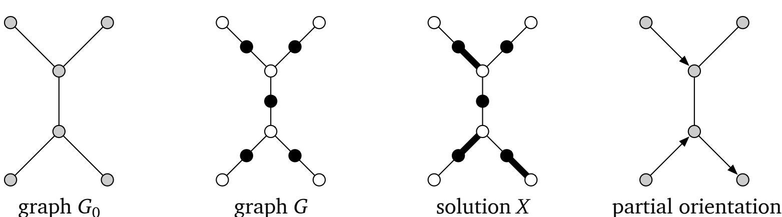


Figure 3 Binary labeling problems with $B = 110$ are partial orientation problems.

▶ **Example 4.6** (regular matching)**.** Let $W = 0100^+$ and $B = 101$. The task is to find a set $X_0 \subseteq E_0$ such that all nodes of degree d are incident to exactly one edge in X_0. In particular, if we have a d-regular graph, X_0 is a perfect matching.

▶ **Example 4.7** (splitting)**.** Let $W = 011^+0$ and $B = 101$. In this problem we will need to color edges red and blue such that all degree-d nodes are incident to at least one red edge and at least one blue edge.

We can also consider e.g. $B = 110$ and interpret X as a *partial orientation*: some edges of G_0 are oriented and some may be left unoriented and again W indicates which indegrees are permitted; see Figure 3. The case of $B = 011$ is equivalent to $B = 110$ – recall Observation 2.3.

As we will see, all other cases $B = 000$, $B = 100$, $B = 001$, and $B = 111$ are either trivial or unsolvable; we will give two examples:

▶ **Example 4.8** (contradiction)**.** Let $W = 011^+$ and $B = 100$. Here black nodes must have X-degree 0, so we must have $X = \emptyset$. However, all white nodes must be adjacent to at least one edge in X. Hence there is no solution to the problem.

▶ **Example 4.9** (trivial)**.** Let $W = 0010^+$ and $B = 111$. Here all white nodes can arbitrarily choose two incident edges and add them to X.

▶ Remark 4.10. Similar ideas can be generalized to hypergraphs. In essence, we can interpret the bipartite graph G as a hypergraph H, where white nodes of G correspond to nodes of H and black nodes of G correspond to hyperedges of H. Now Π is in essence a hypergraph problem. If we set $B = 010^+$, a solution X can be interpreted as an orientation of hyperedges, and if we set $B = 10^+1$, a solution X can be interpreted as a subset of hyperedges. Sinkless orientations in hypergraphs have been studied, e.g., in [12].

4.1 Binary labeling problems in trees

All of the above examples are well-defined also in trees; however, some care is needed when we interpret them. For example, let us revisit the "regular matching" problem from Example 4.6. Let $W = 0100^+$ and $B = 101$. Now in the case of a tree, the task is to find a subset of edges

X such that internal nodes of degree d are incident to exactly one such edge. However, leaf nodes are unconstrained. Informally, if we are in the middle of a d-regular tree, we will need to find a solution that locally looks like a perfect matching, but near leaf nodes and other irregularities the output is more relaxed. One consequence is that for this problem a solution always exists (while there are of course trees in which a perfect matching does not exist).

5 Overview of the key technical ideas

Due to space constraints, the full proofs are deferred to the full version [2]. Here, we give an overview of the key techniques and ideas where we focus on the technically most involved parts. In these parts we show that all problems that cannot be (easily) put into the categories *unsolvable*, $O(1)$ round complexity or $\Theta(n)$ round complexity have complexity $\Theta(\log n)$.

Problems of types I and II are unsolvable. We show that problems of types I and II are problems in which a contradiction occurs, e.g., black nodes can only label their incident edges with 0, but white nodes must have at least one incident edge labeled 1. Such problems clearly cannot be solved.

Problems of types III and IV can be solved in $O(1)$ rounds. We show that problems of types III and IV can be solved without any communication. These problem classes consist of problems in which all nodes can output the same label on all their incident edges, and problems in which black (white) nodes are happy with any labeling.

Problems of types V and VI require $\Theta(n)$ rounds. We show that problems of types V and VI consist of variants of the 2-coloring problem and variants of the problem of consistently orienting a given tree. We prove that all such problems require $\Omega(n)$ rounds, using indistinguishability arguments. In essence, we prove that some nodes need to coordinate their outputs over a linear distance.

> **Key idea: all remaining cases have complexity $\Theta(\log n)$.** So far we have covered relatively simple cases. Surprisingly, we can show that for all problems not of types I–VI there are deterministic $O(\log n)$ upper bounds and deterministic $\Omega(\log n)$ lower bounds.

All other problems can be solved in $O(\log n)$ rounds. To show the upper bound, we define the following notion of *resilience*. Let $\Pi = (d, \delta, W, B)$ be a binary labeling problem. For $0 \le t \le d + 1$ and $0 \le s \le \delta + 1$ we say that Π is (t, s)-*resilient* if both of the following hold:

- bit string W does not contain a substring of the form 0^{d+1-t},
- bit string B does not contain a substring of the form $0^{\delta+1-s}$.

We also introduce three special problem families:

- Problems of the form $\Pi = (0100^+, 0100^+)$ are called *bipartite matching problems*.
- Problems of the form $\Pi = (0100^+, 10^+1)$ are called *hypergraph matching problems*.
- Problems of the form $\Pi = (0100^+, 110)$ are called *edge grabbing problems*.

We show that any problem that is *not* of types I–VI has to fall in one of the following four classes:

(1) $(2, 1)$-resilient problems and $(1, 2)$-resilient problems.

(2) Bipartite matching and equivalent problems.

(3) Hypergraph matching and equivalent problems.
(4) Edge grabbing and equivalent problems.

We show how to use the rake & compress technique [25] to design $O(\log n)$-round algorithms for all of these problems. In brief, we decompose the vertex set of the tree into $L = O(\log n)$ layers, and sequentially construct a valid solution in $O(\log n)$ iterations. We first fix the solutions of all nodes in layer L in parallel, then all nodes in layer $L-1$ in parallel, and so on. Here resilience is crucial, as it guarantees lower layers can still satisfy their requirements even if the higher layers have made arbitrary choices.

All other problems require $\Omega(\log n)$ rounds. To show the matching lower bound, we define two more problem families:

- Problems of the form $\Pi = (1^+01^+, 01^+0)$ are called *forbidden degree problems*.
- Problems of the form $\Pi = (11^+0, 011^+)$ are called *bipartite sinkless orientation problems*.

We then show that all problem that are not unsolvable (types **I** and **II**) or trivial (types **III** and **IV**) are at least as hard as forbidden degree problems or bipartite sinkless orientation problems. Finally, we show that both of these problems require $\Omega(\log n)$ rounds.

The lower bounds for bipartite sinkless orientation problems essentially follow from [10]; however, for forbidden degree problems no lower bounds were known previously.

We show the lower bound using the round elimination technique [9]. We define a new parameterized problem family that we call $\mathsf{FDSO}(s)$ – we emphasize that this is *not* a binary labeling problem, but it turns out that an efficient algorithm for the forbidden degree problem implies an efficient algorithm for $\mathsf{FDSO}(s)$. In the language of [3, 28], $\mathsf{FDSO}(s)$ is defined by the white constraint

$$AX^{d-1}, \quad H^{s+1}X^{d-s-1}, \quad T^{d-s+1}X^{s-1},$$

and black constraint

$$X[AHTX]^{\delta-1}, \quad HT[AHTX]^{\delta-2}.$$

We show that $\mathsf{FDSO}(s)$ is a *fixed point* for the round elimination technique, and a lower bound of $\Omega(\log n)$ then follows.

Randomized complexity. To conclude this work, we take a closer look at the problems of type **VII**, i.e., problems with deterministic complexity $\Theta(\log n)$. Together with prior work [14, 16], our results imply that all such problems have randomized complexity either $\Theta(\log \log n)$ or $\Theta(\log n)$ rounds. Moreover, there are already many known examples of problems of type **VII** that fall in the class of $\Theta(\log \log n)$; examples include sinkless orientation [15], as well as many orientation and splitting problems that are known to be as easy as sinkless orientation [20]. We focus on new hardness results: we give a list of problem families within type **VII** that fall in the class of $\Theta(\log n)$-round randomized complexity.

Our list of problems of type **VII** that require $\Theta(\log n)$ rounds with randomized algorithms is *not* complete. The main open question for future work is completing the list of all such binary labeling problems.

6 Discussion and additional related work

In this work we initiated a systematic investigation of the distributed complexity of LCL problems on trees, and we presented a complete characterization of the deterministic distributed complexity of a specific family of LCL problems, binary labeling problems. To conclude this

work, we will discuss additional connections between the present work and related work – in particular, highlighting the role of binary labeling problems in the recent developments in the distributed computational complexity theory – and we will also introduce new directions for future research.

Sinkless orientation is a binary labeling problem. Sinkless orientation (Example 4.1) is one of the cornerstones of the modern theory of distributed computational complexity. This problem was introduced in our context in 2016 [10], and in Google Scholar there are already more than 30 papers written since 2016 that mention the term "sinkless orientation", all of them related to the theory of distributed computing. There are many variants of the definition, but for our purposes all of them are in essence equivalent to each other, so let us use the following version: there is a fixed parameter $d > 2$, and the task is to orient edges so that nodes of degree d are not sinks (note that low-degree nodes can be sinks and the problem is therefore trivial in cycles).

The distributed complexity of solving sinkless orientation is now completely understood: $\Theta(\log n)$ rounds with deterministic algorithms and $\Theta(\log \log n)$ rounds with randomized algorithms [10, 15, 22]. Yet there are many fundamental questions related to the role of sinkless orientation that we do not understand at all.

One of the mysteries is the following observation: whenever we encounter a problem Π that turns out to be as hard as sinkless orientation, usually the reason for Π being hard is that Π is directly related to sinkless orientation through reductions. For example, sinkless and sourceless orientation (Example 4.3) has the same complexity as sinkless orientation; the upper bound is nontrivial [22], but the lower bound trivially comes from the observation that a sinkless and sourceless orientation gives a sinkless orientation. Before this work, we were not aware of any LCL problem Π that has the same complexity as sinkless orientation – $\Theta(\log n)$ rounds deterministic and $\Theta(\log \log n)$ rounds randomized – with a nontrivial lower bound that is not merely an observation that an algorithm for solving Π directly gives an algorithm for solving sinkless orientation.

Indeed, we did not know if all problems in this complexity class are merely extensions and variants of the same problem!

Our systematic study of binary labeling problems succeeded in shedding new light on this issue: we identified problems that are as hard as sinkless orientation, yet require a *new lower bound proof* that is not based on the previous result of the hardness of sinkless orientation.

Automatic round elimination and fixed points. The new problems that we discovered are also directly related to another ongoing research topic: understanding the *automatic round elimination technique* [9, 28]. As we mentioned in Section 2, the edge labeling formalism makes it easy to apply the round elimination technique, which is an effective technique for proving lower bounds [3, 4, 8–10, 13, 24, 26]. However, there are still many open research questions related to round elimination itself. One of the key questions is about fixed points; here the sinkless orientation problem is a good example.

Let Π be the sinkless orientation problem from Example 4.1. If we start with the assumption that the complexity of Π is $T = o(\log n)$ rounds in regular trees, and apply the round elimination technique [9, 28] in a mechanical manner, we will immediately get the result that the complexity of the same problem Π is $T - 1$ rounds, which is absurd, and hence we immediately get a lower bound (at least for some weak models of distributed computing). A bit more formally, Π is a *nontrivial fixed point* for round elimination, and such fixed points immediately imply nontrivial lower bounds. This is particularly interesting as fixed points can be detected automatically with a computer.

Now let Π' be the problem of finding a sinkless and sourceless orientation from Example 4.3. For a human being, it is now trivial that Π' is at least as hard as Π. However, Π' is not a fixed point for round elimination, and we do not seem to have any automatic way for proving lower bounds for problems similar to Π'.

What is not understood at all is what is the fundamental difference between Π and Π' and why one of them is a fixed point and the other one is not. Indeed, so far there have not been many examples of nontrivial fixed points that are not merely trivial variants of sinkless orientations. Our work now presented the first new examples of nontrivial fixed points, and these new examples will hopefully inspire follow-up work towards a better understanding of fixed points in round elimination in general.

Binary labeling problems vs. homogeneity. LCLs come in many flavors. Some problems are trivial in a regular unlabeled graph – fractional problems are a good example here. However, for many distributed problems the interesting part is specifically what to do in the middle of a regular unlabeled tree – symmetry-breaking problems and orientation and splitting problems fall in this category. Our focus was on the latter case.

In the definition of binary labeling problems, the idea of *focusing on regular parts* is captured in the constraint that only white nodes of degree d and only black nodes of degree δ are relevant. In essence, as soon as we see some irregularities (e.g., leaf nodes of the tree), the problem becomes potentially easier to solve.

The idea of investigating what happens in the middle of a regular tree was previously formalized using a somewhat different idea of *homogeneous LCLs* [8]. Both homogeneous LCLs and binary labeling problems are defined so that the problem is interesting in a d-regular part of a tree and any irregularities make the problem easier to solve. However, this is achieved through a different mechanism:

- Homogeneous LCLs [8]: Instead of solving the original problem Π in some neighborhood, you can create a pointer. The pointers have to form a chain, and a chain of pointers can only terminate at a node with degree different from d.
- Binary labeling problems (this work): The labels are constrained only for nodes of degree d. The labels incident to other nodes are unconstrained.

In practice, this means that homogeneous LCLs are always solvable in trees, and they can be solved in $O(\log n)$ rounds (assuming $d > 2$): we can always find a node v of degree 1 or 2 within distance $O(\log n)$, and hence instead of solving the original problem, we can construct a pointer chain towards v. Therefore homogeneous LCLs are well-suited for the original purpose of investigating distributed computational complexity in the $o(\log n)$ region, but they can neither provide insights into the $\Omega(\log n)$ regime nor classify unsolvable problems. Our work on binary LCLs gives complete answers for both.

On the other hand, homogeneous problems are not restricted to two labels, and hence it is possible that there are complexity classes below $o(\log n)$ that do not exist for binary labeling problems. As we see in Table 3, this is indeed the case: problems with complexity $\Theta(\log^* n)$ exist among homogeneous problems, but as we have seen in this work, they do not exist among binary labeling problems.

Restriction to two labels and $\Theta(\log^* n)$ complexity. In a binary labeling problem we label edges (or half-edges) with two labels: whether it is part of solution X or not. If we looked at the more general case of $|\Sigma| = O(1)$ possible edge labels, we would have a family of problems that is, in essence, more expressive than the family of all homogeneous LCLs.

Table 3 An overview of possible distributed time complexities in trees for different classes of LCLs.

Deterministic complexity	Randomized complexity	General LCLs	Homog. LCLs	**Binary labeling**	Examples
$O(1)$	$O(1)$	YES	YES	**YES**	trivial problems
$\omega(1)$, $o(\log^* n)$	$\omega(1)$, $o(\log^* n)$	?	NO [8]	**NO**	?
$\Theta(\log^* n)$	$\Theta(\log^* n)$	YES	YES	**NO**	$(\Delta+1)$-coloring [17, 24]
$\Theta(\log n)$	$\Theta(\log \log n)$	YES	YES	**YES**	sinkless orientation [10, 15, 22]
$\Theta(\log n)$	$\Theta(\log n)$	YES	YES	**YES**	even orientation (Ex. 4.4)
$\Theta(n^{1/k})$	$\Theta(n^{1/k})$	YES	NO [8]	**NO**	$2\frac{1}{2}$-coloring [16]
$\Theta(n)$	$\Theta(n)$	YES	NO [8]	**YES**	2-coloring (Ex. 4.5)

The case of $|\Sigma| = 2$ that we studied here is the smallest nontrivial case, but there is another reason that makes the case of $|\Sigma| = 2$ interesting: as mentioned above, we have seen that the *complexity class* $\Theta(\log^* n)$ *disappears* when we go from $|\Sigma| = 3$ down to $|\Sigma| = 2$.

There are numerous symmetry-breaking problems that can be solved in $\Theta(\log^* n)$ rounds in graphs of maximum degree $\Delta = O(1)$. Examples include maximal matching, maximal independent set, minimal dominating set, $(\Delta + 1)$-vertex coloring, $(2\Delta - 1)$-edge coloring, weak 2-coloring, and many variants of these problems.

It is known that with 3 edge labels we can encode e.g. the problem of finding a maximal matching [3]; hence as soon as $|\Sigma| \geq 3$, there are edge labeling problems solvable in $\Theta(\log^* n)$ rounds. However, previously it was not known if the use of $|\Sigma| \geq 3$ labels was necessary in order to encode any such problem. After all, the use of $|\Sigma| \geq 3$ labels seems unnatural when we consider problems such as maximal matching, maximal independent set, and weak 2-coloring, all of which in essence ask one to find a subset of edges or a subset of nodes subject to local constraints.

The results of this work imply that none of these problems can be encoded with two labels. We have showed that for binary labeling problems, there is a gap in the deterministic complexity between $\omega(1)$ and $o(\log n)$. Hence binary labeling problems give rise to a different landscape of computational complexity in comparison with any other number of labels. We conjecture that there is no such qualitative difference between e.g. alphabets of size 3 or 4.

References

1 Alkida Balliu, Sebastian Brandt, Yi-Jun Chang, Dennis Olivetti, Mikaël Rabie, and Jukka Suomela. The Distributed Complexity of Locally Checkable Problems on Paths is Decidable. In *Proc. 38th ACM Symposium on Principles of Distributed Computing (PODC 2019)*, pages 262–271. ACM Press, 2019. doi:10.1145/3293611.3331606.

2 Alkida Balliu, Sebastian Brandt, Yuval Efron, Juho Hirvonen, Yannic Maus, Dennis Olivetti, and Jukka Suomela. Classification of distributed binary labeling problems, 2019. arXiv:1911.13294.

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

4 Alkida Balliu, Sebastian Brandt, and Dennis Olivetti. Distributed Lower Bounds for Ruling Sets. In *Proc. 61st IEEE Symposium on Foundations of Computer Science (FOCS 2020)*. IEEE, 2020. arXiv:2004.08282.

5 Alkida Balliu, Sebastian Brandt, Dennis Olivetti, and Jukka Suomela. Almost global problems in the LOCAL model. In *Proc. 32nd International Symposium on Distributed Computing (DISC 2018)*, Leibniz International Proceedings in Informatics (LIPIcs), pages 9:1–9:16. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.DISC.2018.9.

6 Alkida Balliu, Sebastian Brandt, Dennis Olivetti, and Jukka Suomela. How much does randomness help with locally checkable problems? In *Proc. 39th ACM Symposium on Principles of Distributed Computing (PODC 2020)*. ACM Press, 2020. doi:10.1145/3382734.3405715.

7 Alkida Balliu, Juho Hirvonen, Janne H Korhonen, Tuomo Lempiäinen, Dennis Olivetti, and Jukka Suomela. New classes of distributed time complexity. In *Proc. 50th ACM Symposium on Theory of Computing (STOC 2018)*, pages 1307–1318. ACM Press, 2018. doi:10.1145/3188745.3188860.

8 Alkida Balliu, Juho Hirvonen, Dennis Olivetti, and Jukka Suomela. Hardness of Minimal Symmetry Breaking in Distributed Computing. In *Proc. 38th ACM Symposium on Principles of Distributed Computing (PODC 2019)*, pages 369–378. ACM Press, 2019. doi:10.1145/3293611.3331605.

9 Sebastian Brandt. An Automatic Speedup Theorem for Distributed Problems. In *Proc. 38th ACM Symposium on Principles of Distributed Computing (PODC 2019)*, pages 379–388. ACM Press, 2019. doi:10.1145/3293611.3331611.

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 *Proc. 48th ACM Symposium on Theory of Computing (STOC 2016)*, pages 479–488. ACM Press, 2016. doi:10.1145/2897518.2897570.

11 Sebastian Brandt, Juho Hirvonen, Janne H Korhonen, Tuomo Lempiäinen, Patric R J Östergård, Christopher Purcell, Joel Rybicki, Jukka Suomela, and Przemysław Uznański. LCL problems on grids. In *Proc. 36th ACM Symposium on Principles of Distributed Computing (PODC 2017)*, pages 101–110. ACM Press, 2017. doi:10.1145/3087801.3087833.

12 Sebastian Brandt, Yannic Maus, and Jara Uitto. A sharp threshold phenomenon for the distributed complexity of the lovász local lemma. In *Proc. 38th ACM Symposium on Principles of Distributed Computing (PODC 2019)*, pages 389–398. ACM Press, 2019. doi:10.1145/3293611.3331636.

13 Sebastian Brandt and Dennis Olivetti. Truly Tight-in-Δ Bounds for Bipartite Maximal Matching and Variants. In *Proc. 39th ACM Symposium on Principles of Distributed Computing (PODC 2020)*. ACM Press, 2020. doi:10.1145/3382734.3405745.

14 Yi-Jun Chang, Qizheng He, Wenzheng Li, Seth Pettie, and Jara Uitto. The Complexity of Distributed Edge Coloring with Small Palettes. In *Proc. 29th ACM-SIAM Symposium on Discrete Algorithms (SODA 2018)*, pages 2633–2652. Society for Industrial and Applied Mathematics, 2018. doi:10.1137/1.9781611975031.168.

15 Yi-Jun Chang, Tsvi Kopelowitz, and Seth Pettie. An Exponential Separation between Randomized and Deterministic Complexity in the LOCAL Model. In *Proc. 57th IEEE Symposium on Foundations of Computer Science (FOCS 2016)*, pages 615–624. IEEE, 2016. doi:10.1109/FOCS.2016.72.

16 Yi-Jun Chang and Seth Pettie. A Time Hierarchy Theorem for the LOCAL Model. *SIAM Journal on Computing*, 48(1):33–69, 2019. doi:10.1137/17M1157957.

17 Richard Cole and Uzi Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control*, 70(1):32–53, 1986. doi:10.1016/S0019-9958(86)80023-7.

18 Manuela Fischer and Mohsen Ghaffari. Sublogarithmic Distributed Algorithms for Lovász Local Lemma, and the Complexity Hierarchy. In *Proc. 31st International Symposium on Distributed Computing (DISC 2017)*, pages 18:1–18:16, 2017. doi:10.4230/LIPIcs.DISC.2017.18.

19 Mohsen Ghaffari, David G Harris, and Fabian Kuhn. On Derandomizing Local Distributed Algorithms. In *Proc. 59th IEEE Symposium on Foundations of Computer Science (FOCS 2018)*, pages 662–673, 2018. doi:10.1109/FOCS.2018.00069.

20. Mohsen Ghaffari, Juho Hirvonen, Fabian Kuhn, Yannic Maus, Jukka Suomela, and Jara Uitto. Improved distributed degree splitting and edge coloring. In *Proc. 31st International Symposium on Distributed Computing (DISC 2017)*, volume 91 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 19:1–19:15. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2017. `doi:10.4230/LIPIcs.DISC.2017.19`.
21. Mohsen Ghaffari, Fabian Kuhn, and Yannic Maus. On the complexity of local distributed graph problems. In *Proc. 49th ACM SIGACT Symposium on Theory of Computing (STOC 2017)*, pages 784–797. ACM Press, 2017. `doi:10.1145/3055399.3055471`.
22. Mohsen Ghaffari and Hsin-Hao Su. Distributed Degree Splitting, Edge Coloring, and Orientations. In *Proc. 28th ACM-SIAM Symposium on Discrete Algorithms (SODA 2017)*, pages 2505–2523. Society for Industrial and Applied Mathematics, 2017. `doi:10.1137/1.9781611974782.166`.
23. Janne H Korhonen and Jukka Suomela. Towards a complexity theory for the congested clique. In *Proc. 30th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2018)*, pages 163–172. ACM Press, 2018. `doi:10.1145/3210377.3210391`.
24. Nathan Linial. Locality in Distributed Graph Algorithms. *SIAM Journal on Computing*, 21(1):193–201, 1992. `doi:10.1137/0221015`.
25. Gary L. Miller and John H. Reif. Parallel tree contraction and its application. In *Proc. 26th Annual Symposium on Foundations of Computer Science (FOCS 1985)*, pages 478–489. IEEE, 1985. `doi:10.1109/SFCS.1985.43`.
26. Moni Naor. A lower bound on probabilistic algorithms for distributive ring coloring. *SIAM Journal on Discrete Mathematics*, 4(3):409–412, 1991. `doi:10.1137/0404036`.
27. Moni Naor and Larry Stockmeyer. What Can be Computed Locally? *SIAM Journal on Computing*, 24(6):1259–1277, 1995. `doi:10.1137/S0097539793254571`.
28. Dennis Olivetti. Round Eliminator: a tool for automatic speedup simulation, 2020. URL: `https://github.com/olidennis/round-eliminator`.
29. David Peleg. *Distributed Computing: A Locality-Sensitive Approach*. Society for Industrial and Applied Mathematics, 2000. `doi:10.1137/1.9780898719772`.
30. Will Rosenbaum and Jukka Suomela. Seeing Far vs. Seeing Wide: Volume Complexity of Local Graph Problems. In *Proc. 39th ACM Symposium on Principles of Distributed Computing (PODC 2020)*. ACM Press, 2020. `doi:10.1145/3382734.3405721`.
31. Václav Rozhoň and Mohsen Ghaffari. Polylogarithmic-Time Deterministic Network Decomposition and Distributed Derandomization. In *Proc. 52nd Annual ACM Symposium on Theory of Computing (STOC 2020)*, 2020. `doi:10.1145/3357713.3384298`.

The Complexity Landscape of Distributed Locally Checkable Problems on Trees

Yi-Jun Chang
ETH Zürich, Switzerland
yi-jun.chang@eth-its.ethz.ch

Abstract

Recent research revealed the existence of *gaps* in the complexity landscape of *locally checkable labeling* (LCL) problems in the LOCAL model of distributed computing. For example, the deterministic round complexity of any LCL problem on bounded-degree graphs is either $O(\log^* n)$ or $\Omega(\log n)$ [Chang, Kopelowitz, and Pettie, FOCS 2016]. The complexity landscape of LCL problems is now quite well-understood, but a few questions remain open.

For bounded-degree trees, there is an LCL problem with round complexity $\Theta(n^{1/k})$ for each positive integer k [Chang and Pettie, FOCS 2017]. It is conjectured that no LCL problem has round complexity $o(n^{1/(k-1)})$ and $\omega(n^{1/k})$ on bounded-degree trees. As of now, only the case of $k = 2$ has been proved [Balliu et al., DISC 2018].

In this paper, we show that for LCL problems on bounded-degree trees, there is indeed a gap between $\Theta(n^{1/(k-1)})$ and $\Theta(n^{1/k})$ for each $k \geq 2$. Our proof is *constructive* in the sense that it offers a sequential algorithm that decides which side of the gap a given LCL problem belongs to. We also show that it is EXPTIME-hard to distinguish between $\Theta(1)$-round and $\Theta(n)$-round LCL problems on bounded-degree trees. This improves upon a previous PSPACE-hardness result [Balliu et al., PODC 2019].

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Distributed algorithms, LOCAL model, locally checkable labeling

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.18

Related Version A full version of the paper is available at `https://arxiv.org/abs/2009.09645`.

1 Introduction

In this paper, we consider Linial's LOCAL model of distributed computing [17, 21], where the input graph $G = (V, E)$ and the communication network are identical. Each vertex $v \in V$ corresponds to a processor, each edge $e \in E$ corresponds to a communication link, and the computation proceeds in synchronized rounds. There is no restriction on the local computation power and the message size, and the main complexity measure for an algorithm is the number of rounds. We assume that the number of vertices $n = |V|$ and the maximum degree $\Delta = \max_{v \in V} \deg(v)$ are global knowledge.

There is a recent line of research [3, 4, 5, 6, 8, 10, 11, 12, 15, 16, 22] aiming to systematically understand the round complexity of distributed graph problems, with a focus on the *locally checkable labelings* (LCL) problems [19], which is the class of distributed problems whose solution is locally verifiable by examining a constant-radius neighborhood of each vertex. The class of LCL problems is sufficiently general that it encompasses many well-studied problems in the LOCAL model, such as maximal matching, maximal independent set (MIS), $(\Delta + 1)$-vertex coloring, and sinkless orientation. For example, in the $(\Delta + 1)$-vertex coloring problem, the output of each vertex v is a color $c(v) \in \{1, 2, \ldots, \Delta + 1\}$. The output is a legal solution if $c(u) \neq c(v)$ for each edge $e = \{u, v\} \in E$. Each vertex v can locally check if it has a neighbor $u \in N(v)$ with $c(u) = c(v)$ by examining the output within its radius-1 neighborhood.

© Yi-Jun Chang;
licensed under Creative Commons License CC-BY
34th International Symposium on Distributed Computing (DISC 2020).
Editor: Hagit Attiya; Article No. 18; pp. 18:1–18:17
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1.1 The Spectrum of Distributed Complexities

Different from the sequential setting such as the Turing machine or the RAM model, in the complexity landscape of LCL problems in the LOCAL model, large *gaps* exist in the complexity landscape.

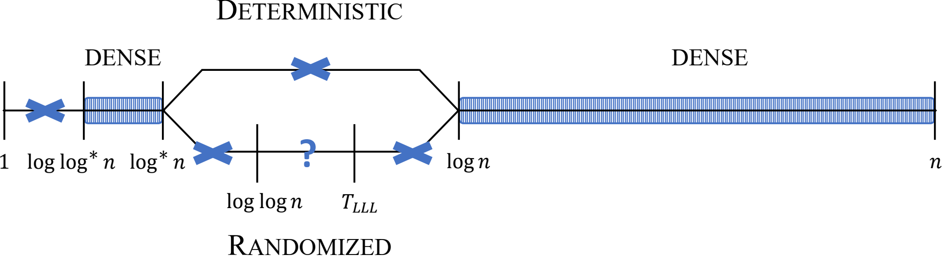


Figure 1 Complexity landscape of LCLs on bounded-degree general graphs.

General graphs. Chang, Kopelowitz, and Pettie [11] showed that for any LCL problem on bounded-degree graphs, its deterministic round complexity is either $O(\log^* n)$ or $\Omega(\log n)$, and its randomized round complexity is either $O(\log^* n)$ or $\Omega(\log\log n)$. Chang and Pettie [12] showed that any $o(\log n)$-round randomized algorithm for an LCL problem can be accelerated to run in $O(T_{\text{LLL}})$ rounds, where T_{LLL} is the randomized complexity of the distributed constructive Lovász Local Lemma (LLL) [14] under a polynomial criterion $pd^c = O(1)$ for any positive constant c. It was conjectured in [12] that $T_{\text{LLL}} = \Theta(\log\log n)$. Chang and Pettie also showed that the gap $\omega(1) - o(\log\log^* n)$ can be derived using the approach of Naor and Stockmeyer [19]. Balliu et al. showed that the two remaining regions $[\Theta(\log\log^* n), \Theta(\log^* n)]$ and $[\Theta(\log n), \Theta(n)]$ are *dense* in that many round complexity functions within these ranges can be realized by LCL problems [5]. See Figure 1 for an illustration of the complexity landscape of LCLs on bounded-degree general graphs.

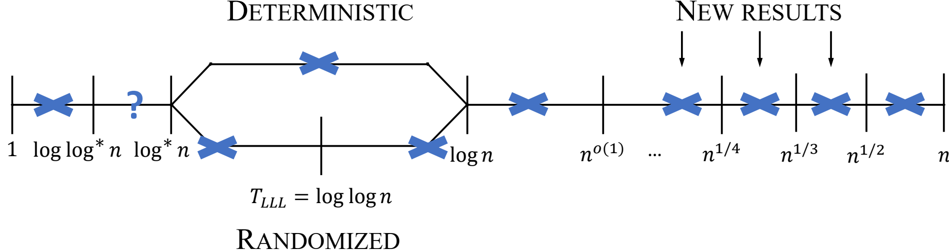


Figure 2 Complexity landscape of LCLs on bounded-degree trees.

Trees. The four gaps in the lower end of the spectrum $[\Theta(1), \Theta(\log n)]$ are the same as the setting of general graphs. It was conjectured in [12] that the $\omega(1) - o(\log\log^* n)$ gap on bounded-degree trees can be extended to $\omega(1) - o(\log^* n)$. So far this conjecture was proved only for the special case of *homogeneous* problems [6]. For the higher end of the spectrum $[\Theta(\log n), \Theta(n)]$, it was proved in [12] that any distributed algorithm that takes $n^{o(1)}$ rounds can be accelerated to run in just $O(\log n)$ rounds, and there exists an LCL problem with complexity $\Theta(n^{1/k})$ for each $k \geq 1$. It was left as an open problem to decide if there are gaps

between them. Recently, Balliu et al. [3] showed that there is indeed a gap between $\Theta(\sqrt{n})$ and $\Theta(n)$, the other cases are still open. See Figure 2 for an illustration of the complexity landscape of LCLs on bounded-degree trees.

New result. In this paper, we prove the existence of the gap $\omega(n^{1/k})$ – $o(n^{1/(k-1)})$ of LCL problems on bounded-degree trees, for each integer $k \geq 2$.

▶ **Theorem 1.** *For any integer $k \geq 2$, for any LCL problem $\mathcal{P}$ on bounded-degree trees, either one of the following holds.*

- *The deterministic and randomized round complexities of $\mathcal{P}$ are $\Omega(n^{1/(k-1)})$.*
- *The deterministic and randomized round complexities of $\mathcal{P}$ are $O(n^{1/k})$.*

Furthermore, there is a sequential algorithm that given an integer $k \geq 2$ and a description of $\mathcal{P}$ decides which side of the gap $\mathcal{P}$ belongs to.

The proof of Theorem 1 is obtained by unifying the approach of Balliu et al. [3] and the approach of Chang and Pettie [12] using a generalized tree decomposition algorithm.

Theorem 1 implies that randomness does not help for LCL problems on bounded-degree trees in the regime of polynomial round complexity.

1.2 The Complexity of Classification

The complexity gaps in Figures 1 and 2 classify the distributed problems into complexity classes. A natural question to ask is whether this classification is *decidable*. Unfortunately, even for grids and tori, it is *undecidable* whether a given LCL problem can be solved in $O(1)$ rounds [9, 19], since LCL on grids can be used to simulate a Turing machine. This undecidability result does not apply to special graph classes such as paths, cycles, and trees. In fact, the proof of the $\omega(\log n)$ – $n^{o(1)}$ gap on bounded-degree trees in [12] is *constructive* in the sense that it gives us an algorithm that can decide whether a given LCL problem on bounded-degree trees has complexity $O(\log n)$ or $n^{\Omega(1)}$.

Much progress has recently been made in understanding to what extent the design of distributed algorithms and the proof of distributed lower bounds can be automated [1, 2, 7, 9, 13, 20]. On paths or cycles, with or without input labels, only three complexity classes are possible: $\Theta(1)$, $\Theta(\log^* n)$, and $\Theta(n)$. Balliu et al. [1] showed that for any given LCL problem $\mathcal{P}$ on paths or cycles, it is *decidable* to check which class $\mathcal{P}$ belongs to, and there is a sequential algorithm that automates the design of an asymptotically optimal distributed algorithm for $\mathcal{P}$. For comparison, the previous proofs [11, 12, 19] establishing this classification did not offer such results.

On the negative side, Balliu et al. [1] showed that the problem of determining the optimal asymptotic distributed complexity is PSPACE-hard, even for paths and cycles with input labels. Since trees can be used to encode input labels, the same PSPACE-hardness result extends to the case of bounded-degree trees without input labels.

New result. Our proof of the existence of the gap $\omega(n^{1/k})$ – $o(n^{1/(k-1)})$ offers a sequential algorithm that decides which side of the gap a given LCL problem $\mathcal{P}$ belongs to. When the locality radius r of the LCL is a constant independent of the description length N of the LCL, the runtime $2^{2^{N^{O(1)}}}$ of our sequential algorithm is *doubly exponential* in $N^{O(1)}$. To complement this result, we show that this problem is inherently very hard by proving that this problem is EXPTIME-hard. Specifically, we say that a round complexity function $T(n)$ is *realizable* if there exists an LCL problem $\mathcal{P}$ whose round complexity is $\Theta(T(n))$ on bounded-degree trees. We prove the following theorem.

▶ **Theorem 2.** *Let $T_1(n) \ll T_2(n)$ be two realizable round complexity functions. Given an LCL problem $\mathcal{P}$ that is promised to have round complexity either $\Theta(T_1(n))$ or $\Theta(T_2(n))$ on bounded-degree trees, it is* EXPTIME*-hard to decide the round complexity of $\mathcal{P}$.*

1.3 Organization

In Section 2, we overview the basics of LCL problems and review the pumping lemma of Chang and Pettie [12]. In Section 3, we review the proof of the $\omega(n^{1/2}) - o(n)$ gap by Balliu et al. [3]. In Section 4, we consider a generalized version of the tree decomposition of Miller and Reif [18] that allows us to unify the approach of Balliu et al. [3] and the approach of Chang and Pettie [12]. In Section 5, we prove Theorem 1 for the case of deterministic algorithms. The complete proofs of Theorems 1 and 2 are left to the full version of the paper.

2 Preliminaries

In the deterministic variant of the LOCAL model, each vertex v has a distinct $O(\log n)$-bit identifier $\mathrm{ID}(v)$. In the randomized variant of the LOCAL model, there are no distinct identifiers, but each vertex has access to a stream of unbiased random bits, and the maximum tolerable global probability of failure is $1/n$. Note that a t-round LOCAL algorithm can be seen as a function that maps a radius-t subgraph centered at v to an output label assigned to v.

2.1 Locally Checkable Labeling

A distributed graph problem is locally checkable if there is some constant r such that the validity of a solution can be checked locally by having each vertex examine its radius-r neighborhood. For example, the maximal independent set (MIS) problem is locally checkable with locality radius $r = 1$, but the maximum independent set problem is not locally checkable.

Formal definition. Formally, an LCL problem $\mathcal{P}$ is specified by the following parameters: the locality radius r, the set of input labels Σ_{in}, the set of output labels Σ_{out}, and the set of allowed configurations $\mathcal{C}$. Each member of $\mathcal{C}$ is a radius-r subgraph H centered at a specific vertex v, where each vertex in H is assigned an input label from Σ_{in} and an output label from Σ_{out}. Note that $|\Sigma_{\mathrm{in}}| = 1$ corresponds to the special case where there is no input label.

An *instance* of an LCL problem $\mathcal{P}$ is a graph $G = (V, E)$ where each vertex is assigned an input label from Σ_{in}. A *solution* for $\mathcal{P}$ on G is a labeling function ϕ_{out} that assigns to each vertex in G an output label from Σ_{out}. We say that ϕ_{out} is *locally consistent* for a vertex $v \in V$ if its radius-r neighborhood $N^r(v)$ is an allowed configuration in $\mathcal{C}$ under the given input labeling and the output labeling ϕ_{out}. The output labeling ϕ_{out} is *legal* if it is locally consistent everywhere.

Graph terminology. Unless otherwise stated, all vertices in all graphs in this paper are assigned input labels from Σ_{in}, and the term *label* refers to *output label*. An *unlabeled graph* is a graph where no vertex is assigned an output label from Σ_{out}. A *partially labeled graph* is a graph with a labeling function $\mathcal{L}$ that maps each vertex v to an element of $\Sigma_{\mathrm{out}} \cup \{\bot\}$. A *completely labeled graph* or a *labeled graph* is a graph with a labeling function $\mathcal{L}$ that maps each vertex v to an element of Σ_{out}.

Description length. We assume that any given LCL problem $\mathcal{P}$ is specified by representing $\mathcal{C}$ as a truth table. Specifically, a *centered graph* is a graph $G = (V, E)$ with a distinguished vertex $s \in V$, and the *radius* of G is defined by $\max_{v \in V} \operatorname{dist}(v, s)$. The truth table representation of $\mathcal{P}$ is a mapping $\mathcal{G}_{r,\Delta,\Sigma_{\text{in}},\Sigma_{\text{out}}} \mapsto \{0, 1\}$, where $\mathcal{G}_{r,\Delta,\Sigma_{\text{in}},\Sigma_{\text{out}}}$ is the set of all centered graphs $G = (V, E)$ of radius at most r with maximum degree Δ where each vertex $v \in V$ is equipped with an input label from Σ_{in} and an output label from Σ_{out}.

If the graph class under consideration is the set of trees of maximum degree Δ, then the description length of $\mathcal{P}$ can be upper bounded by $(1 + |\Sigma_{\text{in}}| \cdot |\Sigma_{\text{out}}|)^{1+\Delta^r}$.

To derive this upper bound, consider the rooted tree T_r of height r where the root v has Δ children, all vertices u with $1 \le \operatorname{dist}(u, v) \le r - 1$ have $\Delta - 1$ children, and all vertices u with $\operatorname{dist}(u, v) = r$ are leaf vertices. The number of trees in $\mathcal{G}_{r,\Delta,\Sigma_{\text{in}},\Sigma_{\text{out}}}$ is at most the number of distinct labeling of the vertices in T_r by $(\Sigma_{\text{in}} \times \Sigma_{\text{out}}) \cup \{\star\}$, where $\star$ is a special symbol indicating the non-existence of a vertex. Hence the description length can be upper bounded by $(1 + |\Sigma_{\text{in}}| \cdot |\Sigma_{\text{out}}|)^{n_r}$, where n_r is the number of vertices in T_r. We have $n_0 = 1$, $n_1 = 1 + \Delta$, and $n_r = 1 + \Delta + \Delta \sum_{i=1}^{r-1} (\Delta - 1)^{r-1}$ for each $r \ge 2$. It is clear that $n_r \le 1 + \Delta^r$ for all r.

Remarks on edge labeling and orientation. In general, an LCL might have edge labels and edge orientation. It is straightforward to encode edge labels and edge orientation as vertex labels. For example, given an input graph G, consider the following pre-processing. For each edge $e = \{u, v\} \in E$, subdivide it into a length-3 path $(u, x_{e,u}, x_{e,v}, v)$ by adding two new vertices $x_{e,u}$ and $x_{e,v}$. Each newly added vertex is assigned a special input label e indicating that it represents a half of an edge. Now an edge orientation $u \to v$ can be encoded as $\phi(x_{e,u}) = 0$ and $\phi(x_{e,v}) = 1$.

2.2 Pumping Lemma

We review the pumping lemma of Chang and Pettie [12], which plays a crucial role in establishing complexity gaps on trees.

Notation for partially labeled graphs. A *partially labeled graph* $\mathcal{G} = (G, \mathcal{L})$ is a graph $G = (V, E)$ together with a function $\mathcal{L} : V \to \Sigma_{\text{out}} \cup \{\bot\}$. The vertices in $\mathcal{L}^{-1}(\bot)$ are *unlabeled*. A *complete labeling* $\mathcal{L}' : V(G) \to \Sigma_{\text{out}}$ for $\mathcal{G}$ is one that labels all vertices and is consistent with $\mathcal{G}$'s partial labeling, i.e., $\mathcal{L}'(v) = \mathcal{L}(v)$ whenever $\mathcal{L}(v) \neq \bot$. A *legal labeling* is a complete labeling that is *locally consistent* for all $v \in V(G)$, i.e., the labeled subgraph induced by $N^r(v)$ is consistent with the given LCL problem $\mathcal{P}$. Here $N^r(v)$ is the set of all vertices within distance r of v. A subgraph of a partially labeled graph $\mathcal{G} = (G, \mathcal{L})$ is a pair $\mathcal{H} = (H, \mathcal{L}')$ such that H is a subgraph of G, and $\mathcal{L}'$ is $\mathcal{L}$ restricted to the domain $V(H)$. With a slight abuse of notation, we usually write $\mathcal{H} = (H, \mathcal{L})$.

An equivalence relation. A tree $\mathcal{H}$ with two distinguished vertices $s, t \in V(H)$ is called a *bipolar* tree. We call s and t the two *poles* of $\mathcal{H}$. We consider the equivalence relation $\overset{\star}{\sim}$ on bipolar trees defined in [12]. We write $\mathsf{Type}(\mathcal{H})$ to denote the equivalence class of the bipolar tree $\mathcal{H}$. The following property of $\overset{\star}{\sim}$ is crucial.

Suppose we are given the following.

- $\mathcal{G}$ is any graph.

- $\mathcal{H}$ is a bipolar subtree of $\mathcal{G}$ with two poles s and t such that the removal of s and t disconnects $\mathcal{H}$ from the rest of $\mathcal{G}$.
- $\mathcal{H}'$ is another bipolar subtree with two poles s' and t' such that $\mathsf{Type}(\mathcal{H}) = \mathsf{Type}(\mathcal{H}')$.
- $\mathcal{L}_\diamond$ is any complete legal labeling of $\mathcal{G}$.

Define the graph $\mathcal{G}'$ as the result of replacing the subgraph $\mathcal{H}$ of $\mathcal{G}$ with $\mathcal{H}'$. Then there exists a legal labeling $\mathcal{L}'$ of $\mathcal{H}'$ meeting the following conditions.

- The following complete labeling $\mathcal{L}'_\diamond$ of $\mathcal{G}'$ is a legal labeling.

$$\mathcal{L}'_\diamond(v) = \begin{cases} \mathcal{L}'(v) & \text{if } v \in \mathcal{H}, \\ \mathcal{L}_\diamond(v) & \text{if } v \in \mathcal{G} \setminus \mathcal{H}. \end{cases}$$

- Such a labeling $\mathcal{L}'$ of $\mathcal{H}'$ can be computed *solely* from $\mathcal{H}'$ and the given labeling $\mathcal{L}_\diamond$ restricted to $\mathcal{H}$.

In view of the above, the vertices in $\mathcal{H}'$ can compute their $\mathcal{L}'$-labels using only information within $\mathcal{H}'$ and the given labeling $\mathcal{L}_\diamond$ restricted to $\mathcal{H}$, without communicating with the vertices outside of $\mathcal{H}'$. Intuitively, this allows us to reduce the task of finding a legal labeling $\mathcal{L}'$ of $\mathcal{G}'$ to the task of finding a legal labeling $\mathcal{L}_\diamond$ of $\mathcal{G}$.

A pumping lemma for bipolar trees. The unique path $(s = u_1, u_2, \ldots, u_k = t)$ connecting the two poles s and t of a bipolar tree $\mathcal{H}$ is called the *core path* of $\mathcal{H}$. The tree $\mathcal{H}$ can be viewed as a string of subtrees $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_k$, where $\mathcal{T}_i$ is the subtree of $\mathcal{H}$ rooted at u_i. For the sake of convenience, we use the following string notation $\mathcal{H} = (\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_k)$ to describe a bipolar tree $\mathcal{H}$. Viewing bipolar trees as strings, the following *pumping lemma* was proved in [12].

There exists a number ℓ_{pump} depending only on the given LCL problem $\mathcal{P}$ such that as long as $k \geq \ell_{\text{pump}}$, any bipolar tree $\mathcal{H} = (\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_k)$ can be decomposed into three substrings $\mathcal{H} = x \circ y \circ z$ meeting the following conditions.

- $|xy| \leq \ell_{\text{pump}}$.
- $|y| \geq 1$.
- $\mathsf{Type}(x \circ y^j \circ z) = \mathsf{Type}(\mathcal{H})$ for each non-negative integer j.

Intuitively, the pumping lemma allows us to extend the length of $\mathcal{H} = (\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_k)$ to arbitrarily long without changing its type, as long as $k \geq \ell_{\text{pump}}$.

3 A Review of the $\omega(n^{1/2}) - o(n)$ Gap

We review the proof of the $\omega(n^{1/2}) - o(n)$ gap by Balliu et al. [3]. Given an $o(n)$-round randomized or deterministic LOCAL algorithm $\mathcal{A}$ for the given LCL problem $\mathcal{P}$, the goal is to design a new randomized or deterministic LOCAL algorithm $\mathcal{A}'$ with round complexity $O(\sqrt{n})$. Within this section, we only apply the pumping lemma on unlabeled graphs, but we will see that when we extend the proof to other gaps, we need to deal with partially labeled graphs.

The skeleton tree. Let the tree $G = (V, E)$ be the underlying network. Let $\tau = \Theta(\sqrt{n})$ be a threshold to be determined. Define the *skeleton tree* G_{skel} as the result of iteratively removing all leaf vertices of G for τ iterations. Specifically, start with $G_0 = G$, and let G_i be the result of removing all leaf vertices of G_{i-1} for each $1 \leq i \leq \tau$, and then we have $G_{\mathsf{skel}} = G_\tau$.

If G_{skel} is empty, then we are already done, since this implies that the diameter of G is $O(\tau) = O(\sqrt{n})$, so $\mathcal{P}$ can be solved trivially in $O(\sqrt{n})$ rounds. In subsequent discussion we assume that G_{skel} is not empty. We will identify a set of disjoint paths $\mathcal{P}$ of G_{skel} meeting the following conditions.

1. Each $P = (v_1, v_2, \ldots, v_x) \in \mathcal{P}$ satisfies the following requirements.
   a. $x \in [\ell_{\text{pump}}, 2\ell_{\text{pump}}]$.
   b. Each v_i is of degree-2 in G_{skel}.
2. Let G' be the subgraph of G_{skel} resulting from removing all paths in $\mathcal{P}$. Let $\mathcal{S}$ denote the set of connected components in G'. Then each connected component $S \in \mathcal{S}$ in G_{skel} has diameter $O(\sqrt{n})$.

The proof of the existence of $\mathcal{P}$ can be found in [3]. We will also provide a proof in Section 4. In this section we only need to use the fact that the skeleton tree G_{skel} and the set of paths $\mathcal{P}$ can be computed in $O(\sqrt{n})$ rounds on G.

Since G_{skel} is constructed by iteratively removing all leaf vertices of G for τ iterations, each vertex v in $G \setminus G_{\mathsf{skel}}$ is reachable to a *unique* vertex u in G_{skel} via the vertices $G \setminus G_{\mathsf{skel}}$.

For any vertex subset U in G_{skel}, we define $U^* \supseteq U$ as the set of vertices in G resulting from adding to U all vertices in $G \setminus G_{\mathsf{skel}}$ reachable to U via the vertices in $G \setminus G_{\mathsf{skel}}$. A consequence of Condition 2 is that the diameter of S^* is $O(\tau + \sqrt{n}) = O(\sqrt{n})$, for each $S \in \mathcal{S}$.

The virtual tree. Consider the *virtual tree* G_{virt} defined as the result of applying the pumping lemma on P^* for each $P = (v_1, v_2, \ldots, v_x) \in \mathcal{P}$ to the graph G. The definition of P^* is in the paragraph above. Here P^* is seen as a bipolar tree with the poles $s = v_1$ and $t = v_x$. Specifically, the pumping lemma allows us to replace each bipolar tree $P^* = (T_1, T_2, \ldots, T_k)$ is by some other bipolar tree $P' = (T'_1, T'_2, \ldots, T'_{x'})$ such that $\mathsf{Type}(P') = \mathsf{Type}(P^*)$, and $x' \in [w, w + \ell_{\text{pump}}]$, where w is some very large number to be determined.

The $O(\sqrt{n})$-round algorithm $\mathcal{A}'$. We are ready to describe our $O(\sqrt{n})$-round algorithm $\mathcal{A}'$. The first step of the algorithm is to compute the skeleton tree G_{skel} and the set of paths $\mathcal{P}$ in $O(\sqrt{n})$ rounds. After that, we can simulate the virtual tree G_{virt} by having the vertices in each $P \in \mathcal{P}$ simulate the virtual bipolar tree P' resulting from the pumping lemma. We compute a legal labeling $\mathcal{L}_{\mathsf{virt}}$ of G_{virt} by a simulation of $\mathcal{A}$ on G_{virt}. We will later see that the simulation can also be done in $O(\sqrt{n})$ rounds. Finally, we will show that the labeling $\mathcal{L}_{\mathsf{virt}}$ can be transformed into a legal labeling $\mathcal{L}$ of G using another $O(\sqrt{n})$ rounds.

Simulation of $\mathcal{A}$ on the virtual tree. It is clear that the number of vertices in G_{virt} can be upper bounded by $O(n^2 w)$, since $|\mathcal{P}| \leq n$ and the number of vertices in each P' is $O(nw)$. We simulate the given algorithm $\mathcal{A}$ on the virtual tree G_{virt} assuming that the number of vertices is $n' = O(n^2 w)$.

Since the round complexity of $\mathcal{A}$ on an n'-vertex graph is $o(n')$, by selecting w as a sufficiently large number depending on n, the round complexity of $\mathcal{A}$ can be made much smaller than $0.1w$. Therefore, to simulate $\mathcal{A}$ on G_{virt}, each vertex v in G_{virt} only needs to gather all information within radius $0.1w$ to v. We make the following observations.

- For each $S \in \mathcal{S}$, the subgraph S^* has diameter $O(\sqrt{n})$.

- For each $P \in \mathcal{P}$, the number of vertices in the core path of the bipolar subtree P' is within $[w, w + \ell_{\text{pump}}]$.

By these facts, it is straightforward to see that each vertex v in G_{virt} is able to gather all information within radius $0.1w$ to v in $O(\sqrt{n})$ rounds of communication in the underlying network G. For example, if $v \in S^*$ for some $S \in \mathcal{S}$, then v only need to learn the following.

- The subgraph induced by the set S^*.
- The virtual bipolar tree P', for each path $P \in \mathcal{P}$ adjacent to S.

Remember that P' can be computed from P^*. Since the diameter of S^* (for each $S \in \mathcal{S}$) and the diameter of P^* (for each $P \in \mathcal{P}$) are $O(\sqrt{n})$, this information gathering can be done in $O(\sqrt{n})$ rounds in G.

Computing a legal labeling of G. Suppose we have computed a legal labeling $\mathcal{L}_{\mathsf{virt}}$ of G_{virt}. We show how to use this legal labeling $\mathcal{L}_{\mathsf{virt}}$ to obtain a legal labeling $\mathcal{L}$ of G in $O(\sqrt{n})$ rounds. For each $S \in \mathcal{S}$, the labeling of the vertices in S^* is unchanged, i.e., $\mathcal{L}(v) = \mathcal{L}_{\mathsf{virt}}(v)$. For each path $P \in \mathcal{P}$, the $\mathcal{L}$-labels of the vertices in P^* are computed as follows.

Remember that G_{virt} is the result of replacing P^* with P', for each $P \in \mathcal{P}$, and the two bipolar trees P' and P^* have the same type. In view of the property of $\overset{\star}{\sim}$ described in Section 2.2, there exists a labeling $\mathcal{L}'$ of P^* such that if we replace the bipolar subtree P' (labeled with $\mathcal{L}_{\mathsf{virt}}$) by the bipolar subtree P^* (labeled with $\mathcal{L}'$), the legality of the labeling of the underlying graph is maintained. Moreover, such a labeling $\mathcal{L}'$ of P^* can be computed from the labeling $\mathcal{L}_{\mathsf{virt}}$ restricted to P', without using any information outside of P'. Thus, we can carry out this procedure, in parallel for each $P \in \mathcal{P}$, and this takes $O(\sqrt{n})$ rounds, since the diameter of P^* is at most $2\tau + 2\ell_{\text{pump}} - 1 = O(\sqrt{n})$, for each $P \in \mathcal{P}$. After that, we obtain a desired legal labeling $\mathcal{L}$ of G.

4 A Generalized Tree Decomposition

Miller and Reif [18] considered the following decomposition algorithm. Start with a tree $G = (V, E)$; remove the vertices in V by repeatedly doing the following two operations alternately: Rake (removing all leaf vertices) and Compress (removing all degree-2 vertices). It is known that $O(\log n)$ iterations suffice to remove all vertices in the tree [18]. Variants of this decomposition have turned out to be useful in the design of LOCAL algorithms [10, 12].

In this section, we consider a generalized version of this decomposition, which allows us to show the existence of $\mathcal{P}$ needed in Section 3, and to extend the proof idea in Section 3 to other gaps.

We start with a formal definition of our decomposition, which is parameterized by two integers $\ell \geq 1$ and $\gamma \geq 1$, and it decomposes the vertices in the tree G into

$$V = V_1^{\mathsf{R}} \cup V_1^{\mathsf{C}} \cup V_2^{\mathsf{R}} \cup V_2^{\mathsf{C}} \cup V_3^{\mathsf{R}} \cup V_3^{\mathsf{C}} \cup \cdots .$$

Let L denote the highest number i such that $V_i^{\mathsf{C}} \cup V_{i+1}^{\mathsf{R}} \cup V_{i+1}^{\mathsf{C}} \cup \cdots$ is empty. We define G_i^{C} as the subgraph induced by the vertices $\left(\bigcup_{j=i+1}^{L} V_j^{\mathsf{R}}\right) \cup \left(\bigcup_{j=i}^{L-1} V_j^{\mathsf{C}}\right)$, which is the set of all vertices that are in V_i^{C} or higher layers. Similarly, we define G_i^{R} as the subgraph induced by the vertices $\left(\bigcup_{j=i}^{L} V_j^{\mathsf{R}}\right) \cup \left(\bigcup_{j=i}^{L-1} V_j^{\mathsf{C}}\right)$.

We require the sets V_i^{R} and V_i^{C} to satisfy some requirements. Each connected component of the subgraph induced by V_i^{R} must be a rooted tree with height at most $\gamma - 1$, and only the root can possibly have neighbors in $V_i^{\mathsf{C}} \cup V_{i+1}^{\mathsf{R}} \cup V_{i+1}^{\mathsf{C}} \cup \cdots$. Each connected component of the subgraph induced by V_i^{C} must be a path with $x \in [\ell, 2\ell]$ vertices, and only the endpoints can

possibly have neighbors in $V^{\mathsf{R}}_{i+1} \cup V^{\mathsf{C}}_{i+1} \cup V^{\mathsf{R}}_{i+2} \cup \cdots$. See Figure 3 for an illustration, where each triangle represents a rooted tree. The precise requirements are as follows.

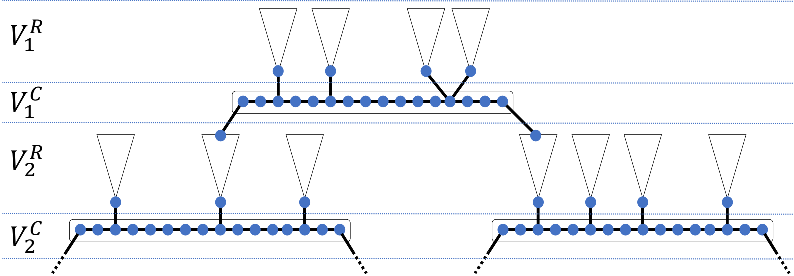


Figure 3 Top layers in a generalized tree decomposition.

Requirements for V^{R}_i. Let S be a connected component of the subgraph induced by V^{R}_i. Then there is a root vertex $z \in S$ such that the following conditions are met.

- z has at most one neighbor in G^{C}_i, and each $v \in S \setminus \{z\}$ has no neighbor in G^{C}_i.
- Each $v \in S \setminus \{z\}$ satisfies $\mathrm{dist}(v, z) \le \gamma - 1$.

Note that for the special case of $\gamma = 1$, the set V^{R}_i is an independent set.

Requirements for V^{C}_i. Let S be a connected component of the subgraph induced by V^{C}_i. Then S is a path $(u_1, u_2, \ldots, u_x)$ with $x \in [\ell, 2\ell]$ such that the following is true for each $u_j \in S$.

- For the case $1 < j < x$ (i.e., u_j is an intermediate vertex), u_j has no neighbor in G^{R}_{i+1}.
- Consider the case $j = 1$ or $j = x$ (i.e., u_j is an endpoint). If $x \ge 2$, then u_j has exactly one neighbor in G^{R}_{i+1}. If $x = 1$, then u_j has exactly two neighbors in G^{R}_{i+1}.

A decomposition $V = V^{\mathsf{R}}_1 \cup V^{\mathsf{C}}_1 \cup V^{\mathsf{R}}_2 \cup V^{\mathsf{C}}_2 \cup V^{\mathsf{R}}_3 \cup V^{\mathsf{C}}_3 \cup \cdots$ satisfying the above requirements is called a *(γ, ℓ)-decomposition.* We will see in Lemma 7 that for any positive integers $k = O(1)$ and $\ell = O(1)$, and for any

$$\gamma \ge n^{1/k} (\ell/2)^{1-1/k},$$

an (γ, ℓ)-decomposition with $L = k$ can be computed in $O(n^{1/k})$ rounds deterministically.

4.1 The Decomposition Algorithm

Our algorithm constructing the above decomposition uses the following modified Rake and Compress operations defined in [12]. Here U is a subset of V representing the set of vertices that are not yet removed.

Rake: Each $v \in U$ removes itself if one of the following conditions is met.

1. $\deg_U(v) = 0$.
2. $\deg_U(v) = 1$ and the unique neighbor u of v in U has $\deg_U(u) > 1$.
3. $\deg_U(v) = 1$ and the unique neighbor u of v in U has $\deg_U(u) = 1$ and $\mathrm{ID}(v) > \mathrm{ID}(u)$.

Compress: Each $v \in U$ removes itself if v belongs to a path P such that $|V(P)| \ge \ell$ and $\deg_U(u) = 2$ for each $u \in V(P)$.

The purpose of Condition 3 in the Rake operation is to break tie for the special case where v is in a component of U that is a length-1 path. This is to ensure that we remove an independent set of vertices in a Rake operation.

▶ **Definition 3** ([12]). *Let P be a path. A subset $I \subset V(P)$ is called an (α, β)-independent set if the following conditions are met: (i) I is an independent set that does not contain either endpoint of P, and (ii) each connected component of the subgraph induced by $V(P) \setminus I$ has at least α vertices and at most β vertices, unless $|V(P)| < \alpha$, in which case $I = \emptyset$.*

It is a folklore that an $(\ell, 2\ell)$-independent set of a path graph can be computed in $O(\log^* n)$ rounds deterministically when $\ell = O(1)$ [1,12,17].

The algorithm. The decomposition algorithm begins with $U = V(G)$ and $i = 1$. In iteration i, we do the following.

1. Do γ Rake operations.
2. Do one Compress operation.
3. Update the iteration number $i \leftarrow i + 1$.

Repeatedly do this until $U = \emptyset$, and then we proceed to the following post-processing step.

The post-processing step. Let R_i (resp., C_i) be the set of vertices removed during a Rake (resp., Compress) operation in the ith iteration. For each path P that is a connected component of the subgraph induced by C_i, Find an $(\ell, 2\ell)$-independent set I_P of P. Define C_i^* as the subset of C_i that is the union of I_P for each P that is a connected component of the subgraph induced by C_i. Let L be the highest number i such that $R_i \cup C_{i-1} \neq \emptyset$. Then a partition $V = \left(\bigcup_{i=1}^{L} V_i^{\mathsf{R}}\right) \cup \left(\bigcup_{i=1}^{L-1} V_i^{\mathsf{C}}\right)$ is defined by setting $V_i^{\mathsf{R}} = R_i \cup C_{i-1}^*$ and $V_i^{\mathsf{C}} = C_i \setminus C_i^*$. What we have done in the post-processing step is promoting each vertex in the independent set I_P to the next layer, and this ensures that the requirement on the size of paths for V_i^{C} is met.

Analysis. We analyze the decomposition $V = \left(\bigcup_{i=1}^{L} V_i^{\mathsf{R}}\right) \cup \left(\bigcup_{i=1}^{L-1} V_i^{\mathsf{C}}\right)$ produced using the above algorithm. The proofs of the following two lemmas follow immediately from the description of the decomposition algorithm.

▶ **Lemma 4** (Properties of V_i^{R}). *Let S be a connected component of the subgraph induced by V_i^{R}. Then there is a root vertex $z \in S$ such that the following conditions are met.*

- *z has at most one neighbor in G_i^{C}, and each $v \in S \setminus \{z\}$ has no neighbor in G_i^{C}.*
- *Each $v \in S \setminus \{z\}$ satisfies $\mathrm{dist}(v, z) \leq \gamma - 1$.*

Proof. The first case is when S contains a vertex u that is in I_P for some P during the post-processing step, we must have $S = \{u\}$, and u has no neighbor in G_i^{C}. In this case, setting $z = u$ works.

The second case is when $S \subseteq R_i$. We select $z \in S$ as the last vertex removed from U during the decomposition algorithm, among all vertices in S. Since we do γ Rake operations in each iteration, each $v \in S \setminus \{z\}$ satisfies $\mathrm{dist}(v, z) \leq \gamma - 1$. It is straightforward to see that z is the only vertex in S that may have a neighbor in G_i^{C}; and z can have at most one such neighbor. ◀

▶ **Lemma 5** (Properties of V_i^{C}). *Let S be a connected component of the subgraph induced by V_i^{C}. Then S is a path $(u_1, u_2, \ldots, u_x)$ with $x \in [\ell, 2\ell]$ such that the following is true for each $u_j \in S$.*

- *For the case $1 < j < x$ (i.e., u_j is an intermediate vertex), u_j has no neighbor in G_{i+1}^{R}.*
- *Consider the case $j = 1$ or $j = x$ (i.e., u_j is an endpoint). If $x \geq 2$, then u_j has exactly one neighbor in G_{i+1}^{R}. If $x = 1$, then u_j has exactly two neighbors in G_{i+1}^{R}.*

Proof. In view of the post-processing step and the definition of an (α, β)-independent set, S is a path $(u_1, u_2, \ldots, u_x)$ with $x \in [\ell, 2\ell]$. It is straightforward to verify that the conditions specified in the lemma are met. ◀

Next, we analyze the round complexity of the decomposition algorithm and the number L in the decomposition. We remark that the case of $\gamma = 1$ is considered and analyzed in [12].

▶ **Lemma 6** ([12]). *Suppose $\gamma = 1$ and $\ell \geq 1$ is a constant. An (γ, ℓ)-decomposition with $L = O(\log n)$ of a tree G can be computed in $O(\log n)$ rounds deterministically.*

In this paper, we are only interested in the case of $\gamma \gg 1$.

▶ **Lemma 7.** *Suppose $\gamma \geq n^{1/k}(\ell/2)^{1-1/k}$, for some positive integers $k = O(1)$ and $\ell = O(1)$. An (γ, ℓ)-decomposition with $L = k$ of a tree G can be computed in $O(n^{1/k})$ rounds deterministically.*

Proof. Consider an arbitrary vertex $v \in V$, and root the tree G at v. Define S_i as the connected component containing v in the subgraph induced by the set U at the beginning of the ith iteration. Define S'_i as the connected component containing v in the subgraph induced by the set U at the beginning of the Compress operation during the ith iteration. Note that $S_1 = V$. To prove the lemma, it suffices to show that $S'_k = \emptyset$.

Let A be the number of degree-2 vertices in S'_i that are not removed during the ith Compress. Let B be the number of vertices in S'_i whose degree is not 2 at the beginning of the ith Compress. Note that $|S_{i+1}| = A + B$.

We observe that $A \leq (\ell - 1)(B - 1)$, as the degree-2 vertices in S'_i that are not removed during the ith Compress form connected components of at most $\ell - 1$ vertices. Specifically, consider the tree T resulting from contracting each degree-2 vertex in S'_i. The number of vertices in T equals B, and the number of edges in T is at least $A/(\ell - 1)$. Hence $A/(\ell - 1) \leq (B - 1)$.

We also observe that the number of degree-1 vertices in S'_i at the beginning of the ith Compress is at least $B/2$. As each degree-1 vertex of S'_i must be adjacent to a connected component of $S_i \setminus S'_i$ of size at least γ, we have $|B|/2 < |S_i|/\gamma$. Therefore,

$$|S_{i+1}| = A + B < \ell B < \frac{\ell}{2\gamma} \cdot |S_i|.$$

In order to have $S'_k = \emptyset$, it suffices that $|S_k| \leq \gamma$. Indeed, we have

$$|S_k| \leq \left(\frac{\ell}{2\gamma}\right)^{(k-1)} \cdot n \leq \gamma.$$

For the round complexity, the main part of the algorithm costs $O((\gamma + \ell)k) = O(\gamma) = O(n^{1/k})$ rounds. The post-processing step costs $O(\log^* n)$ rounds, as $\ell = O(1)$. ◀

The set of paths $\mathcal{P}$. We revisit the proof in Section 3 and prove that the required set of paths $\mathcal{P}$ can be computed in $O(\sqrt{n})$ rounds. We run our algorithm for constructing a (γ, ℓ)-decomposition with the parameters $\gamma = \tau = n^{1/2}(\ell/2)^{1/2} = \Theta(\sqrt{n})$ and $\ell = \ell_{\text{pump}} = \Theta(1)$. Here τ is the parameter in the definition of the skeleton tree G_{skel} in Section 3. Remember that G_{skel} is the result of iteratively removing all leaf vertices of G for τ iterations.

By Lemma 7, our (γ, ℓ)-decomposition satisfies $L = 2$, and it decomposes V into three sets V_1^{R}, V_1^{C}, and V_2^{R}, and the decomposition can be computed in $O(\sqrt{n})$ rounds. It is clear from the description of the algorithm that $G_{\mathsf{skel}} = G_1^{\mathsf{C}}$ is exactly the subgraph induced by $V_1^{\mathsf{C}} \cup V_2^{\mathsf{R}}$. Selecting $\mathcal{P}$ as the set of all connected components of V_1^{C} satisfies all the requirements of $\mathcal{P}$ stated in Section 3.

5 Extension to Other Gaps

In this section, we prove Theorem 1 for the case of deterministic algorithms by extending the proof of the $\omega(n^{1/2}) - o(n)$ gap by Balliu et al. [3] described in Section 3. The complete proof of Theorem 1 is left to the full version of the paper.

5.1 Proof Idea

The main idea of the proof of Theorem 1 is as follows. Let k be any positive constant, To prove the existence of the gap $\omega(n^{1/k}) - o(n^{1/(k-1)})$, for any given $o(n^{1/(k-1)})$-round deterministic algorithm $\mathcal{A}$ for a given LCL problem $\mathcal{P}$, we need to be able to design a new deterministic algorithm $\mathcal{A}'$ with round complexity $O(n^{1/k})$.

We compute a (γ, ℓ)-decomposition, with $\gamma = \Theta(n^{1/k})$ and $\ell \geq \ell_{\text{pump}}$ to decompose V into the subsets $V_1^{\mathsf{R}}, V_1^{\mathsf{C}}, V_2^{\mathsf{R}}, \ldots, V_{k-1}^{\mathsf{R}}, V_{k-1}^{\mathsf{C}}, V_k^{\mathsf{R}}$, and then apply the pumping lemma to extend each path in $V_1^{\mathsf{C}}, V_2^{\mathsf{C}} \ldots, V_{k-1}^{\mathsf{C}}$ to a path of length within $[w, w + \ell_{\text{pump}}]$, in order to produce a virtual tree with $O(w^{k-1})$ vertices, omitting the dependency on n. By Lemma 7, such a decomposition can be computed in $O(n^{1/k})$ rounds.

If we select w to be sufficiently large, the execution of a given $o(n^{1/(k-1)})$-round algorithm $\mathcal{A}$ takes less than $0.1w$ rounds on the virtual tree. As each connected component induced by V_i^{R} is a rooted tree with diameter $O(n^{1/k})$, the simulation of $\mathcal{A}$ can be done in $O(n^{1/k})$ rounds in the underlying network G. Hence we obtain an $O(n^{1/k})$-round algorithm $\mathcal{A}'$ for the same problem.

This approach does not work immediately, as we will encounter some issues described below, but these issues can be overcome using the graph operations defined in [12].

An issue in pumping bipolar subtrees. The reason that we can apply the pumping lemma for V_1^{C} in Section 3 is that each connected component P of V_1^{C} is naturally associated with a bipolar tree P^*. We do not have this property for the connected components of V_i^{C} for $i > 1$, since each vertex $v \in V_1^{\mathsf{R}} \cup V_1^{\mathsf{C}} \cup \cdots \cup V_i^{\mathsf{R}} = V(G) \setminus V(G_i^{\mathsf{C}})$ might be reachable to more than one connected component of V_i^{C} via the vertices in $V(G) \setminus V(G_i^{\mathsf{C}})$. See Figure 3.

Let us recall the virtual tree construction in Section 3. Define G' as the graph resulting from pumping the paths in V_1^{C}. Formally, for each connected component P of V_1^{C}, replace P^* by a much longer bipolar tree P' with $\mathsf{Type}(P^*) = \mathsf{Type}(P')$, where P^* is the bipolar subtree of G induced by the vertices in P and all vertices in V_1^{R} that are reachable to a vertex in P via the vertices in V_1^{R}. Note that G' is the same as the virtual tree G_{virt} in Section 3, if we use $G_{\mathsf{skel}} = G_1^{\mathsf{C}}$ and let $\mathcal{P}$ be the set of connected components of V_1^{C}.

As we are in the $k > 2$ case, we would like to also pump the paths in V_2^{C} in this graph G' in a way similar to the case of V_1^{C}. As discussed above, a difference between V_1^{C} and V_2^{C} is that it is possible that a vertex $v \in V(G') \setminus V(G_2^{\mathsf{C}})$ is reachable to multiple connected components in V_2^{C} via the vertices in $V(G') \setminus V(G_2^{\mathsf{C}})$, so we are unable to associate a bipolar tree P^* to each connected component P of V_2^{C}.

Recall that in our high-level proof idea we will ultimately simulate an algorithm $\mathcal{A}$ on a virtual tree, and the runtime of $\mathcal{A}$ will be less than $0.1w$. Again consider the graph G' and one of its bipolar subtree P' resulting from pumping P^* for some connected component P of V_1^{C}. The virtual bipolar tree P' separates the graph G' into two parts, and the vertices in one part does not need to communicate with the vertices in the other part in the simulation of $\mathcal{A}$, as its runtime is less than $0.1w$. Recall that the core path of P' has at least w vertices.

Motivated by the above discussion, we consider the graph G'' defined as the result of applying the following operation on G' for each virtual bipolar tree P'. Let u and v be the two vertices in $V(G') \setminus V(P')$ adjacent to the two poles s and t of P' via the edges $\{u, s\}$ and $\{v, t\}$. We *duplicate* P' into two identical bipolar subtrees, one is attached to u via $\{u, s\}$, the other is attached to v via $\{v, t\}$. Note that this is the Duplicate-Cut operation defined in [12]. See Figure 4 for an illustration.

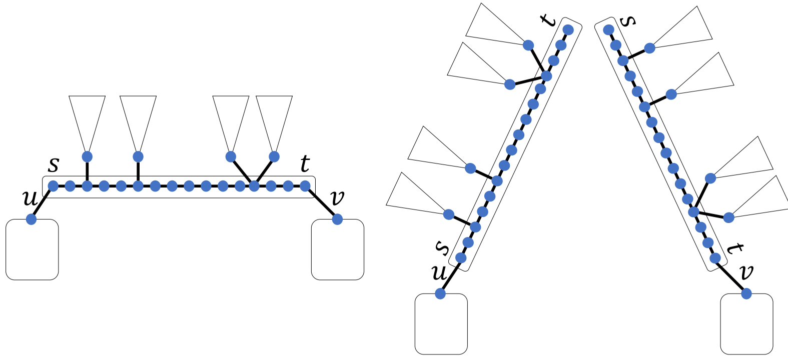


Figure 4 The Duplicate-Cut operation.

Let P be a connected component of V_2^{C} in the graph G''. With respect to G'', we are able to define P^* in the same way as the case of V_1^{C}. Specifically, we define P^* as the bipolar subtree of G'' induced by the vertices in P and all vertices in $V(G'') \setminus V(G_2^{\mathsf{C}})$ that are reachable to a vertex in P via the vertices in $V(G'') \setminus V(G_2^{\mathsf{C}})$. Using this approach recursively, we can pump the paths of all layers $V_1^{\mathsf{C}}, V_2^{\mathsf{C}}, \ldots, V_{k-1}^{\mathsf{C}}$.

An issue caused by duplicating bipolar trees. The duplication of bipolar subtrees in Duplicate-Cut also causes an issue. Consider the graphs G, G', and G'' defined above. As discussed in Section 3, given a legal labeling of G', we can obtain a legal labeling of G using a property of $\overset{\star}{\sim}$ and the fact that pumping does not alter the type of a bipolar tree. However, when we try to obtain a legal labeling $\mathcal{L}'$ of G' from a given legal labeling $\mathcal{L}''$ of G'', we encounter an issue that the two copies of a bipolar subtree P' resulting from applying Duplicate-Cut in G' might be labeled differently in $\mathcal{L}''$.

To resolve this issue, before the duplication of P' in the construction of G'' from G', we let some vertices near the middle of P' to first commit to a certain labeling. Such a labeling is computed by simulating the given $o(n^{1/(k-1)})$-round algorithm $\mathcal{A}$, pretending that the number of vertices is $O(w^{k-1})$, omitting the dependence on n. We can assume that the runtime of $\mathcal{A}$ on P' is at most $0.1w$ by selecting w to be sufficiently large.

Specifically, let P^* be a bipolar tree that we would like to apply the pumping lemma. We give a different way of constructing P' from P^*. We write P^* as a string of subtrees $(\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_x)$. Let $(v_1, v_2, \ldots, v_x)$ be the core path of P^* and $e = \{v_{\lfloor x/2 \rfloor}, v_{\lfloor x/2 \rfloor + 1}\}$ be the middle edge of the core path. Consider the decomposition $P^* = \mathcal{X} \circ \mathcal{Y} \circ \mathcal{Z}$, where $\mathcal{Y} = (\mathcal{T}_{\lfloor x/2 \rfloor - r + 1}, \ldots, \mathcal{T}_{\lfloor x/2 \rfloor + r})$ is the middle part. We apply the pumping lemma on $\mathcal{X}$ and $\mathcal{Z}$ to extend them to longer bipolar trees whose whose size of core path is within $[w, w + \ell_{\text{pump}}]$, and then we assign output labels to the vertices in $N^{r-1}(e) = N^{r-1}(v_{\lfloor x/2 \rfloor}) \cup N^{r-1}(v_{\lfloor x/2 \rfloor + 1})$ by simulating the algorithm $\mathcal{A}$, whose runtime is less than $0.1w$. The resulting partially labeled bipolar tree is P'. Note that this construction of P' from P^* is the same as the one in [12] using the operations Label and Extend. In this paper, we call this operation Label-Extend. See Figure 5 for an illustration.

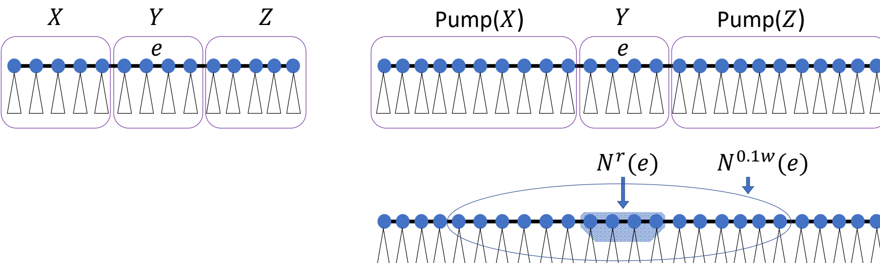


Figure 5 The Label-Extend operation.

We briefly explain why doing this labeling of middle vertices resolves the issue. Suppose G_a is the result of apply Duplicate-Cut to some bipolar subtree P' in G_b, where this bipolar tree P' is constructed as above and its middle vertices have been assigned output labels. In a given legal labeling of G_a, the two copies of P' might be labeled differently, but their middle vertices must be labeled the same. We decompose $P' = P_s \circ P_t$ into two parts by cutting along the middle edge e. The pole s is in P_s, and the other pole t is in P_t. We name the two copies of P' in G_a by P'_s and P'_t based on the poles they use to connect to the rest of the graph. To obtain a legal labeling of G_b from a given legal labeling of G_a, we simply label P_s by adapting the labeling of P'_s in G_a, and label P_t by adapting the labeling of P'_t in G_a. The legality of the resulting labeling of G_b is easy to verify.

See Figure 6 for an example. The top-left figure illustrates the bipolar subtree P' in G_b, where its middle vertices have been assigned output labels. The top-right figure illustrates the graph G_a, which results from applying Duplicate-Cut to P' in G_b. The down-right figure illustrates the given legal labeling of G_a. The down-left figure shows the legal labeling of G_b obtaining from the given legal labeling of G_a.

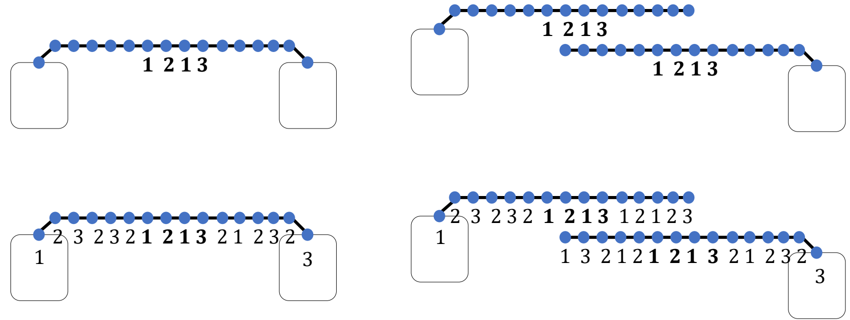


Figure 6 Obtaining a legal labeling.

5.2 A Sequence of Virtual Graphs

The approach discussed above naturally leads to a sequence of partially labeled virtual graphs $\mathcal{R}_1^{\mathsf{R}}, \mathcal{R}_1^{\mathsf{C}}, \mathcal{R}_2^{\mathsf{R}}, \mathcal{R}_2^{\mathsf{C}}, \ldots, \mathcal{R}_k^{\mathsf{R}}$. Each virtual graph has a *real* and an *imaginary* part. Each real vertex corresponds to a vertex in the underlying network G. The graph $\mathcal{R}_i^{\mathsf{R}}$ will have G_i^{R} as its real part, and the graph $\mathcal{R}_i^{\mathsf{C}}$ will have G_i^{C} as its real part. The imaginary part of these graphs are subtrees attached to the real vertices. In the actual distributed implementation, the simulation of the imaginary subtrees attached to a real vertex v are handled by v in the underlying network G.

Construction of $\mathcal{R}_1^{\mathsf{R}}$. The graph $\mathcal{R}_1^{\mathsf{R}}$ is unlabeled and it equals the underlying network G.

Construction of $\mathcal{R}_1^{\mathsf{C}}$. The graph $\mathcal{R}_1^{\mathsf{C}}$ is almost identical to $G = \mathcal{R}_1^{\mathsf{R}}$. In $\mathcal{R}_1^{\mathsf{C}}$, only the vertices in G_1^{C} are real. For each connected component S of $V_1^{\mathsf{R}} = \mathcal{R}_1^{\mathsf{R}} \setminus G_1^{\mathsf{C}}$, there is at most one vertex $v \in G_1^{\mathsf{C}}$ that is adjacent to S. If such a vertex v exists, then S becomes an imaginary subtree stored in the real vertex v. Otherwise, S is not included in $\mathcal{R}_1^{\mathsf{C}}$.

Construction of $\mathcal{R}_2^{\mathsf{R}}$. The graph $\mathcal{R}_2^{\mathsf{R}}$ is the graph G' in Section 5.1. Formally, the graph $\mathcal{R}_2^{\mathsf{R}}$ is constructed by applying the following operation to each connected component $P = (v_1, v_2, \ldots, v_x)$ of V_1^{C} in $\mathcal{R}_1^{\mathsf{C}}$. Let $P^* = (\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_x)$ be the bipolar subtree induced by P and the imaginary subtrees therein. Replace P^* by the partially labeled bipolar tree P' which is the result of applying Label-Extend to P^*, and then apply Duplicate-Cut to P^*.

Construction of $\mathcal{R}_2^{\mathsf{C}}$. The graph $\mathcal{R}_2^{\mathsf{C}}$ is almost identical to $\mathcal{R}_2^{\mathsf{R}}$. In $\mathcal{R}_2^{\mathsf{C}}$, only the vertices in G_2^{C} are real. Due to the Duplicate-Cut operation, for each connected component S of $\mathcal{R}_2^{\mathsf{R}} \setminus G_2^{\mathsf{C}}$, there is at most one vertex $v \in G_2^{\mathsf{C}}$ that is adjacent to S. If such a vertex v exists, then S becomes an imaginary subtree stored in the real vertex v. Otherwise S is not included in $\mathcal{R}_2^{\mathsf{C}}$.

Construction of the other graphs. The rest of the partially labeled graphs $\mathcal{R}_3^{\mathsf{R}}$, $\mathcal{R}_3^{\mathsf{C}}$, $\mathcal{R}_4^{\mathsf{R}}$, $\mathcal{R}_4^{\mathsf{C}}$, ..., $\mathcal{R}_k^{\mathsf{R}}$ are constructed analogously. In the end, $\mathcal{R}_k^{\mathsf{R}}$ is a virtual graph with $O(w^{k-1})$ vertices, omitting the dependence on n. It is straightforward to see that the sequence of virtual graphs $\mathcal{R}_1^{\mathsf{R}}, \mathcal{R}_1^{\mathsf{C}}, \mathcal{R}_2^{\mathsf{R}}, \mathcal{R}_2^{\mathsf{C}}, \ldots, \mathcal{R}_k^{\mathsf{R}}$ can be constructed in $O(n^{1/k})$ rounds.

Completing the labeling. Recall that the partial labelings of $\mathcal{R}_1^{\mathsf{R}}, \mathcal{R}_1^{\mathsf{C}}, \mathcal{R}_2^{\mathsf{R}}, \mathcal{R}_2^{\mathsf{C}}, \ldots, \mathcal{R}_k^{\mathsf{R}}$ are computed using the operation Label-Extend, which is based on simulating $\mathcal{A}$ while assuming that the number of vertices is $n' = O(w^{k-1})$, omitting the dependence on n. By the correctness of $\mathcal{A}$, each of these partial labelings can be completed into a complete legal labeling. Since each connected component of the real part of $\mathcal{R}_k^{\mathsf{R}}$ has at most $O(n^{1/k})$ vertices, a complete legal labeling of $\mathcal{R}_k^{\mathsf{R}}$ can be found in $O(n^{1/k})$ rounds by a brute-force information gathering. Once we have a complete labeling of $\mathcal{R}_k^{\mathsf{R}}$, we can start from this complete labeling to obtain a complete legal labeling for $\mathcal{R}_{k-1}^{\mathsf{C}}, \mathcal{R}_{k-1}^{\mathsf{R}}, \mathcal{R}_{k-2}^{\mathsf{C}}, \ldots, \mathcal{R}_1^{\mathsf{R}} = G$ in $O(n^{1/k})$ rounds in view of the discussion in Section 5.1, as these graphs are constructed by applying Label-Extend and then applying Duplicate-Cut to the bipolar trees resulting from Label-Extend.

The round complexity for finding a legal labeling of $G = \mathcal{R}_1^{\mathsf{R}}$ using this approach is $O(n^{1/k})$ because the size of each connected component of V_i^{R} is $O(n^{1/k})$.

Hence we have the $\omega(n^{1/k}) - o(n^{1/(k-1)})$ gap for LCL problems on bounded-degree trees for the case of deterministic algorithms. That is, given a deterministic $o(n^{1/(k-1)})$-round algorithm $\mathcal{A}$ for $\mathcal{P}$, we can construct another deterministic $O(n^{1/k})$-round algorithm $\mathcal{A}'$.

A note about unique identifiers. A subtle issue about the simulation of $\mathcal{A}$ is that the simulation needs distinct identifiers. Specifically, to guarantee the correctness of a deterministic τ-round algorithm for an LCL problem with locality radius r, it suffices that any two vertices within distance $2\tau + 2r$ have distinct identifiers [11].

We only simulate $\mathcal{A}$ when we apply Label-Extend. When we do the simulation of $\mathcal{A}$, we can locally generate distinct identifiers of length $O(\log n')$ for all vertices in $N^{0.1w+r}(e)$, where e is the middle edge of the core path of the bipolar tree on which we run $\mathcal{A}$, and $n' = O(w^{k-1})$, omitting the dependence on n. This partial ID assignment satisfies the requirement that, for any two vertices u and v that are assigned identifiers and are within distance $2 \cdot 0.1w + 2r$, their identifiers are distinct.

References

1. Alkida Balliu, Sebastian Brandt, Yi-Jun Chang, Dennis Olivetti, Mikaël Rabie, and Jukka Suomela. The distributed complexity of locally checkable problems on paths is decidable. In *Proceedings of the 38th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 262–271. ACM Press, 2019.
2. Alkida Balliu, Sebastian Brandt, Yuval Efron, Juho Hirvonen, Yannic Maus, Dennis Olivetti, and Jukka Suomela. Classification of distributed binary labeling problems. In *Proceedings of the 34th International Symposium on Distributed Computing (DISC)*, 2020.
3. Alkida Balliu, Sebastian Brandt, Dennis Olivetti, and Jukka Suomela. Almost global problems in the LOCAL model. In *Proceedings of the 32nd International Symposium on Distributed Computing (DISC)*, 2018.
4. Alkida Balliu, Sebastian Brandt, Dennis Olivetti, and Jukka Suomela. How much does randomness help with locally checkable problems? In *Proceedings of the 39th Symposium on Principles of Distributed Computing (PODC)*, pages 299–308. ACM, 2020.
5. Alkida Balliu, Juho Hirvonen, Janne H. Korhonen, Tuomo Lempiäinen, Dennis Olivetti, and Jukka Suomela. New classes of distributed time complexity. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 1307–1318. ACM, 2018.
6. Alkida Balliu, Juho Hirvonen, Dennis Olivetti, and Jukka Suomela. Hardness of minimal symmetry breaking in distributed computing. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 369–378, 2019.
7. Sebastian Brandt. An automatic speedup theorem for distributed problems. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 379–388, 2019.
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 *Proceedings of the 48th ACM Symposium on the Theory of Computing (STOC)*, pages 479–488, 2016.
9. Sebastian Brandt, Juho Hirvonen, Janne H. Korhonen, Tuomo Lempiäinen, Patric R.J. Östergård, Christopher Purcell, Joel Rybicki, Jukka Suomela, and Przemysław Uznaundefinedski. LCL problems on grids. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 101–110, 2017.
10. Yi-Jun Chang, Qizheng He, Wenzheng Li, Seth Pettie, and Jara Uitto. Distributed edge coloring and a special case of the constructive lovász local lemma. *ACM Trans. Algorithms*, 16(1), 2019.
11. Yi-Jun Chang, Tsvi Kopelowitz, and Seth Pettie. An exponential separation between randomized and deterministic complexity in the local model. *SIAM J. Comput.*, 48(1):122–143, 2019.
12. Yi-Jun Chang and Seth Pettie. A time hierarchy theorem for the local model. *SIAM J. Comput.*, 48(1):33–69, 2019.
13. Yi-Jun Chang, Jan Studený, and Jukka Suomela. Distributed graph problems through an automata-theoretic lens. *arXiv:2002.07659*, 2020.
14. Kai-Min Chung, Seth Pettie, and Hsin-Hao Su. Distributed algorithms for the Lovász local lemma and graph coloring. *Distributed Computing*, 30:261–280, 2017.
15. Manuela Fischer and Mohsen Ghaffari. Sublogarithmic distributed algorithms for Lovász local lemma with implications on complexity hierarchies. In *Proceedings of the 31st International Symposium on Distributed Computing (DISC)*, pages 18:1–18:16, 2017.
16. Mohsen Ghaffari, David G. Harris, and Fabian Kuhn. On derandomizing local distributed algorithms. In *Proceedings of 59th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 662–673, 2018.
17. Nathan Linial. Locality in distributed graph algorithms. *SIAM J. Comput.*, 21(1):193–201, 1992.

18 Gary L. Miller and John H. Reif. Parallel tree contraction–Part I: fundamentals. *Advances in Computing Research*, 5:47–72, 1989.

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

20 Dennis Olivetti. Brief announcement: Round eliminator: A tool for automatic speedup simulation. In *Proceedings of the 39th Symposium on Principles of Distributed Computing (PODC)*, pages 352–354. ACM, 2020.

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

22 Václav Rozhoň and Mohsen Ghaffari. Polylogarithmic-time deterministic network decomposition and distributed derandomization. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, 2020.

Improved Hardness of Approximation of Diameter in the CONGEST Model

Ofer Grossman
MIT, Cambridge, MA, USA
ofer.grossman@gmail.com

Seri Khoury
University of California, Berkeley, CA, USA
seri_khoury@berkeley.edu

Ami Paz
Faculty of Computer Science, Universität Wien, Austria
ami.paz@univie.ac.at

Abstract

We study the problem of approximating the diameter D of an unweighted and undirected n-node graph in the CONGEST model. Through a connection to extremal combinatorics, we show that a $(6/11 + \epsilon)$-approximation requires $\Omega(n^{1/6}/\log n)$ rounds, a $(4/7 + \epsilon)$-approximation requires $\Omega(n^{1/4}/\log n)$ rounds, and a $(3/5 + \epsilon)$-approximation requires $\Omega(n^{1/3}/\log n)$ rounds. These lower bounds are robust in the sense that they hold even against algorithms that are allowed to return an additional small additive error. Prior to our work, only lower bounds for $(2/3 + \epsilon)$-approximation were known [Frischknecht et al. SODA 2012, Abboud et al. DISC 2016].

Furthermore, we prove that distinguishing graphs of diameter 3 from graphs of diameter 5 requires $\Omega(n/\log n)$ rounds. This stands in sharp contrast to previous work: while there is an algorithm that returns an estimate $\lfloor 2/3D \rfloor \le \tilde{D} \le D$ in $\tilde{O}(\sqrt{n} + D)$ rounds [Holzer et al. DISC 2014], our lower bound implies that any algorithm for returning an estimate $2/3D \le \tilde{D} \le D$ requires $\tilde{\Omega}(n)$ rounds.

2012 ACM Subject Classification Mathematics of computing → Approximation algorithms; Theory of computation → Distributed computing models

Keywords and phrases Distributed graph algorithms, Approximation algorithms, Lower bounds

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.19

Funding *Ofer Grossman*: Supported by a Fannie and John Hertz Foundation Fellowship.
Ami Paz: Supported by the Austrian Science Fund (FWF): P 33775-N, Fast Algorithms for a Reactive Network Layer.

Acknowledgements We thank Noga Alon for useful pointers regarding generalized polygons. This work was done while the fist two authors were visiting the Simon's Institute for the Theory of Computing.

1 Introduction and Related Work

The diameter D of a graph is one of the most fundamental parameters in graph theory. In distributed computing, the diameter is of utmost importance, as it captures the minimal number of rounds needed for a message to traverse all the nodes in the network. The complexity of computing the exact or approximate value of the diameter has been extensively studied in the distributed setting [1, 6, 7, 14, 16–19, 21, 23].

In the standard CONGEST model, the complexity of computing the exact diameter is $\Theta(n/\log n + D)$ rounds [14, 19]. On the other hand, there is a folklore algorithm yielding a 1/2-approximation for the diameter in $O(D)$ rounds: running a BFS (from an arbitrary node) and returning its depth.

© Ofer Grossman, Seri Khoury, and Ami Paz;
licensed under Creative Commons License CC-BY
34th International Symposium on Distributed Computing (DISC 2020).
Editor: Hagit Attiya; Article No. 19; pp. 19:1–19:16
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Table 1 A summary of the state of the art results for diameter approximation.

Approx.	Bound	Ref. and Comments
Exact	$\tilde{\Theta}(n)$	[14, 19]
2 vs. 3	$\tilde{\Omega}(n)$	[18]
$\lfloor 2/3D \rfloor < \tilde{D} \le D$	$O(n^{1/2} + D)$	[16]
$2/3 + \epsilon$	$\tilde{\Omega}(n)$	[1]
3 vs. 5	$\tilde{\Omega}(n)$	**This paper** (Theorem 4)
$3/5 + \epsilon$	$\tilde{\Omega}(n^{1/3})$	**This paper** (Theorem 3)
$4/7$	$O(n^{1/3} + D)$	[3]
$4/7 + \epsilon$	$\tilde{\Omega}(n^{1/4})$	**This paper** (Theorem 2)
$6/11 + \epsilon$	$\tilde{\Omega}(n^{1/6})$	**This paper** (Theorem 1)
$1/2$	$O(D)$	Folklore

This raises the following natural question: for values of $1/2 < \alpha < 1$, how hard is it to α-approximate the diameter? Conversely, one may wonder how good of an approximation α is achievable in sub-linear time, and even in sub-polynomial time, as stated next.

- For which values of α does there exist a sub-polynomial time α-approximation algorithm for the diameter?
- For which values of α does there exist a truly sub-linear time α-approximation algorithm for the diameter?

We make progress on both these questions. For the first, we show that α must be at most $6/11$. For the second, we show that α must be at most $3/5$. The previous best known upper bound on α, for both cases, was $2/3$ [1]. All the results that are presented in this work, as well as the ones we compare with, are for unweighted and undirected graphs.

Our proofs use the well-established technique of reductions from communication complexity to distributed computing. Our main technical novelty is an interesting connection between extremal combinatorics, and specifically the existence of *generalized polygons* [15], and diameter approximation in the distributed setting. This extends prior work connecting extremal combinatorics and distributed computing [2, 8, 10, 12].

1.1 Our Contribution

Polynomial lower bound for .546-approximation. Our main result is that no sub-polynomial time algorithm can get better than a 6/11-approximation.

▶ **Theorem 1.** *For any constant $0 < \epsilon < 5/11$, any algorithm for finding a $(6/11 + \epsilon)$-approximation for the diameter in the* CONGEST *model requires $\Omega(n^{1/6}/\log n)$ rounds.*

We prove analogous theorems for $(4/7 + \epsilon)$-approximation and $(3/5 + \epsilon)$-approximation, with lower bounds of $\Omega(n^{1/4}/\log n)$ and $\Omega(n^{1/3}/\log n)$, respectively.

▶ **Theorem 2.** *For any constant $0 < \epsilon < 3/7$, any algorithm for finding a $(4/7 + \epsilon)$-approximation for the diameter in the* CONGEST *model requires $\Omega(n^{1/4}/\log n)$ rounds.*

▶ **Theorem 3.** *For any constant $0 < \epsilon < 2/5$, any algorithm for finding a $(3/5 + \epsilon)$-approximation for the diameter in the* CONGEST *model requires $\Omega(n^{1/3}/\log n)$ rounds.*

These resutls hold even against constant diameter graphs and even against randomized algorithms that succeed with probability at least 2/3. Prior to our work, besides the near-linear lower bound of [14] for exact diameter, only a lower bound for $(2/3+\epsilon)$-approximation was known: Abboud et al. [1] showed an $\Omega(n/\log^3 n)$ lower bound for this approximation factor. We note that Theorems 1, 2, and 3, as well as the aforementioned lower bound, also apply for algorithms that allow a constant *additive* error, in addition to the multiplicative one, as we explain in Section 1.2.

Near-linear lower bound for distinguishing diameter 3 vs 5. Next, we prove that distinguishing graphs of diameter 3 from graphs of diameter 5 requires a near-linear number of rounds.

▶ **Theorem 4.** *Any algorithm for distinguishing graphs of diameter 3 from graphs of diameter 5 in the* CONGEST *model requires $\Omega(n/\log n)$ rounds.*

We find this result rather surprising. There exist an algorithm [16] running in $O(\sqrt{n\log n}+D)$ rounds and returning an estimate $\lfloor\frac{2D}{3}\rfloor \le \tilde{D} \le D$. While the rounding in this equation might seem like an artifact of the proof, Theorem 4 shows that it is actually necessary.

That is, an algorithm for finding an estimate $\frac{2}{3}D \le \tilde{D} \le D$ can be used to distinguish diameter 3 from diameter 5, and we show that such a distinction must require $\Omega(n/\log n)$ rounds – much more than the $O(\sqrt{n\log n}+D)$ running time of the algorithm.

1.2 Robust Approximation

When dealing with diameter approximation, an important distinction to make is between *robust* and *non-robust* lower bounds. For example, as discussed above, an algorithm that finds an approximation $\tilde{D}$ of the diameter satisfying $\lfloor\frac{2D}{3}\rfloor \le \tilde{D} \le D$ does not in general imply a $\frac{2}{3}$-approximation.

However, as the diameter gets larger, the approximation ratio does approach 2/3. One way to view this is by saying that our 3 vs 5 lower bound is not a "robust" lower bound for $(3/5+\epsilon)$-approximation. To show a "robust" lower bound for $(3/5+\epsilon)$-approximation, we need a stronger result, i.e., that for any constant β, it is hard to distinguish between graphs of diameter $(3/5)D-\beta$ and graphs of diameter D. This would show that finding a $(3/5+\epsilon)$-approximation of the diameter is hard not only in some low-diameter graphs, but also more generally. We formally define the notions of α-approximation for diameter and robust diameter lower bound.

▶ **Definition 5** (α-approximation for diameter)**.** *We say that an estimate $\tilde{D}$ is an α-approximation for diameter if*

$$\alpha D \le \tilde{D} \le D.$$

▶ **Definition 6** (Robust diameter lower bound)**.** *We say that α-approximating the diameter is robustly $T(n)$-hard if for any constant β, there is no algorithm which returns a value $\tilde{D}$ satisfying*

$$\alpha D-\beta \le \tilde{D} \le D$$

in $o(T(n))$ rounds in the CONGEST *model.*

In this paper, we prove both robust and non-robust lower bounds. The lower bounds presented in Theorems 1, 2, and 3 are robust, while the lower bound that is presented in Theorem 4 is not robust.

Notably, for a $(3/5+\epsilon)$-approximation, we give a robust lower bound of $\Omega(n^{1/3}/\log n)$, and a non-robust lower bound of $\Omega(n/\log n)$. The work of [16] rules out a robust lower bound better than $\Omega(\sqrt{n\log n})$, even for 2/3-approximation. This shows that there is an inherent, and large, gap between the robust and non-robust lower bounds.

This distinction between robust and non-robust approximation has been noted before, though not using this terminology. Holzer and Wattenhofer [18] showed that distinguishing diameter 2 from diameter 3 requires $\Omega(n/\log n)$ rounds, a result that can be viewed as a non-robust $(2/3+\epsilon)$-approximation lower bound. A robust lower bound of $\Omega(n/\log^3 n)$ for the same approximation ratio was later proven by Abboud et al. [1].

1.3 Further Related Work

The lower bound of Abboud et al. [1] for $(2/3+\epsilon)$-approximation follows from a lower bound for distinguishing between diameter $4\ell+2$ and $6\ell+1$, for some constant $\ell > 1$. Bringmann and Forster improved this result by showing the same hardness for distinguishing diameter $2\ell+1$ and $3\ell+1$ [5].

Very recently, in a concurrent and independent work [3], the authors show an upper bound of $O(n^{1/3}+D)$ for computing a 4/7-approximation for diameter.

All the results that are presented in this work are for unweighted graphs. For weighted graphs, Holzer and Pinsker [17] showed that $(1/2+\epsilon)$-approximation requires $\Omega(n/\log n)$ rounds. For (1/2)-approximation in the weighted case, one can compute single source shortest paths. The state of the art algorithm for single source shortest paths in the CONGEST model is by Forster and Nanongkai [13], who showed two algorithms for the problem. The first running in $\tilde{O}(\sqrt{nD})$ rounds and the second running in $\tilde{O}(\sqrt{n}D^{1/4}+n^{3/5}+D)$ rounds.

Road-map. In Section 2 we start with some basic definitions. The technical heart of the paper is in Sections 3 and 4. Theorem 4 is proved in Section 3, and Theorems 1, 2, and 3 are proved in Section 4.

2 Preliminaries

2.1 The Model

In the CONGEST model [22], a synchronized communication network of n computationally unbounded nodes is modeled by its communication graph $G=(V,E)$. Each of the nodes has a unique $O(\log n)$-bit identifier. The computation is split into rounds, and in each round each node can send a (possibly different) $O(\log n)$-bit message to each of its neighbors. The goal of the nodes is to compute some function of the network (e.g., its diameter, the value of the minimum vertex cover, etc.) while minimizing the number of communication rounds.

For a graph H that is not the input graph, we denote its set of nodes and edges by V_H and E_H, respectively. The distance between two nodes u,v in a graph G is denoted by $\mathrm{d}_G(u,v)$, and is the minimum number of hops in a path between them in G. The diameter D of the graph is the maximum distance between two nodes in it. The girth of the graph g is the minimum length of a cycle in it.

2.2 Communication Complexity

In the two-party communication setting [20, 26], two players, Alice and Bob, are given two input strings, $x,y\in\{0,1\}^K$, respectively, and need to jointly compute a function $f:\{0,1\}^K\times\{0,1\}^K\to\{\text{TRUE},\text{FALSE}\}$ of their inputs, using a predefined communication protocol. The *communication complexity* of a function f is defined as follows. Definition 7 is a special case of [11, Definition 1].

▶ **Definition 7** (Communication Complexity). *Let $K \geq 1$ be an integer, f be a Boolean function $f : \{0,1\}^K \times \{0,1\}^K \to \{\text{TRUE}, \text{FALSE}\}$, and $\mathcal{Q}$ be the family of protocols that compute f correctly with probability at least $2/3$. Given 2 inputs $x, y \in \{0,1\}^K$, denote by $\Pi_Q(x,y)$ the transcript of a protocol Q on the inputs x, y, i.e., the sequence of bits that are exchanged between Alice and Bob. The cost of a protocol Q is $\text{Cost}(Q) = \max_{x,y\in\{0,1\}^k} |\Pi_Q(x,y)|$.*

The communication complexity of f, denoted by $CC_f(K)$, is defined to be the minimum cost over all the possible protocols that compute f correctly with probability at least $2/3$: $CC_f(K) = \min_{Q\in\mathcal{Q}} \text{Cost}(Q)$.

The set-disjointness function is defined as follows. For two strings $x, y \in \{0,1\}^K$, we say that x and y are not disjoint if and only if there is some index $i \in [K]$ such that $x_i = y_i = 1$. Otherwise we say that the strings are disjoint. It is well known that the communication complexity of set-disjointness is $\Omega(K)$ [24].

▶ Remark 8. Adding 0 bits to both input strings in matching locations does not change the output. Thus, we can assume a constant fraction of both input strings is 0 without affecting the asymptotic communication complexity. We use this fact in Section 4.

2.3 Lower Bound Graphs

Our lower bounds use the standard notion of family of lower bound graphs (see, e.g., [9]).

▶ **Definition 9** (Family of Lower Bound Graphs). *Let $K > 1$ be an integer, $f : \{0,1\}^K \times \{0,1\}^K \to \{\text{TRUE}, \text{FALSE}\}$ be a boolean function, and P be a graph predicate. A family of graphs $\left\{G_{(x,y)} = (V_{(x,y)}, E_{(x,y)}) \mid x, y \in \{0,1\}^K\right\}$ where each $G_{(x,y)}$ has a partition of the set of nodes $V_{(x,y)} = V_A \dot\cup V_B$ is said to be a family of* lower bound graphs for the CONGEST model w.r.t. f and P *if the following properties hold:*

1. *Only the existence of nodes in V_A or edges in $V_A \times V_A$ may depend on x;*
2. *Only the existence of nodes in V_B or edges in $V_B \times V_B$ may depend on y;*
3. *$G_{(x,y)}$ satisfies the predicate P iff $f(x,y) = \text{TRUE}$.*

For such a family, we denote by $C = E(V_A, V_B)$ the *cut*, i.e., the set of edges between V_A and V_B.

We use the following theorem, which is standard in the context of reductions to communication complexity (see, for example [1,9,10,14,17]). Its proof is by a standard simulation argument and appears in [9].

▶ **Theorem 10.** *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 of lower bound graphs for the CONGEST model w.r.t. f and P then any algorithm for deciding P in the CONGEST model requires $\Omega\left(\frac{\text{CC}_f(K)}{|C|\log n}\right)$ rounds.*

2.4 Generalized Polygons

Our proofs in Section 4 use the existence of generalized polygons [15]. A generalized polygon is an incidence relation whose incidence graph has several nice properties. In our context, we use the following key property of a a generalized polygon's incidence graph: its girth is twice its diameter. For the sake of simplifying the presentation, we also use the fact that the incidence graphs are balanced.

We use the notation $H = (L, R, E_H)$ to denote a bipartite graph H, where the bi-partition of the vertex set of H is L and R, and the set of edges of H is E_H. When $|L| = |R| = p$, we say that H is a *balanced* bipartite graph of size $2p$.

▶ **Definition 11.** *For two integers $p \geq t \geq 3$, we denote by* $\mathrm{Ex}(p,t)$ *the maximum number of edges in a balanced bipartite graph of size* $2p$*, diameter* t*, and girth* $2t$*.*

For $t \in \{3,4,6\}$, there are generalized polygons whose incidence graph has $2p$ nodes, diameter t, girth $2t$, and $\Theta(p^{1+\frac{1}{t-1}})$ edges. The cases of $t = 3$ and $t = 4$ were shown by Singelton [25], and Benson [4] gave a simplified proof and extended the result for $t = 6$. This is summarized in the following theorem, which will be used later without explicitly re-mentioning generalized polygons.

▶ **Theorem 12** ([4, 25]). *For* $t \in \{3,4,6\}$*, it holds that* $\mathrm{Ex}(p,t) = \Omega(p^{1+\frac{1}{t-1}})$*.*

3 Diameter 3 vs 5

In this section we prove the following theorem.

▶ **Theorem 4.** *Any algorithm for distinguishing graphs of diameter 3 from graphs of diameter 5 in the* CONGEST *model requires* $\Omega(n/\log n)$ *rounds.*

To prove Theorem 4, we show a family of lower bound graphs $\{G_{(x,y)} \mid x,y \in \{0,1\}^K\}$ with respect to the set-disjointness function and the graph predicate that distinguishes between graphs of diameter 3 and graphs of diameter 5. That is, the predicate is defined only on a graph G with either 3 or 5, and is TRUE if and only if G has diameter 5. We start with the fixed graph construction G and then we show how to get the graph $G_{(x,y)}$ given two strings $x, y \in \{0,1\}^K$.

The Fixed Graph Construction G. The fixed graph construction is defined as follows. There are 8 sets of nodes $S, C^1, A^1, B^1, C^2, A^2, B^2, T$, each of size $p = n/8$. Each of the sets S, A^1, B^1, A^2, B^2, T is an independent set, and C^1 and C^2 are cliques. The nodes in the sets are denoted $S = \{s_i \mid i \in [p]\}$, $T = \{t_i \mid i \in [p]\}$, and for $h \in \{1,2\}$, $A^h = \{a^h_i \mid i \in [p]\}$, $B^h = \{b^h_i \mid i \in [p]\}$, and $C^h = \{C^h_i \mid i \in [p]\}$.

The connections between the sets are defined as follows. Each pair of sets $H_1 \neq H_2 \in \{S, C^1, A^1, B^1\}$ is connected by a perfect matching, where we connect the i'th node in H_1 to the i'th node in H_2. For example, the sets S and C^1 are connected by the perfect matching $\{(s_i, c^1_i) \mid i \in [p]\}$. Similarly, each pair of sets in $\{T, C^2, A^2, B^2\}$ is connected by a perfect matching. This concludes the fixed graph construction G. Let $K = p^2$. We define the graph $G_{(x,y)}$, given two strings $x, y \in \{0,1\}^K$, as follows.

Obtaining $G_{(x,y)}$ from G and $x, y \in \{0,1\}^K$. For each of the strings x and y, we index the $K = p^2$ positions by $x_{(i,j)}$ and $y_{(i,j)}$ for $i, j \in [p]$. The set of nodes of $G_{(x,y)}$ is exactly as in G. The set of edges of $G_{(x,y)}$ contains all the edges in G, and the following edges between pairs of nodes in $A^1 \times A^2$ and between pairs of nodes in $B^1 \times B^2$.

$$\{(a^1_i, a^2_j) \mid x_{(i,j)} = 0\}; \quad \{(b^1_i, b^2_j) \mid y_{(i,j)} = 0\}.$$

That is, if $x_{(i,j)} = 0$, we add an edge between a^1_i and a^2_j, and if $y_{(i,j)} = 0$, we add an edge between b^1_i and b^2_j. This concludes the definition of $G_{(x,y)}$ (See also Figure 1, for an illustration). Next, we prove that $G_{(x,y)}$ has diameter 3 if the strings x and y are disjoint, and otherwise it has diameter at least 5. We prove this in Lemmas 13 and 14.

▶ **Lemma 13.** *If the strings x and y are disjoint, then the diameter of $G_{(x,y)}$ is 3.*

Proof. We show that for any two nodes u, v, $d_{G_{(x,y)}}(u,v) \leq 3$. Let $L = S \cup C^1 \cup A^1 \cup B^1$, and let $R = T \cup C^2 \cup A^2 \cup B^2$ The proof is by the following case analysis.

1. $u, v \in L$ or $u, v \in R$: We prove the claim for the case in which $u, v \in L$. The case for which $u, v \in R$ is similar. Observe that any node in L is connected by an edge to some node in C^1. Hence, since C^1 is a clique, this implies that $d_{G_{(x,y)}}(u,v) \leq 3$.
2. $u \in L$ and $v \in R$: Hence, u belongs to one of the sets in $\{S, C^1, A^1, B^1\}$ and v belongs to one of the sets in $\{T, C^2, A^2, B^2\}$. We assume that $u \in S$ and $v \in T$; the proof for the other cases is similar. Let i be such that $u = s_i$, and j such that $v = t_j$. Since the sets are disjoint, it holds that either $x_{(i,j)} = 0$, or $y_{(i,j)} = 0$ (or both). Hence, either there is an edge between a_i^1 and a_j^2, or there is an edge between b_i^1 and b_j^2 (or both), and assume the former without loss of generality. Since s_i is connected to a_i^1 and t_j is connected to a_j^2, we have $d_{G_{(x,y)}}(s_i, t_j) \leq 3$. Furthermore, one can verify that the distance between s_i and t_j cannot be smaller than 3, which implies that the diameter of the graph is 3. ◀

▶ **Lemma 14.** *If the strings x and y are not disjoint, then the diameter of $G_{(x,y)}$ is at least 5.*

Proof. As the sets are not disjoint, there are $i, j \in [p]$ for which it holds that $x_{(i,j)} = y_{(i,j)} = 1$. We show that in this case, any path P from s_i to t_j is of length at least 5, i.e., $d_{G_{(s_i,t_j)}}(u,v) \geq 5$. Observe that any path P from s_i to t_j must either pass from a node in A^1 to a node in A^2, or from a node in B^1 to a node in B^2. We assume that former case; the latter is similar. The proof is by the following case analysis.

1. The path P visits a node $a_{j'}^2 \in A^2$ for which $j' \neq j$: Observe that $d_{G_{(x,y)}}(s_i, a_{j'}^2) \geq 2$, and that $d_{G_{(x,y)}}(a_{j'}^2, t_j) = 3$. Hence, $d_{G_{(x,y)}}(s_i, t_j) \geq 5$.
2. The path P visits a_j^2. Since $x_{(i,j)} = 1$, there is no edge between a_i^1 and a_j^2. This implies that $d_{G_{(x,y)}}(s_i, a_j^2) \geq 4$, and hence $d_{G_{(x,y)}}(s_i, t_j) \geq 5$. ◀

Proof of Theorem 4. First, we define $V_A = S \cup T \cup A^1 \cup A^2 \cup C^1 \cup C^2$, and $V_B = B^1 \cup B^2$. Lemmas 13 and 14 imply that $\{G_{(x,y)} \mid x, y \in \{0,1\}^K\}$ is a family of lower bound graphs with respect to the set-disjointness problem and the graph predicate that distinguishes between graphs of diameter 3 and graphs of diameter 5. Observe that the cut size is $E(V_A, V_B) = \Theta(p)$, and $p = \Theta(n)$. Hence, since the length of the input strings is $K = p^2$, and since the communication complexity of set-disjointness is $\Omega(K) = \Omega(p^2)$, Theorem 10, implies that any algorithm for deciding whether a graph has diameter 3 or 5 in the CONGEST model requires $\Omega(p^2/p \log p) = \Omega(p/\log p) = \Omega(n/\log n)$ rounds. ◀

The connectivity of $G_{(x,y)}$. One may wonder about the connectivity of $G_{(x,y)}$. If the graph $G_{(x,y)}$ is not connected, then the construction wouldn't be meaningful as there is a trivial lower bound of $\Omega(D)$, where D is the diameter of the graph, which is ∞ in graphs that are not connected. Observe that the only case in which $G_{(x,y)}$ is not connected is when $x = y = 1^K$. To ensure connectivity (and in fact constant diameter, due to the cliques C^1 and C^2), we can assume that at least one of the strings x or y has a zero bit. Clearly, the communication complexity of set-disjointness doesn't change under this assumption. In fact, Remark 8 allows to make an even stronger assumption, which we only need in the next section.

4 Robust Lower Bounds

In this section we prove robust lower bounds for $(6/11 + \epsilon)$-approximation, $(4/7 + \epsilon)$-approximation, and $(3/5 + \epsilon)$-approximation of the diameter. Our lower bounds follow from the following theorem.

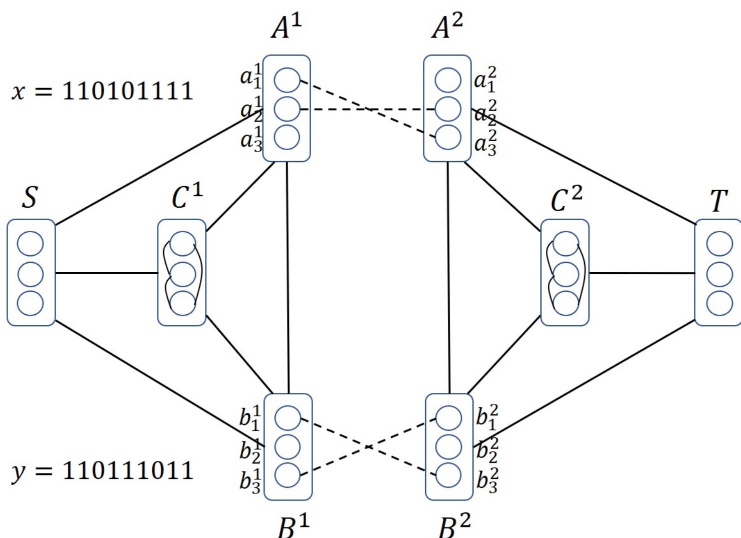


Figure 1 Diameter 3 vs 5: An example for the graph construction $G_{(x,y)}$ for $p = 3$: There are 8 sets of nodes $S, C^1, A^1, A^2, C^2, B^1, B^2, T$, each of size $p = 3$. Each of the sets S, A^1, A^2, B^1, B^2, T forms an independent set, and the sets C^1 and C^2 are cliques. In this diagram, an edge between two sets represents a perfect matching connecting them. For example, the edge between S and C^1 represents all the edges in $\{(s_i, c_i^1) \mid i \in [p]\}$. The dashed edges between A^1 and A^2 are the input edges which depend on the input sting x. Recall that we index the $p^2 = 9$ positions of x by pairs of indices $(i, j) \in [p] \times [p]$. In this example, we have that $x_{(1,3)} = x_{(2,2)} = 0$, and all the other bits of x are 1's. Hence, the only edges between A^1 and A^2 are (a_1^1, a_3^2) and (a_2^1, a_2^2). Similarly, the dashed edges between B^1 and B^2 represent the input edges which depend on the string y. Since in this example we have $y_{(1,3)} = y_{(3,1)} = 0$, and all the other bits of y are 1's, the only edges between B^1 and B^2 are (b_1^1, b_3^2) and (b_3^1, b_1^2).

▶ **Theorem 15.** *Let $t \in \{3, 4, 6\}$. For any constant $0 < \epsilon < 1 - \frac{t}{2t-1}$, any algorithm for computing a $(\frac{t}{2t-1} + \epsilon)$-approximation to the diameter in the* CONGEST *model requires $\Omega\left(n^{1/t}/\log n\right)$ rounds, where n is the number of nodes in the input graph.*

The theorem has the following consequences, when plugging in $t = 6$, $t = 4$ and $t = 3$, in this order.

▶ **Theorem 1.** *For any constant $0 < \epsilon < 5/11$, any algorithm for finding a $(6/11 + \epsilon)$-approximation for the diameter in the* CONGEST *model requires $\Omega(n^{1/6}/\log n)$ rounds.*

▶ **Theorem 2.** *For any constant $0 < \epsilon < 3/7$, any algorithm for finding a $(4/7 + \epsilon)$-approximation for the diameter in the* CONGEST *model requires $\Omega(n^{1/4}/\log n)$ rounds.*

▶ **Theorem 3.** *For any constant $0 < \epsilon < 2/5$, any algorithm for finding a $(3/5 + \epsilon)$-approximation for the diameter in the* CONGEST *model requires $\Omega(n^{1/3}/\log n)$ rounds.*

Recall that $\mathrm{Ex}(p, t)$ is the maximum number of edges of a balanced bipartite graph of size $2p$, diameter t, and girth $2t$ (see Definition 11). Let $t \in \{3, 4, 6\}$, $p \geq t$ and let $K = \mathrm{Ex}(p, t)$. To prove Theorem 15, we show a family of lower bound graphs $\left\{G_{(x,y)} \mid x, y \in \{0,1\}^K\right\}$ with respect to the set-disjointness function and the graph predicate that distinguishes between graphs of diameter $t(b + 1) + 1$ and graphs of diameter $(2t - 1)b$, for some integer $b = \Theta(1/\epsilon)$ that will be chosen later.

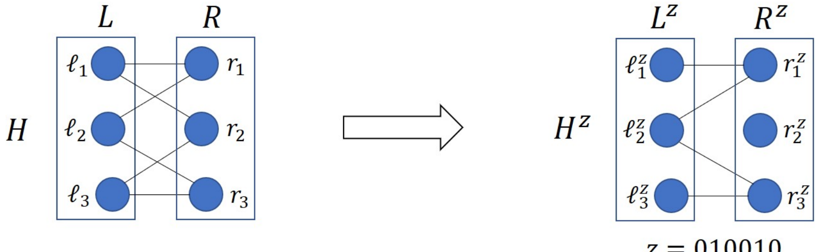


Figure 2 An example for H and H^z. In this example $t = p = 3$ and therefore H is a bipartite graph of diameter $t = 3$ and girth $2t = 6$. For these parameters, we have $K = \text{Ex}(3, 3) = 6$. Recall that $\pi : E_H \to [K]$ is an arbitrary $1 : 1$ mapping from the set of pairs $(\ell_i, r_j) \in E_H$ to $[K]$. In this example, we choose $\pi(\ell_1, r_1) = 1, \pi(\ell_1, r_2) = 2, \pi(\ell_2, r_1) = 3, \pi(\ell_2, r_3) = 4, \pi(\ell_3, r_2) = 5$ and $\pi(\ell_3, r_3) = 6$. Furthermore, in this example we have $z = 010010$. Hence, since H^z is obtained from H by keeping only the edges that correspond to the 0 bits in z, we have that the only edges in H^z are $(\ell_1^z, r_1^z), (\ell_2^z, r_1^z), (\ell_2^z, r_3^z)$ and (ℓ_3^z, r_3^z).

The rest of this section is organized as follows. In section 4.1, we start with the description of $G_{(x,y)}$ given two strings $x, y \in \{0,1\}^K$. In section 4.2, we show that $\left\{G_{(x,y)} \mid x, y \in \{0,1\}^K\right\}$ is a family of lower bound graphs with the required properties. In Section 4.3, we deduce Theorem 15. While the graphs $G_{(x,y)}$ need to be connected, we ignore this fact in Section 4.1; in Section 4.4 we show how to slightly modify the construction so that the graphs become connected (and even of constant diameter) for any x and y.

4.1 Description of $G_{(x,y)}$

Given two strings $x, y \in \{0,1\}^K$, we describe the graph $G_{(x,y)}$ in three steps. In the first step, given a string $z \in \{0,1\}^K$, we define a bipartite graph H^z. Roughly speaking, H^z is obtained from a densest possible balanced bipartite graph H of size $2p$, diameter t and girth $2t$, where we keep only some of the edges of H in H^z according to the string z. In the second step, we define a graph $\widetilde{H}^z$, which is obtained form H^z by stretching each edge to a path of length b. In the third step, we describe how to get $G_{(x,y)}$ from $\widetilde{H}^x$ and $\widetilde{H}^y$.

Description of H^z. Let $H = (L, R, E_H)$ be a balanced bipartite graph of size $2p$, diameter t, girth $2t$, and a maximum number of edges. That is, the number of edges of H is $|E_H| = \text{Ex}(p, t) = K$. We denote the nodes of H by $L = \{\ell_1, \cdots, \ell_p\}$ and $R = \{r_1, \cdots, r_p\}$. Furthermore, let $\pi : E_H \to [K]$ be an enumeration of E_H, that is, π is an arbitrary ordering over the set of pairs $E_H \subseteq L \times R$. By this mapping, each bit of a string $z \in \{0,1\}^K$ corresponds to a unique edge in E_H.

Given a string $z \in \{0,1\}^K$, the graph H^z is defined as follows. H^z is a version of H where we keep only the edges for which the corresponding bits in z are 0. More formally, $H^z = (L^z, R^z, E_{H^z})$ is a balanced bipartite graph with $|L^z| = |R^z| = p$, where $L^z = \{\ell_1^z, \cdots, \ell_p^z\}$ and $R^z = \{r_1^z, \cdots, r_p^z\}$. A pair of nodes $(\ell_i^z, r_j^z) \in L^z \times R^z$ is connected by an edge in H^z if (ℓ_i, r_i) is an edge of H and $z_{\pi(\ell_i, r_i)} = 0$, that is, $E_{H^z} = \{(\ell_i^z, r_j^z) \mid (\ell_i, r_j) \in E_H \wedge z_{\pi(\ell_i, r_j)} = 0\}$. See Figure 2 for an illustration of obtaining H^z from H and an input string $z \in \{0,1\}^K$.

Description of $\widetilde{H}^z$. $\widetilde{H}^z$ is obtained from H^z by replacing each edge $(\ell_i^z, r_j^z) \in E_{H^z}$ with a path of $b + 1$ nodes and b edges, starting at ℓ_i^z and ending at r_j^z, where b is some positive integer to be chosen later. We denote this path by $P^z_{(\ell_i^z, r_j^z)}$. We slightly abuse notation and

denote the set of nodes on this path also by $P^z_{(\ell^z_i, r^z_j)}$. We sometimes treat $P^z_{(\ell^z_i, r^z_j)}$ as a set of nodes, and sometimes as a path, but this will be clear from the context. Hence, the set of nodes of $\widetilde{H}^z$ is

$$V_{\widetilde{H}^z} = L^z \cup R^z \cup \bigcup_{(\ell^z_i, r^z_j) \in E_{H^z}} P^z_{(\ell^z_i, r^z_j)}$$

and the edges of $\widetilde{H}^z$ are only the ones on the paths in $\{P^z_{(\ell^z_i, r^z_j)} \mid (\ell^z_i, r^z_j) \in E_{H^z}\}$. Observe that $\widetilde{H}^z$ is not necessarily bipartite.

Obtaining $G_{(x,y)}$ from $\widetilde{H}^x$ and $\widetilde{H}^y$. Given two input strings $x, y \in \{0,1\}^K$, $G_{(x,y)}$ is composed of $\widetilde{H}^x$ and $\widetilde{H}^y$ where we add a perfect matching between L^x and L^y, $\{(\ell^x_i, \ell^y_i) \mid i \in [p]\}$, and a perfect matching between R^x and R^y, $\{(r^x_i, r^y_i) \mid i \in [p]\}$. This concludes our construction. See also figures 3 and 4 for illustrations of $G_{(x,y)}$. In these figures, we also illustrate $\widetilde{H}^z$ for $z = x \wedge y$, where the string $x \wedge y \in \{0,1\}^K$ is defined by $(x \wedge y)_h = x_h \cdot y_h$ for any $h \in [K]$. That is, $(x \wedge y)_h = 1$ if and only if $x_h = y_h = 1$. The reason that we illustrate $\widetilde{H}^z$ in the same figures is that our proof heavily relies on comparing distances in $G_{(x,y)}$ to distances in $\widetilde{H}^z$ for $z = x \wedge y$. Figure 3 is an illustration of the two graphs when the strings are not disjoint, while Figure 4 is an illustration of the two graphs when the strings are disjoint. Before we prove that $\left\{G_{(x,y)} \mid x, y \in \{0,1\}^K\right\}$ is a family of lower bound graphs, we show the following two useful properties of the balanced bipartite graph $H = (L, R, E_H)$ that was described above.

▶ **Property 1.** *If t is odd, then the distance between any two nodes $u, v \in L$ in H is at most $t - 1$. Similarly, the distance between any two nodes $u, v \in R$ in H is at most $t - 1$.*

Proof. The distance between every two nodes in H is at most its diameter t, but the distance between every two nodes in the same side of the bi-partition is even, so it is at most $t - 1$. ◀

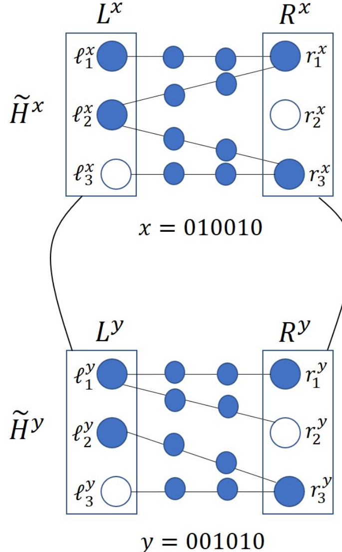
▶ **Property 2.** *If t is even, then the distance between any pair of nodes $u \in L$ and $v \in R$ in H is at most $t - 1$.*

Proof. The distance between every two nodes in H is at most its diameter t, and the distance between two nodes in different sides of the bi-partition is odd, so it is at most $t - 1$. ◀

4.2 $G_{(x,y)}$ is a family of lower bound graphs

Our goal in this section is to prove that $\{G_{(x,y)} \mid x, y \in \{0,1\}^K\}$ is a family of lower bound graphs with respect to the set-disjointness function and the graph predicate that distinguishes graphs of diameter $t(b+1)+1$ from graphs of diameter $(2t-1)b$. For the rest of the paper, $z = x \wedge y$. Our proof relies on comparing distances between nodes in $G_{(x,y)}$ to distances between nodes in $\widetilde{H}^z$. While the proof contains many technical details that require some care, it follows from the following simple intuition.

Intuition and overview of the proof. First, it is not very hard to see that the diameter of $G_{(x,y)}$ is roughly equal to the diameter of $\widetilde{H}^z$ (up to an additive $t+1$). Hence, it suffices to argue that if the strings x and y are disjoint, then the diameter of $\widetilde{H}^z$ is at most tb, and otherwise the diameter of $\widetilde{H}^z$ is at least $(2t-1)b$. The main idea is to note that H^z is isomorphic to H if and only if the strings x and y are disjoint. Hence, if the strings are disjoint, the diameter of H^z is equal to the diameter of H which is t, and since $\widetilde{H}^z$ is obtained from H^z by stretching each edge to a path of length b, the diameter of $\widetilde{H}^z$ is tb.



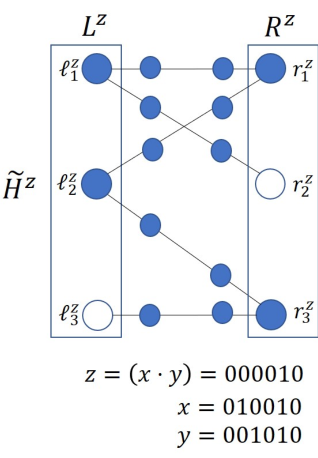
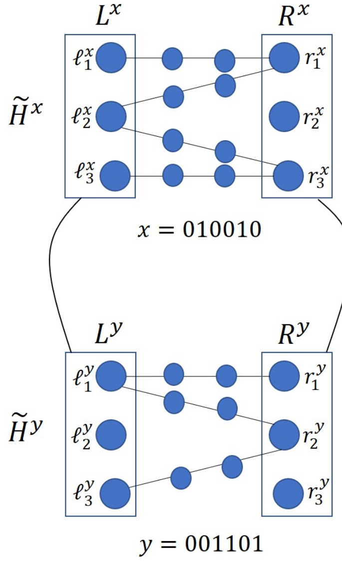


Figure 3 An Illustration of $G_{(x,y)}$ (on the left) and $\widetilde{H}^z$ for $z = x \wedge y$ (on the right). $G_{(x,y)}$ is composed of $\widetilde{H}^x$ and $\widetilde{H}^y$, where we add a perfect matching between L^x and L^y, $\{(\ell_i^x, \ell_i^y) \mid i \in [p]\}$, and a perfect matching between R^x and R^y, $\{(r_i^x, r_i^y) \mid i \in [p]\}$. The edge between L^x and L^y in this figure represents the matching between them. Similarly, the edge between R^x and R^y in this figure represents the matching between them. The parameters in this example are exactly as the ones chosen for Figure 2. That is, $t = p = 3$, $b = 3$, $\pi(\ell_1, r_1) = 1, \pi(\ell_1, r_2) = 2, \pi(\ell_2, r_1) = 3, \pi(\ell_2, r_3) = 4, \pi(\ell_3, r_2) = 5$ and $\pi(\ell_3, r_3) = 6$. Furthermore, we have $x = 010010, y = 001010$ and $z = x \wedge y = 000010$. Hence, the only paths of length b that we have in $\tilde{H}^x$ are $P^x_{(\ell_1^x, r_1^x)}, P^x_{(\ell_2^x, r_1^x)}, P^x_{(\ell_2^x, r_3^x)}$ and $P^x_{(\ell_3^x, r_3^x)}$. And the only paths of length b that we have in $\tilde{H}^y$ are $P^y_{(\ell_1^y, r_1^y)}, P^y_{(\ell_1^y, r_2^y)}, P^y_{(\ell_2^y, r_3^y)}$ and $P^y_{(\ell_3^y, r_3^y)}$. Observe that the path $P^x_{(\ell_3^x, r_2^x)}$ doesn't exist in $\widetilde{H}^x$, and that the path $P^y_{(\ell_3^y, r_2^y)}$ doesn't exist in $\widetilde{H}^y$. This is because $x_{\pi(\ell_3, r_2)} = x_5 = 1$ and $y_{\pi(\ell_3, r_2)} = y_5 = 1$. This implies that $z_{\pi(\ell_3, r_2)} = z_5 = 1$ as well, and therefore the path $P^z_{(\ell_3^z, r_2^z)}$ doesn't exist in $\widetilde{H}^z$. It is easy to see that the distance between ℓ_3^z and r_2^z in $\widetilde{H}^z$ is $5b = (2t-1)b$. One can verify that the distance between ℓ_3^x and r_2^x in $G_{(x,y)}$ is at least $5b = (2t-1)b$ as well. This illustrates that when the strings x and y are disjoint, the diameter of $G_{(x,y)}$ is at least $(2t-1)b$.

On the other hand, if H^z is not isomorphic to H, then there is an edge in H for which the corresponding edge in H^z doesn't exist. Since the girth of H is $2t$, it implies that there are two nodes in H^z at distance at least $(2t-1)$ from each other. Hence, the distance between the corresponding two nodes in $\widetilde{H}^z$ is at least $(2t-1)b$. Next, we formalize these ideas and give a more detailed proof. The non-disjointness case is proved in Lemma 19, which uses claims 16, 17 and 18. The disjointness case is proved in Lemma 23, which uses claims 20, 21, and 22. Recall that given a graph G and two nodes u, v in it, we denote by $\mathrm{d}_G(u, v)$ the distance between u and v in G.

Non-disjointness case

▷ Claim 16. If x and y are not disjoint, then the diameter of H^z is at least $2t-1$ and the diameter of $\widetilde{H}^z$ is at least $(2t-1)b$. In particular, there are $\ell_i^z \in L^z$ and $r_j^z \in R^z$ such that $\mathrm{d}_{\widetilde{H}^z}(\ell_i^z, r_j^z) \geq (2t-1)b$.



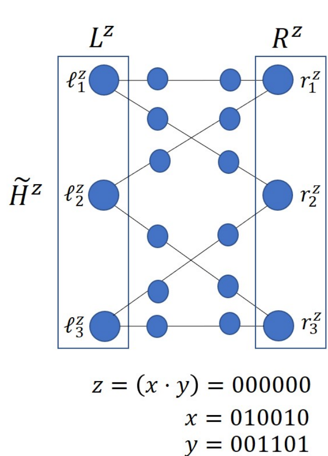


Figure 4 An illustration of $G_{(x,y)}$ and $\widetilde{H}^z$ for $z = x \wedge y$. The parameters in this example are exactly as the ones chosen for Figures 2 and 3. The only difference in this example compared to the one in Figure 3 is that the strings x and y are disjoint. Hence, $z = x \wedge y$ is an all zeros string. It is easy to see that in this case the diameter of $\widetilde{H}^z$ is tb. The key point of the proof is that the diameter of $G_{(x,y)}$ is not very much larger than the diameter of $\widetilde{H}^z$ (in fact, it is larger by at most $t+1$, which is negligible compared to tb for values of $b \gg t$). To illustrate this in this example, we show a path of length $tb + t = 3b + 3$ from ℓ_1^x and r_3^y. We start by moving from ℓ_1^x to ℓ_1^y using the edge (ℓ_1^x, ℓ_1^y) that is part of the matching between L^x and L^y. Then, we use the path of length b from ℓ_1^y to r_2^y, and the path of length b from r_2^y to ℓ_3^y. After that, we use the edge (ℓ_3^y, ℓ_3^x) to move to ℓ_3^x, and the path of length b from ℓ_3^x to r_3^x. Finally, we use the edge (r_3^x, r_3^y) to reach r_3^y. This example illustrates that the diameter of $G_{(x,y)}$ is relatively small if the strings are disjoint.

Proof. Observe that if the strings x and y are not disjoint, then there is an $h \in [K]$ for which it holds that $x_h = y_h = 1$. Hence, $z_h = 1$. Since H^z is obtained from H by keeping only the edges that correspond to the 0 bits in z, it follows that there is an edge $(\ell_i, r_j) \in H$ such that there is no edge between the corresponding pair (ℓ_i^z, r_j^z) in H^z. Hence, since H has girth $2t$, if follows that the distance between ℓ_i^z and r_j^z in H^z is at least $2t - 1$. Since $\widetilde{H}^z$ is obtained from H^z by replacing each edge with a path of length b, it follows that the diameter of $\widetilde{H}^z$ is at least $(2t-1)b$. ◁

▷ Claim 17. For any $(\ell_i, r_j) \in E_H$, if one of the paths $P^x_{(\ell_i^x, r_j^x)}$ and $P^y_{(\ell_i^y, r_j^y)}$ exists in $G_{(x,y)}$, then the path $P^z_{(\ell_i^z, r_j^z)}$ exists in $\widetilde{H}^z$. Similarly, if $P^z_{(\ell_i^z, r_j^z)}$ exists in $\widetilde{H}^z$, then either $P^x_{(\ell_i^x, r_j^x)}$ exists in $G_{(x,y)}$ or $P^y_{(\ell_i^y, r_j^y)}$ exists in $G_{(x,y)}$.

Proof. Let $h = \pi(\ell_i, r_j)$. Observe that if one of the paths $P^x_{(\ell_i^x, r_j^x)}$ and $P^y_{(\ell_i^y, r_j^y)}$ exists in $G_{(x,y)}$, then it must be the case that either $x_h = 0$ or $y_h = 0$. Hence, $z_h = 0$. Therefore there is an edge between ℓ_i^z and r_j^z in H^z, which is stretched to a path $P^z_{(\ell_i^z, r_j^z)}$ in $\widetilde{H}^z$. The other direction of the claim is proved similarly. ◁

▷ Claim 18. For any $\ell_i^x \in L^x$ and $r_j^x \in R^x$, it holds that $\mathrm{d}_{G_{(x,y)}}(\ell_i^x, r_j^x) \geq \mathrm{d}_{\widetilde{H}^z}(\ell_i^z, r_j^z)$.

Proof. Consider a shortest path between ℓ_i^x and r_j^x in $G_{(x,y)}$. Observe that this path is composed of edges crossing from $\widetilde{H}^x$ to $\widetilde{H}^y$ or vice versa (i.e., edges in $(L^x \times L^y) \cup (R^x \times R^y)$), and of paths of length b crossing from $L^x \cup L^y$ to $R^x \cup R^y$ or vice versa. Let q be the number of paths of length b crossing from $L^x \cup L^y$ to $R^x \cup R^y$ (or vice versa) that are used by the shortest path. And denote these paths by $P^1, P^2, \cdots, P^q$. Clearly, $\mathrm{d}_{G_{(x,y)}}(\ell_i^x, r_j^x) \geq qb$. Hence, it suffices to show that $qb \geq \mathrm{d}_{\widetilde{H}^z}(\ell_i^z, r_j^z)$.

For this, observe that for any $h \in [q]$, there are $ih, jh \in [p]$ and $w \in \{x, y\}$, for which $P^h = P^w_{(\ell^w_{ih}, r^w_{jh})}$ (That is, P^h is connecting either a pair $(\ell^x_{ih}, r^x_{jh}) \in L^x \times R^x$ or a pair $(\ell^w_{ih}, r^w_{jh}) \in L^y \times R^y$). Hence, by Claim 17, this implies that for any $h \in [q]$, the path $P^z_{(\ell^z_{ih}, r^z_{jh})}$ exists in $\widetilde{H}^z$. Therefore, by starting at ℓ_i^z and following these q paths of length b in $\widetilde{H}^z$ we reach r_j^z. Hence, $qb \geq \mathrm{d}_{\widetilde{H}^z}(\ell_i^z, r_j^z)$. ◁

▶ **Lemma 19.** *If x and y are not disjoint, then the diameter of $G_{(x,y)}$ is at least $(2t-1)b$.*

Proof. By Claim 16, if the strings are not disjoint, then there are $\ell_i^z \in L^z$ and $r_j^z \in R^z$ such that $\mathrm{d}_{\widetilde{H}^z}(\ell_i^z, r_j^z) \geq (2t-1)b$. Furthermore, by Claim 18, it holds that $\mathrm{d}_{G_{(x,y)}}(\ell_i^x, r_j^x) \geq \mathrm{d}_{\widetilde{H}^z}(\ell_i^z, r_j^z)$. Hence, there are two nodes in $G_{(x,y)}$ at distance at least $(2t-1)b$ from each other. ◀

Disjointness case

▷ Claim 20. If x and y are disjoint, then H^z is isomorphic to H. In particular, this implies:

1. H^z has diameter t.
2. for odd values of t, and for any two nodes $u, v \in V_{\widetilde{H}^z}$, if $u, v \in L^z$ or $u, v \in R^z$, then $\mathrm{d}_{\widetilde{H}^z}(u, v) = (t-1)b$.
3. for even values of t, and for any two nodes $u, v \in V_{\widetilde{H}^z}$ such that $u \in L^z$ and $v \in R^z$, it holds that $\mathrm{d}_{\widetilde{H}^z}(u, v) = (t-1)b$.

Proof. Observe that if the strings x and y are disjoint, then for any $h \in [K]$, either $x_h = 0$ or $y_h = 0$, so $z = x \wedge y$ is the all-zero string. Therefore, since H^z is obtained from H by keeping the edges that correspond to the 0 bits in z, H^z is isomorphic to H. Since H has diameter t, H^z also has diameter t.

Moreover, since H^z is isomorphic to H, by Property 1, it holds that for odd values of t, and for u, v that are on the same side (i.e., $u, v \in L^z$ or $u, v \in R^z$), it holds that $\mathrm{d}_{H^z}(u, v) = t-1$, and therefore $\mathrm{d}_{\widetilde{H}^z}(u, v) = (t-1)b$.

Similarly, by Property 2, it holds that for even values of t, and for u, v that are not on the same side (i.e., $u \in L^z$ and $v \in R^z$), it holds that $\mathrm{d}_{H^z}(u, v) = t-1$, and therefore $\mathrm{d}_{\widetilde{H}^z}(u, v) = (t-1)b$. ◁

For two nodes $u, v \in L^x \cup L^y \cup R^x \cup R^y$ in $G_{(x,y)}$, we say that u and v are on the same side if $u, v \in L^x \cup L^y$ or $u, v \in R^x \cup R^y$. Similarly, we say that u and v are on different sides if $u \in L^x \cup L^y$ and $v \in R^x \cup R^y$.

▷ Claim 21. For odd values of t, if x and y are disjoint then for any two nodes $u, v \in L^x \cup L^y \cup R^x \cup R^y$ that are on the same side (i.e., either $u, v \in L^x \cup L^y$ or $u, v \in R^x \cup R^y$) it holds that $\mathrm{d}_{G_{(x,y)}}(u, v) \leq (t-1)b + t$.

Due to space limitations, the proof of Claim 21 is deferred to the full version.

▷ Claim 22. For even values of t, if x and y are disjoint then for any two nodes $u, v \in L^x \cup L^y \cup R^x \cup R^y$ that are on different sides (i.e., $u \in L^x \cup L^y$ and $v \in R^x \cup R^y$) it holds that $\mathrm{d}_{G_{(x,y)}}(u, v) = (t-1)b + t$.

Due to space limitations, the proof of Claim 22 is deferred to the full version.

▶ **Lemma 23.** *If x and y are disjoint, then the diameter of $G_{(x,y)}$ is at most $tb+t+1$.*

Proof. Let $u, v \in V_{G_{(x,y)}}$ be two nodes in $G_{(x,y)}$. Let a_u^ℓ be the distance from u to the closest node in $L^x \cup L^y$, and let a_u^r be the distance from u to the closest node in $R^x \cup R^y$. a_v^ℓ and a_v^r are defined similarly for v. For example, if $u \in L^x \cup L^y$ then $a_u^\ell = 0$. The key point to note is that either $a_u^\ell + a_v^\ell \leq b+1$, or $a_u^r + a_v^r \leq b+1$. That is, either taking the two nodes to the left side of $G_{(x,y)}$ (i.e., to $L^x \cup L^y$) costs at most $b+1$, or taking the two nodes to the right side costs at most $b+1$. Similarly, it holds that either $a_u^\ell + a_v^r \leq b+1$, or $a_u^r + a_v^\ell \leq b+1$. The rest of the proof is by the following case analysis.

1. t is odd: By Claim 21, the distance between any two nodes in $L^x \cup L^y$ and the distance between any two nodes in $R^x \cup R^y$ is at most $(t-1)b+t$. Furthermore, we can either move u and v to $L^x \cup L^y$ by using at most $b+1$ steps (in total, for moving both u and v), or we can move u and v to $R^x \cup R^y$ by using at most $b+1$ steps (in total, for moving both u and v). After moving u and v to one of the sides, we can use Claim 21 and deduce that $\mathrm{d}_{G_{(x,y)}}(u,v) \leq (t-1)b+t+b+1 = tb+t+1$.
2. t is even: By Claim 22, the distance between any $u' \in L^x \cup L^y$ and $v' \in R^x \cup R^y$ is at most $(t-1)b+t$. Furthermore, we can either move u to $L^x \cup L^y$ and v to $R^x \cup R^y$ by using at most $b+1$ steps (in total, for moving both u and v), or we can move u to $R^x \cup R^y$ and v to $L^x \cup L^y$ by using at most $b+1$ steps (in total, for moving both u and v). Hence, we can use Claim 22 and deduce that $\mathrm{d}_{G_{(x,y)}}(u,v) \leq (t-1)b+t+b+1 = tb+t+1$. ◀

4.3 Proof of Theorem 15

First, we define the following partition $V = V_A \dot\cup V_B$ of the set of nodes of $G_{(x,y)}$. $V_A = V_{\widetilde{H}^x}$, and $V_B = V_{\widetilde{H}^y}$. Hence, the size of the cut $C = E(V_A, V_B)$ is $\Theta(p)$. This is because the only edges connecting between nodes in $\widetilde{H}^x$ and nodes in $\widetilde{H}^y$ in $G_{(x,y)}$ are the $2p$ edges of the matching between L^x and L^y, and the matching between R^x and R^y.

Since our goal is to show a lower bound as a function of the number of nodes n in the input graph $G_{(x,y)}$, we need to analyze the size of the cut and the size of the input strings with respect to n. By Theorem 12, we have that for $t \in \{3,4,6\}$, $K = \mathrm{Ex}(p,t) = \Omega(p^{1+\frac{1}{t-1}})$. Furthermore, by Remark 8 we can assume that a constant fraction of the bits in the strings x and y are 0. Hence, these 0 bits are translated to paths of length b in $G_{(x,y)}$. This implies that the number of nodes in $G_{(x,y)}$ is $n = \Theta(Kb) = \Omega(p^{1+\frac{1}{t-1}}b) = \Omega(p^{1+\frac{1}{t-1}})$ for constant values of b. Therefore, the size of the cut C is $\Theta(p) = O(n^{\frac{t-1}{t}})$.

Lemmas 19 and 23 imply that $\{G_{(x,y)} \mid x, y \in \{0,1\}^K\}$ is a family of lower bound graphs with respect to the set-disjointness function and the graph predicate that distinguishes between graphs of diameter $tb+t+1$ and graphs of diameter $(2t-1)b$. Hence, by Theorem 10 and the fact that the communication complexity of set-disjointness is $\Omega(K)$, we have that any algorithm for distinguishing between these two cases in the CONGEST model requires $\Omega\left(\frac{K}{|C|\log n}\right) = \Omega\left(\frac{n}{n^{(t-1)/t}\log n}\right) = \Omega\left(\frac{n^{1/t}}{\log n}\right)$ rounds. To get this lower bound for $(\frac{t}{2t-1}+\epsilon)$-approximation for diameter, we need that $\left(\frac{t}{2t-1}+\epsilon\right)(2t-1)b > tb+t+1$. Hence, we can pick $b = \Theta\left(\frac{t+1}{\epsilon(2t-1)}\right) = \Theta\left(\frac{1}{\epsilon}\right)$.

4.4 Handling the connectivity issue

One may wonder about the connectivity of the graph $G_{(x,y)}$. As the construction was described so far, there could be some values of x and y such that $G_{(x,y)}$ is not connected. In this section, we show how to slightly modify the construction of $G_{(x,y)}$ so that it is always connected, and in fact of constant diameter, without changing the analysis. Observe that

it suffices to make $\widetilde{H}^x$ always connected. This is because any node in $\widetilde{H}^y$ has some path connecting it to a node in $\widetilde{H}^x$. Since $\widetilde{H}^x$ is obtained from H^x by stretching each edge in H^x to a path of length b, it suffices to make H^x always connected, regardless of the input string x.

Recall that given a string $x \in \{0,1\}^K$, we defined H^x to be the graph obtained from H where we keep only the edges that correspond to the 0 bits in x. Recall that H is a balanced bipartite graph of size $2p$, diameter t and girth $2t$. Of course, H is always connected. But since some of the edges of H may not exist in H^x, H^x may not be connected. To ensure that H^x is connected, let S be a shortest paths tree starting from an arbitrary node in H. Of course, the number of edges in S is $O(p)$, which is small with respect to the size of the input string x which is $K = \mathrm{Ex}(p,t)$.

We modify the definition of H^x such that the edges that correspond to the spanning tree S always exist in H^x. In particular, their existence in H^x doesn't depend on x. For this, we need to modify the size of the string x to $K - |S| = \Theta(K)$, so that only the edges that are not in S depend on x. The proof that $\{G_{(x,y)} \mid (x,y) \in \{0,1\}^{K-|S|} \times \{0,1\}^K\}$ is a family of lower bound graphs remains exactly the same as in Section 4.2. Furthermore, since the size of x didn't change asymptotically, the deduced lower bound from Section 4.3 doesn't change asymptotically. Observe that under the new definition of H^x, the diameter of H^x is at most $2t$. Hence, the diameter of $\widetilde{H}^x$ is at most $2tb$. It is not very hard to verify that the diameter of $G_{(x,y)}$ in this case is at most $2tb + 2b + 2$, which is constant for constant values of t and b.

References

1 Amir Abboud, Keren Censor-Hillel, and Seri Khoury. Near-linear lower bounds for distributed distance computations, even in sparse networks. In *Distributed Computing - 30th International Symposium, DISC*, volume 9888 of *Lecture Notes in Computer Science*, pages 29–42. Springer, 2016. doi:10.1007/978-3-662-53426-7_3.

2 Amir Abboud, Keren Censor-Hillel, Seri Khoury, and Christoph Lenzen. Fooling views: A new lower bound technique for distributed computations under congestion. *Distributed Computing*, pages 1–15, 2020.

3 Bertie Ancona, Keren Censor-Hillel, Yuval Efron, Mina Dalirrooyfard, and Virginia Vassilevska Williams. Distributed distance approximation. In *submission*, 2020.

4 Clark T. Benson. Minimal regular graphs of girths eight and twelve. *Canadian Journal of Mathematics*, 18:1091–1094, 1966. doi:10.4153/CJM-1966-109-8.

5 Karl Bringmann and Sebastian Krinninger. Brief announcement: A note on hardness of diameter approximation. In *31st International Symposium on Distributed Computing, DISC*, volume 91 of *LIPIcs*, pages 44:1–44:3. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.DISC.2017.44.

6 Keren Censor-Hillel, Michal Dory, Janne H. Korhonen, and Dean Leitersdorf. Fast approximate shortest paths in the congested clique. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC*, pages 74–83. ACM, 2019. doi:10.1145/3293611.3331633.

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

8 Keren Censor-Hillel, Telikepalli Kavitha, Ami Paz, and Amir Yehudayoff. Distributed construction of purely additive spanners. In *Proceedings of the 30th International Symposium on Distributed Computing, DISC*, pages 129–142, 2016.

9 Keren Censor-Hillel, Seri Khoury, and Ami Paz. Quadratic and near-quadratic lower bounds for the CONGEST model. In *31st International Symposium on Distributed Computing, DISC*, pages 10:1–10:16, 2017.

10 Andrew Drucker, Fabian Kuhn, and Rotem Oshman. On the power of the congested clique model. In *Proceedings of the 33rd ACM Symposium on Principles of Distributed Computing, PODC*, pages 367–376, 2014.

11 Yuval Efron, Ofer Grossman, and Seri Khoury. Beyond alice and bob: Improved inapproximability for maximum independent set in CONGEST. In Yuval Emek and Christian Cachin, editors, *PODC '20: ACM Symposium on Principles of Distributed Computing, Virtual Event*, pages 511–520. ACM, 2020. doi:10.1145/3382734.3405702.

12 Orr Fischer, Tzlil Gonen, Fabian Kuhn, and Rotem Oshman. Possibilities and impossibilities for distributed subgraph detection. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures, SPAA*, pages 153–162. ACM, 2018.

13 Sebastian Forster and Danupon Nanongkai. A faster distributed single-source shortest paths algorithm. In *59th IEEE Annual Symposium on Foundations of Computer Science, FOCS*, pages 686–697. IEEE Computer Society, 2018. doi:10.1109/FOCS.2018.00071.

14 Silvio Frischknecht, Stephan Holzer, and Roger Wattenhofer. Networks cannot compute their diameter in sublinear time. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 1150–1162, 2012.

15 Zoltán Füredi and Miklós Simonovits. *The History of Degenerate (Bipartite) Extremal Graph Problems*, pages 169–264. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. doi:10.1007/978-3-642-39286-3_7.

16 Stephan Holzer, David Peleg, Liam Roditty, and Roger Wattenhofer. Distributed 3/2-approximation of the diameter. In *Proceedings of the 28th International Symposium on Distributed Computing, DISC*, pages 562–564, 2014.

17 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*, volume 46 of *LIPIcs*, pages 6:1–6:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015. doi:10.4230/LIPIcs.OPODIS.2015.6.

18 Stephan Holzer and Roger Wattenhofer. Optimal distributed all pairs shortest paths and applications. In *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC*, pages 355–364, 2012.

19 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 *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA*, pages 439–441, 2016.

20 Eyal Kushilevitz and Noam Nisan. *Communication Complexity*. Cambridge University Press, New York, NY, USA, 1997.

21 Christoph Lenzen and David Peleg. Efficient distributed source detection with limited bandwidth. In *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC*, pages 375–382, 2013.

22 David Peleg. *Distributed Computing: A Locality-Sensitive Approach*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

23 David Peleg, Liam Roditty, and Elad Tal. Distributed algorithms for network diameter and girth. In *Proceedings of the 39th International Colloquium on Automata, Languages, and Programming, ICALP*, pages 660–672, 2012.

24 Alexander A. Razborov. On the distributional complexity of disjointness. *Theor. Comput. Sci.*, 106(2):385–390, 1992.

25 R. Singleton. On minimal graphs of maximum even girth. *Journal of Combinatorial Theory*, 1:306–332, December 1966. doi:10.1016/S0021-9800(66)80054-6.

26 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, STOC*, pages 209–213, 1979.

Twenty-Two New Approximate Proof Labeling Schemes

Yuval Emek
Technion – Israel Institute of Technology, Haifa, Israel
yemek@technion.ac.il

Yuval Gil
Technion – Israel Institute of Technology, Haifa, Israel
yuval.gil@campus.technion.ac.il

Abstract

Introduced by Korman, Kutten, and Peleg (Distributed Computing 2005), a *proof labeling scheme (PLS)* is a system dedicated to verifying that a given configuration graph satisfies a certain property. It is composed of a centralized *prover*, whose role is to generate a proof for yes-instances in the form of an assignment of labels to the nodes, and a distributed *verifier*, whose role is to verify the validity of the proof by local means and accept it if and only if the property is satisfied. To overcome lower bounds on the label size of PLSs for certain graph properties, Censor-Hillel, Paz, and Perry (SIROCCO 2017) introduced the notion of an *approximate proof labeling scheme (APLS)* that allows the verifier to accept also some no-instances as long as they are not "too far" from satisfying the property.

The goal of the current paper is to advance our understanding of the power and limitations of APLSs. To this end, we formulate the notion of APLSs in terms of *distributed graph optimization problems (OptDGPs)* and develop two generic methods for the design of APLSs. These methods are then applied to various classic OptDGPs, obtaining twenty-two new APLSs. An appealing characteristic of our APLSs is that they are all *sequentially efficient* in the sense that both the prover and the verifier are required to run in (sequential) polynomial time. On the negative side, we establish "combinatorial" lower bounds on the label size for some of the aforementioned OptDGPs that demonstrate the optimality of our corresponding APLSs. For other OptDGPs, we establish conditional lower bounds that exploit the sequential efficiency of the verifier alone (under the assumption that NP $\neq$ co-NP) or that of both the verifier and the prover (under the assumption that P $\neq$ NP, with and without the unique games conjecture).

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Theory of computation → Approximation algorithms analysis

Keywords and phrases proof labeling schemes, distributed graph problems, approximation algorithms

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.20

Related Version A full version of the paper is available at `https://arxiv.org/abs/2007.14307` [10].

Funding This work has been supported by an Israeli Science Foundation grant number 1016/17.

1 Introduction

1.1 Model

Consider a connected undirected graph $G = (V, E)$ and denote $n = |V|$ and $m = |E|$. For a node $v \in V$, we stick to the convention that $N(v) = \{u \mid (u, v) \in E\}$ denotes the set of v's *neighbors* in G. An edge is said to be *incident* on v if it connects between v and one of its neighbors.

In the realm of *distributed graph algorithms*, the nodes of graph $G = (V, E)$ are associated with processing units that operate in a decentralized fashion. We assume that node $v \in V$ distinguishes between its incident edges by means of *port numbers*, i.e., a bijection between

© Yuval Emek and Yuval Gil;
licensed under Creative Commons License CC-BY
34th International Symposium on Distributed Computing (DISC 2020).
Editor: Hagit Attiya; Article No. 20; pp. 20:1–20:14
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

the set of edges incident on v and the integers in $\{1, ..., |N(v)|\}$. Additional graph attributes, such as node ids, edge orientation, and edge and node weights, are passed to the nodes by means of an *input assignment* $\mathsf{I} : V \to \{0,1\}^*$ that assigns to each node $v \in V$, a bit string $\mathsf{I}(v)$, referred to as v's *local input*, that encodes the additional attributes of v and its incident edges. The nodes return their output by means of an *output assignment* $\mathsf{O} : V \to \{0,1\}^*$ that assigns to each node $v \in V$, a bit string $\mathsf{O}(v)$, referred to as v's *local output*. We often denote $G_{\mathsf{I},\mathsf{O}} = \langle G, \mathsf{I}, \mathsf{O}\rangle$ and $G_{\mathsf{I}} = \langle G, \mathsf{I}\rangle$ and refer to these tuples as an *input-output (IO) graph* and an *input graph*, respectively.[1]

A *distributed graph problem (DGP)* Π is a collection of IO graphs $G_{\mathsf{I},\mathsf{O}}$. In the context of a DGP Π, an input graph G_{I} is said to be *legal* (and the graph G and input assignment I are said to be *co-legal*) if there exists an output assignment O such that $G_{\mathsf{I},\mathsf{O}} \in \Pi$, in which case we say that O is a *feasible solution* for G_{I} (or simply for G and I). Given a DGP Π, we may slightly abuse the notation and write $G_{\mathsf{I}} \in \Pi$ to denote that G_{I} is legal.

A *distributed graph minimization problem (MinDGP)* (resp., *distributed graph maximization problem (MaxDGP)*) Ψ is a pair $\langle \Pi, f\rangle$, where Π is a DGP and $f : \Pi \to \mathbb{Z}$ is a function, referred to as the *objective function* of Ψ, that maps each IO graph $G_{\mathsf{I},\mathsf{O}} \in \Pi$ to an integer value $f(G_{\mathsf{I},\mathsf{O}})$.[2] Given a co-legal graph G and input assignment I, define

$$OPT_{\Psi}(G, \mathsf{I}) = \inf_{\mathsf{O}: G_{\mathsf{I},\mathsf{O}} \in \Pi} \{f(G_{\mathsf{I},\mathsf{O}})\}$$

if Ψ is a MinDGP; and

$$OPT_{\Psi}(G, \mathsf{I}) = \sup_{\mathsf{O}: G_{\mathsf{I},\mathsf{O}} \in \Pi} \{f(G_{\mathsf{I},\mathsf{O}})\}$$

if Ψ is a MaxDGP. We often use the general term *distributed graph optimization problem (OptDGP)* to refer to MinDGPs as well as MaxDGPs. Given a OptDGP $\Psi = \langle \Pi, f\rangle$ and co-legal graph G and input assignment I, the output assignment O is said to be an *optimal solution* for G_{I} (or simply for G and I) if O is a feasible solution for G_{I} and $f(G_{\mathsf{I},\mathsf{O}}) = OPT_{\Psi}(G, \mathsf{I})$.

Let us demonstrate our definitions through the example of the maximum weight matching problem in bipartite graphs, i.e., explaining how it fits into the framework of a MaxDGP $\Psi = \langle \Pi, f\rangle$. Given a graph $G = (V, E)$ and an input assignment I, the input graph G_{I} is legal (with respect to Π) if G is bipartite and I encodes an edge weight function $w : E \to \mathbb{Z}$. Formally, for every node $v \in V$, the local input assignment $\mathsf{I}(v)$ is set to be a vector, indexed by the port numbers of v, defined so that if edge $e = (u, u') \in E$ corresponds to ports $1 \leq i \leq |N(u)|$ and $1 \leq i' \leq |N(u')|$ at nodes u and u', respectively, then both the i-th entry in $\mathsf{I}(u)$ and the i'-th entry in $\mathsf{I}(u')$ hold the value $w(e)$. Given a legal input graph $G_{\mathsf{I}} \in \Pi$, the output assignment O is a feasible solution for G_{I} if O encodes a matching $\mu \subseteq E$ in G. Formally, the local output assignment $\mathsf{O}(v)$ is set to the port number corresponding to e if there exists an edge $e \in \mu$ incident on v; and to $\bot$ otherwise. The objective function f of Ψ is defined so that for an IO graph $G_{\mathsf{I},\mathsf{O}} \in \Pi$ with corresponding edge weight function $w_{\mathsf{I}} : E \to \mathbb{Z}$ and matching $\mu_{\mathsf{O}} \subseteq E$, the value of f is set to $f(G_{\mathsf{I},\mathsf{O}}) = \sum_{e \in \mu_{\mathsf{O}}} w_{\mathsf{I}}(e)$. Following this notation, a feasible solution O for co-legal G and I is optimal if and only if μ_{O} is a maximum weight matching in G with respect to the edge weight function w_{I}.

While the formulation introduced in the current section is necessary for the general definitions presented in Section 1.1.1 and the generic methods developed in Section 3, in [10], when considering IO graphs in the context of specific DGPs and OptDGPs, we often do

[1] Refer to Table 1 for a full list of the abbreviations used in this paper.

[2] We assume for simplicity that the images of the objective functions used in the context of this paper, are integral. Lifting this assumption and allowing for real numerical values would complicate some of the arguments, but it does not affect the validity of our results.

Table 1 A list of abbreviations.

Term	Abbreviation	Reference
input-output graph	IO graph	Section 1.1
distributed graph problem	DGP	Section 1.1
distributed graph minimization problem	MinDGP	Section 1.1
distributed graph maximization problem	MaxDGP	Section 1.1
distributed graph optimization problem	OptDGP	Section 1.1
gap proof labeling scheme	GPLS	Section 1.1.1
proof labeling scheme	PLS	Section 1.1.1
approximate proof labeling scheme	APLS	Section 1.1.1
decision proof labeling scheme	DPLS	Section 1.1.1
approximate decision proof labeling scheme	ADPLS	Section 1.1.1
verifiable centralized approximation	VCA	Section 3.2

not explicitly describe the input and output assignments, but rather take a more natural high-level approach. For example, in the context of the aforementioned maximum weight matching problem in bipartite graphs, we may address the input edge weight function and output matching directly without providing an explanation as to how they are encoded in the input and output assignments, respectively. The missing details would be clear from the context and could be easily completed by the reader.

1.1.1 Proof Labeling Schemes

In this section we present the notions of proof labeling schemes [26] and approximate proof labeling schemes [5] for OptDGPs and their decision variants. To unify the definitions of these notions, we start by introducing the notion of gap proof labeling schemes based on the following definition.

A *configuration graph* $G_S = \langle G, S \rangle$ is a pair consisting of a graph $G = (V, E)$ and a function $S : V \to \{0,1\}^*$ assigning a bit string $S(v)$ to each node $v \in V$. In particular, an input graph G_{I} is a configuration graph, where $S(v) = \mathsf{I}(v)$, and an IO graph $G_{\mathsf{I},\mathsf{O}}$ is a configuration graph, where $S(v) = \mathsf{I}(v) \cdot \mathsf{O}(v)$.

Fix some universe $\mathcal{U}$ of configuration graphs. A *gap proof labeling scheme (GPLS)* is a mechanism designed to distinguish the configuration graphs in a *yes-family* $\mathcal{F}_Y \subset \mathcal{U}$ from the configuration graphs in a *no-family* $\mathcal{F}_N \subset \mathcal{U}$, where $\mathcal{F}_Y \cap \mathcal{F}_N = \emptyset$. This is done by means of a (centralized) *prover* and a (distributed) *verifier* that play the following roles: Given a configuration graph $G_S \in \mathcal{U}$, if $G_S \in \mathcal{F}_Y$, then the prover assigns a bit string $L(v)$, called the *label* of v, to each node $v \in V$. Let $L^N(v)$ be the vector of labels assigned to v's neighbors. The verifier at node $v \in V$ is provided with the 3-tuple $\langle S(v), L(v), L^N(v) \rangle$ and returns a Boolean value $\varphi(v)$.

We say that the verifier *accepts* G_S if $\varphi(v) = \texttt{True}$ for all nodes $v \in V$; and that the verifier *rejects* G_S if $\varphi(v) = \texttt{False}$ for at least one node $v \in V$. The GPLS is said to be *correct* if the following requirements hold for every configuration graph $G_S \in \mathcal{U}$:

▶ **R1.** *If $G_S \in \mathcal{F}_Y$, then the prover produces a label assignment $L : V \to \{0,1\}^*$ such that the verifier accepts G_S.*

▶ **R2.** *If $G_S \in \mathcal{F}_N$, then for any label assignment $L : V \to \{0,1\}^*$, the verifier rejects G_S.*

We emphasize that no requirements are made for configuration graphs $G_S \in \mathcal{U} \setminus (\mathcal{F}_Y \cup \mathcal{F}_N)$; in particular, the verifier may either accept or reject these configuration graphs (the same holds for configuration graphs that do not belong to the universe $\mathcal{U}$).

The performance of a GPLS is measured by means of its *proof size* defined to be the maximum length of a label $L(v)$ assigned by the prover to the nodes $v \in V$ assuming that $G_S \in \mathcal{F}_Y$. We say that GPLS admits a *sequentially efficient prover* if for any configuration graph $G_S \in F_Y$, the sequential runtime of the prover is polynomial in the number of bits used to encode G_S; and that it admits a *sequentially efficient verifier* if the sequential runtime of the verifier in node $v \in V$ is polynomial in $|S(v)|$, $|L(v)|$, and $\sum_{u \in N(v)} |L(u)|$. The GPLS is called *sequentially efficient* if both its prover and verifier are sequentially efficient.

Proof Labeling Schemes for OptDGPs. Consider some OptDGP $\Psi = \langle \Pi, f \rangle$ and let $\mathcal{U} = \{G_{\mathsf{I},\mathsf{O}} \mid G_{\mathsf{I}} \in \Pi\}$. A *proof labeling scheme (PLS)* for Ψ is defined as a GPLS over $\mathcal{U}$ by setting the yes-family to be

$$\mathcal{F}_Y = \{G_{\mathsf{I},\mathsf{O}} \in \Pi \mid f(G_{\mathsf{I},\mathsf{O}}) = OPT_\Psi(G, \mathsf{I})\}$$

and the no-family to be $\mathcal{F}_N = \mathcal{U} \setminus \mathcal{F}_Y$. In other words, a PLS for Ψ determines for a given IO graph $G_{\mathsf{I},\mathsf{O}} \in \mathcal{U}$ whether the output assignment $\mathsf{O} : V \to \{0,1\}^*$ is an optimal solution (which means in particular that it is a feasible solution) for the co-legal graph $G = (V, E)$ and input assignment $\mathsf{I} : V \to \{0,1\}^*$.

In the realm of OptDGPs, it is natural to relax the definition of a PLS so that it may also accept feasible solutions that only approximate the optimal ones. Specifically, given an approximation parameter $\alpha \geq 1$, an *α-approximate proof labeling scheme (α-APLS)* for a OptDGP $\Psi = \langle \Pi, f \rangle$ is defined in the same way as a PLS for Ψ with the sole difference that the no-family is defined by setting

$$\mathcal{F}_N = \begin{cases} \mathcal{U} \setminus \{G_{\mathsf{I},\mathsf{O}} \in \Pi \mid f(G_{\mathsf{I},\mathsf{O}}) \leq \alpha \cdot OPT_\Psi(G, \mathsf{I})\}\ , & \text{if } \Psi \text{ is a MinDGP} \\ \mathcal{U} \setminus \{G_{\mathsf{I},\mathsf{O}} \in \Pi \mid f(G_{\mathsf{I},\mathsf{O}}) \geq OPT_\Psi(G, \mathsf{I})/\alpha\}\ , & \text{if } \Psi \text{ is a MaxDGP} \end{cases} .$$

Decision Proof Labeling Schemes for OptDGPs. Consider some MinDGP (resp., MaxDGP) $\Psi = \langle \Pi, f \rangle$ and let $\mathcal{U} = \{G_{\mathsf{I}} \mid G_{\mathsf{I}} \in \Pi\}$. A *decision proof labeling scheme (DPLS)* for Ψ and a parameter $k \in \mathbb{Z}$ is defined as a GPLS over $\mathcal{U}$ by setting the yes-family to be

$$\mathcal{F}_Y = \begin{cases} \{G_{\mathsf{I}} \in \Pi \mid OPT_\Psi(G, \mathsf{I}) \geq k\}\ , & \text{if } \Psi \text{ is a MinDGP} \\ \{G_{\mathsf{I}} \in \Pi \mid OPT_\Psi(G, \mathsf{I}) \leq k\}\ , & \text{if } \Psi \text{ is a MaxDGP} \end{cases}$$

and the no-family to be $\mathcal{F}_N = \mathcal{U} \setminus \mathcal{F}_Y$. In other words, given an input graph $G_{\mathsf{I}} \in \Pi$, a DPLS for Ψ and k decides if $f(G_{\mathsf{I},\mathsf{O}}) \geq k$ (resp., $f(G_{\mathsf{I},\mathsf{O}}) \leq k$) for every feasible output assignment $\mathsf{O} : V \to \{0,1\}^*$. Notice that while PLSs address the task of verifying the optimality of a given output assignment O, that is, verifying that no output assignment admits an objective value smaller (resp., larger) than $f(G_{\mathsf{I},\mathsf{O}})$, in DPLSs, the output assignment O is not specified and the task is to verify that no output assignment admits an objective value smaller (resp., larger) than the parameter k, provided as part of the DPLS task.

Similarly to PLSs, the definition of DPLS admits a natural relaxation. Given an approximation parameter $\alpha \geq 1$, an *α-approximate decision proof labeling scheme (α-ADPLS)* for a OptDGP $\Psi = \langle \Pi, f \rangle$ and a parameter $k \in \mathbb{Z}$ is defined in the same way as a DPLS for Ψ and k with the sole difference that the no-family is defined by setting

$$\mathcal{F}_N = \begin{cases} \mathcal{U} \setminus \{G_{\mathsf{I}} \in \Pi \mid OPT_\Psi(G, \mathsf{I}) \geq k/\alpha\}\ , & \text{if } \Psi \text{ is a MinDGP} \\ \mathcal{U} \setminus \{G_{\mathsf{I}} \in \Pi \mid OPT_\Psi(G, \mathsf{I}) \leq \alpha \cdot k\}\ , & \text{if } \Psi \text{ is a MaxDGP} \end{cases} .$$

We often refer to an α-ADPLS without explicitly mentioning its associated parameter k; this should be interpreted with a universal quantifier over all parameters $k \in \mathbb{Z}$.

1.2 Related Work and Discussion

Distributed verification is the task of locally verifying a global property of a given configuration graph by means of a centralized prover and a distributed verifier. Various models for distributed verification have been introduced in the literature including the PLS model [26] as defined in Section 1.1.1, the *locally checkable proofs (LCP)* model [18], and the distributed complexity class *non-deterministic local decision (NLD)* [15, 3]. Refer to [12] for a comprehensive survey on the topic of distributed verification.

The current paper focuses on the PLS (and DPLS) model. This model was introduced by Korman, Kutten, and Peleg in [26] and has been extensively studied since then, see, e.g., [25, 4, 29, 13, 30, 14, 11]. A specific family of tasks that attracted a lot of attention in this regard is that of designing PLSs for classic optimization problems. Papers on this topic include [25], where a PLS for minimum spanning tree is shown to have a proof size of $O(\log n \log W)$, where W is the maximum weight, and [18], where a PLS for maximum weight matching in bipartite graphs is shown to have a proof size of $O(\log W)$.

In parallel, numerous researchers focused on establishing impossibility results for PLSs and DPLSs, usually derived from non-deterministic communication complexity lower bounds [27]. Such results are provided, e.g., in [2], where a proof size of $\tilde{\Omega}(n^2)$ is shown to be required for many classic optimization problems, and in [18], where an $\Omega(n^2/\log n)$ lower bound is established on the proof size of DPLSs for the problem of deciding if the chromatic number is larger than 3. For the minimum spanning tree problem, the authors of [25] proved that their $O(\log n \log W)$ upper bound on the proof size is asymptotically optimal, relying on direct combinatorial arguments.

The lower bounds on the proof size of PLSs (and DPLSs) for some optimization problems have motivated the authors of [5] to introduce the APLS (and ADPLS) notion as a natural relaxation thereof. This motivation is demonstrated by the task of verifying that the unweighted diameter of a given graph is at most k: As shown in [5], the diameter task admits a large gap between the required proof size of a DPLS, shown to be $\Omega(n/k)$, and the proof sizes of (3/2)-ADPLS and 2-ADPLS shown to be $O(\sqrt{n}\log^2 n)$ and $O(\log n)$, respectively. To the best of our knowledge, APLSs (and ADPLSs) have not been studied otherwise until the current paper.

One of the generic methods developed in the current paper for the design of APLSs for an abstract OptDGP relies on a *primal dual* approach applied to the linear program that encodes the OptDGP, after relaxing its integrality constraints (see Section 3.1). This can be viewed as a generalization of a similar approach used in the literature for concrete OptDGPs. Specifically, this primal dual approach is employed in [18] to obtain their PLS for maximum weight matching in bipartite graphs with a proof size of $O(\log W)$. A similar technique is used by the authors of [5] to achieve a 2-APLS for maximum weight matching in general graphs with the same proof size.

While most of the PLS literature (including the current work) focuses on deterministic schemes, an interesting angle that has been studied recently is randomization in distributed proofs, i.e., allowing the verifier to reach its decision in a randomized fashion. The notion of *randomized proof labeling schemes* was introduced in [17], where the strength of randomization in the PLS model is demonstrated by a universal scheme that enables one to reduce the amount of required communication in a PLS exponentially by allowing a (probabilistic) one-sided error. Another interesting generalization of PLSs is the *distributed interactive proof* model, introduced recently in [24] and studied further in [28, 7, 16].

On Sequential Efficiency. In this paper, we focus on sequentially efficient schemes, restricting the prover and verifier to "reasonable computations". We argue that beyond the interesting theoretical implications of this restriction (see Section 1.3), it also carries practical justifications: A natural application of PLSs is found in *local checking* for self-stabilizing algorithms [1], where the verifier's role is played by the detection module and the prover is part of the correction module [26]. Any attempt to implement these modules in practice clearly requires sequential efficiency on behalf of both the verifier and the prover (although, for the latter, the sequential efficiency condition alone is not sufficient as the correction module is also distributed).

While most of the PLSs presented in previous papers are naturally sequentially efficient, there are a few exceptions. One example of a scheme that may require intractable computations on the verifier side is the universal PLS presented in [26] that enables the verification of any decidable graph property with a label size of $O(n^2)$ simply by encoding the entire structure of the graph within the label. A PLS that inherently relies on sequentially inefficient prover can be found, e.g., in [18], where a scheme is constructed to decide if the graph contains a Hamiltonian cycle.

1.3 Our Contribution

Our goal in this paper is to explore the power and limitations of APLSs and ADPLS for OptDGPs. We start by developing two generic methods: a primal dual method for the design of sequentially efficient APLSs that expands and generalizes techniques used by Göös and Suomela [18] and Censor-Hillel, Paz, and Perry [5]; and a method that exploits the local properties of centralized approximation algorithms for the design of sequentially efficient ADPLSs. Next, we establish black-box reductions between APLSs and ADPLSs for certain families of OptDGPs. Based (mainly) on these generic methods and reductions, we design a total of twenty-two new sequentially efficient APLSs and ADPLSs for various classic optimization problems; refer to Tables 2 and 3 for a summary of these results.

On the negative side, we establish an $\Omega(\log \kappa)$ lower bound on the proof size of a $\frac{\kappa+1}{\kappa}$-APLS for maximum b-matching (in fact, this lower bound applies even for the simpler case of maximum matching) and minimum edge cover in graphs of odd-girth $2\kappa + 1$; and an $\Omega(\log n)$ lower bound on the proof size of a PLS for minimum edge cover in odd rings. These lower bounds, that rely on combinatorial arguments and hold regardless of sequential efficiency, match the proof size established in our corresponding APLSs for these OptDGPs, thus proving their optimality.

Additional lower bounds are established under the restriction of the verifier and/or prover to sequentially efficient computations, based on hardness assumptions in (sequential) computational complexity theory. Consider a OptDGP Ψ that corresponds to an optimization problem that is NP-hard to approximate within $\alpha \geq 1$. We first note that under the assumption that NP $\neq$ co-NP, the yes-families of both an α-APLS for Ψ and an α-ADPLS for Ψ (with some parameter $k \in \mathbb{Z}$) are languages in the complexity class co-NP $\setminus$ NP. Therefore, restricting the verifier to sequentially efficient computations implies that Ψ admits neither an α-APLS, nor an α-ADPLS, with a polynomial proof size. This provides additional motivation for the study of APLSs and ADPLSs over their exact counterparts.

Furthermore, the (weaker) assumption that P $\neq$ NP suffices to rule out the existence of α-ADPLS for Ψ when both the verifier and prover are required to be sequentially efficient. This is due to the fact that the yes-family of an α-ADPLS for Ψ (with some parameter $k \in \mathbb{Z}$) is a co-NP complete language, combined with the trivial observation that any sequentially efficient GPLS can be simulated by a centralized algorithm in polynomial time. We note

Table 2 α-APLS results and proof sizes. Refer to [10, Table 2] for references to these results.

Problem	Graph family	α	Proof size
minimum edge cover	odd-girth $\geq 2\kappa+1$	$(\kappa+1)/\kappa$	$O(\log \kappa)$
	odd-girth $\geq 2\kappa+1$	$(\kappa+1)/\kappa$	$\Omega(\log \kappa)$
	odd rings	1	$O(\log n)$
	odd rings	1	$\Omega(\log n)$
	bipartite	1	1
maximum b-matching	odd-girth $\geq 2\kappa+1$	$(\kappa+1)/\kappa$	$O(\log \kappa)$
	odd-girth $\geq 2\kappa+1$	$(\kappa+1)/\kappa$	$\Omega(\log \kappa)$
	bipartite	1	1
min-weight vertex cover	any	2	$O(\log n + \log W)$
min-weight dominating set	any	$\ln(n)+1$	$O(\log n + \log W)$
traveling salesperson	metric	2	$O(\log n + \log W)$
minimum Steiner tree	metric	2	$O(\log n + \log W)$
maximum flow	directed	1	1
max-weight cut	any	2	1

Table 3 α-ADPLS results and proof sizes. Refer to [10, Table 3] for references to these results.

Problem	Graph family	α	Proof size
minimum edge cover	odd-girth $\geq 2\kappa+1$	$(\kappa+1)/\kappa$	$O(\log n)$
	odd rings	1	$O(\log n)$
	bipartite	1	$O(\log n)$
maximum b-matching	odd-girth $\geq 2\kappa+1$	$(\kappa+1)/\kappa$	$O(\log n + \log W)$
	bipartite	1	$O(\log n + \log W)$
min-weight vertex cover	any	2	$O(\log n + \log W)$
min-weight dominating set	any	$\ln(n)+1$	$O(\log n + \log W)$
traveling salesperson	metric	2	$O(\log n + \log W)$
minimum Steiner tree	metric	2	$O(\log n + \log W)$
maximum flow	directed	1	$O(\log n + \log W)$
max-weight cut	any	2	$O(\log n + \log W)$

that most of the OptDGPs considered in this paper correspond to NP-hard optimization problems; refer to Table 4 for their known inapproximability results with and without the unique games conjecture [21].

1.4 Paper's Organization

The rest of the paper is organized as follows. Following some preliminaries presented in Section 2, our generic methods for the design of APLSs and ADPLSs are developed in Section 3. The reductions between APLSs and ADPLSs are presented in Section 4. Finally, the bounds we establish for concrete OptDGPS are established in [10].

2 Preliminaries

Linear Programming and Duality. A *linear program (LP)* consists of a linear objective function that one wishes to optimize (i.e., minimize or maximize) subject to linear inequality constraints. The *standard form* of a minimization (resp., maximization) LP is $\min\{\mathbf{c}^{\mathrm{T}}\mathbf{x} \mid$

Table 4 Known inapproximability bounds with and without the unique games conjecture.

Problem	Inapproximability	Inapproximability w. UGC
min-weight vertex cover	$1.36 - \varepsilon$ [8]	$2 - \varepsilon$ [23]
min-weight dominating set	$(1 - o(1)) \cdot \ln n$ [9]	
metric traveling salesperson	$123/122 - \varepsilon$ [20]	
metric minimum Steiner tree	$96/95 - \varepsilon$ [6]	
max-weight cut	$1.063 - \varepsilon$ [19]	$1.139 - \varepsilon$ [22]

$\mathbf{A}\mathbf{x} \geq \mathbf{b} \wedge \mathbf{x} \geq \mathbf{0}\}$ (resp., $\max\{\mathbf{c}^{\mathrm{T}}\mathbf{x} \mid \mathbf{A}\mathbf{x} \leq \mathbf{b} \wedge \mathbf{x} \geq \mathbf{0}\}$), where $\mathbf{x} = \{x_j\} \in \mathbb{R}^{\ell}$ is a vector of variables and $\mathbf{A} = \{a_{i,j}\} \in \mathbb{R}^{k \times \ell}$, $\mathbf{b} = \{b_i\} \in \mathbb{R}^k$, and $\mathbf{c} = \{c_j\} \in \mathbb{R}^{\ell}$ are a matrix and vectors of coefficients. An *integer linear program (ILP)* is a LP augmented with integrality constraints. In [10], we formulate OptDGPs as LPs and ILPs. In the latter case, we often turn to a *LP relaxation* of the problem, i.e., a LP obtained from an ILP by relaxing its integrality constraints.

Every LP admits a corresponding *dual program* (in this context, we refer to the original LP as the *primal program*). Specifically, for a minimization (resp., maximization) LP in standard form, its dual is a maximization (resp., minimization) LP, formulated as $\max\{\mathbf{b}^{\mathrm{T}}\mathbf{y} \mid \mathbf{A}^{\mathrm{T}}\mathbf{y} \leq \mathbf{c} \wedge \mathbf{y} \geq \mathbf{0}\}$ (resp., $\min\{\mathbf{b}^{\mathrm{T}}\mathbf{y} \mid \mathbf{A}^{\mathrm{T}}\mathbf{y} \geq \mathbf{c} \wedge \mathbf{y} \geq \mathbf{0}\}$).

LP duality has the following useful properties. Let $\mathbf{x}$ and $\mathbf{y}$ be feasible solutions to the primal and dual programs, respectively. The *weak duality* theorem states that $\mathbf{c}^{\mathrm{T}}\mathbf{x} \geq \mathbf{b}^{\mathrm{T}}\mathbf{y}$ (resp., $\mathbf{c}^{\mathrm{T}}\mathbf{x} \leq \mathbf{b}^{\mathrm{T}}\mathbf{y}$). The *strong duality* theorem states that $\mathbf{x}$ and $\mathbf{y}$ are optimal solutions to the primal and dual programs, respectively, if and only if $\mathbf{c}^{\mathrm{T}}\mathbf{x} = \mathbf{b}^{\mathrm{T}}\mathbf{y}$. The *relaxed complementary slackness* conditions are stated as follows, for given parameters $\beta, \gamma \geq 1$.

- Primal relaxed complementary slackness:
  For every primal variable x_j, if $x_j > 0$, then $c_j/\beta \leq \sum_{i=1}^{k} a_{ij} y_i \leq c_j$ (resp., $c_j \leq \sum_{i=1}^{k} a_{ij} y_i \leq \beta \cdot c_j$).
- Dual relaxed complementary slackness:
  For every dual variable y_i, if $y_i > 0$, then $b_i \leq \sum_{j=1}^{\ell} a_{ij} x_j \leq \gamma \cdot b_i$ (resp., $b_i/\gamma \leq \sum_{j=1}^{\ell} a_{ij} x_j \leq b_i$).

If the (primal and dual) relaxed complementary slackness conditions hold, then it is guaranteed that $\mathbf{c}^{\mathrm{T}}\mathbf{x} \leq \beta \cdot \gamma \cdot \mathbf{b}^{\mathrm{T}}\mathbf{y}$ (resp., $\mathbf{c}^{\mathrm{T}}\mathbf{x} \geq \frac{1}{\beta \cdot \gamma} \cdot \mathbf{b}^{\mathrm{T}}\mathbf{y}$) which, combined with the aforementioned weak duality theorem, implies that $\mathbf{x}$ approximates an optimal primal solution by a multiplicative factor of $\beta \cdot \gamma$. Moreover, the relaxed complementary slackness conditions with parameters $\beta = \gamma = 1$, often referred to simply as the *complementary slackness* conditions, hold if and only if $\mathbf{x}$ and $\mathbf{y}$ are optimal.

Let $\Psi = \langle \Pi, f \rangle$ be a OptDGP that can be represented as an ILP. Let P be its LP relaxation and D the dual LP of P. Given parameters $\beta, \gamma \geq 1$, we say that Ψ is *(β, γ)-fitted* if for any optimal (integral) solution $\mathbf{x}$ for the ILP corresponding to P, there exists a feasible solution $\mathbf{y}$ for D such that the relaxed primal and dual complementary slackness conditions hold for $\mathbf{x}$ and $\mathbf{y}$ with parameters β and γ, respectively.

Comparison Schemes. Let $\mathcal{U}$ be the universe of IO graphs $G_{\mathsf{I},\mathsf{O}}$ where $\mathsf{I} : V \to \{0,1\}^*$ is an input assignment that encodes a unique id represented using $O(\log n)$ bits for each node $v \in V$ (possibly among other input components). For a function $h : \{0,1\}^* \times \{0,1\}^* \to \mathbb{R}$ and parameter $k \in \mathbb{Z}$, an (h,k)-*comparison scheme* is a mechanism designed to decide if

$\sum_{v \in V} h(\mathsf{I}(v), \mathsf{O}(v)) \geq k$ for a given IO graph $G_{\mathsf{I},\mathsf{O}} \in \mathcal{U}$. Formally, an (h,k)-comparison scheme is defined as a GPLS over $\mathcal{U}$ by setting the yes-family to be $\mathcal{F}_Y = \{G_{\mathsf{I},\mathsf{O}} \in \mathcal{U} \mid \sum_{v \in V} h(\mathsf{I}(v), \mathsf{O}(v)) \geq k\}$ and the no family to be $\mathcal{F}_N = \mathcal{U} \setminus \mathcal{F}_Y$. Notice that the task of deciding if $\sum_{v \in V} h(\mathsf{I}(v), \mathsf{O}(v)) \leq k$ can be achieved by a $(-h,-k)$-comparison scheme, where $-h : \{0,1\}^* \times \{0,1\}^* \to \mathbb{R}$ is defined by setting $-h(a,b) = -1 \cdot h(a,b)$ for every $a, b \in \{0,1\}^*$.

The following lemma has been established by Korman et al. [26, Lemma 4.4].

▶ **Lemma 2.1.** *Given a function $h : \{0,1\}^* \times \{0,1\}^* \to \mathbb{R}$ that is computable in polynomial time and an integer $k \in \mathbb{Z}$, there exists a sequentially efficient (h,k)-comparison scheme with proof size $O(\log n + H)$, where H is the maximal number of bits required to represent $h(\mathsf{I}(v), \mathsf{O}(v))$ for any $v \in V$.*

Additional Definitions. A *feasibility scheme* for a DGP Π is a GPLS over the universe $\mathcal{U} = \{G_{\mathsf{I},\mathsf{O}} \mid G_{\mathsf{I}} \in \Pi\}$ with the yes-family $\mathcal{F}_Y = \Pi$ and the no-family $\mathcal{U} \setminus \mathcal{F}_Y$. The *odd-girth* of a graph $G = (V, E)$ is the length of the shortest odd cycle contained in G.

3 Methods

In this section, we present two generic methods that facilitate the design of sequentially efficient APLSs and ADPLSs with small proof sizes for many OptDGPs. These methods are used in most of the results established later on in [10].

3.1 The Primal Dual Method

LP duality theory can be a useful tool in the design of a $(\beta \cdot \gamma)$-APLS for a (β,γ)-fitted OptDGP Ψ (as shown in [5, 18]). The main idea of this approach is to use the relaxed complementary slackness conditions to verify that the output assignment $\mathsf{O} : V \to \{0,1\}^*$ of a given IO graph $G_{\mathsf{I},\mathsf{O}}$ is approximately optimal for G and I with respect to Ψ. Specifically, the prover provides the verifier with a proof that there exists a feasible dual solution $\mathbf{y}$ within a multiplicative factor of $\beta \cdot \gamma$ from the primal solution $\mathbf{x}$ derived from the output assignment O; the verifier then verifies the primal and dual feasibility of $\mathbf{x}$ and $\mathbf{y}$, respectively, as well as their relaxed complementary slackness conditions.

We take a particular interest in the following family of OptDGPs. Consider a OptDGP $\Psi = \langle \Pi, f \rangle$ that can be represented by an ILP that admits a LP relaxation P whose matrix form is given by the variable vector $\mathbf{x} = \{x_j\} \in \mathbb{R}^{\ell}$ and coefficient matrix and vectors $\mathbf{A} = \{a_{i,j}\} \in \mathbb{R}^{k \times \ell}$, $\mathbf{b} = \{b_i\} \in \mathbb{R}^{k}$, and $\mathbf{c} = \{c_j\} \in \mathbb{R}^{\ell}$. We say that Ψ is *locally verifiable* if for every IO graph $G_{\mathsf{I},\mathsf{O}} \in \Pi$, there exist mappings $v : [k] \to V$ and $e : [\ell] \to E$ that satisfy the following conditions: (1) $a_{i,j} = 0$ for every $i \in [k]$ and $j \in [\ell]$ such that $e(j)$ is not incident on $v(i)$; (2) the variable x_j is encoded in the local output $\mathsf{O}(u)$ of node $u \in V$ for every $j \in [k]$ such that $e(j)$ is incident on u; and (3) the coefficients $a_{i,j}$, $a_{i',j}$, b_i, and c_j are either universal constants or encoded in the local input $\mathsf{I}(u)$ of node $u \in V$ for every $i, i' \in [k]$ and $j \in [\ell]$ such that $v(i) = u$, $v(i') = u'$, and $e(j) = (u, u')$.

The *primal dual* method facilitates the design of an α-APLS, $\alpha = \beta \cdot \gamma$, for a (β,γ)-fitted and locally verifiable OptDGP $\Psi = \langle \Pi, f \rangle$ whose goal is to determine for a given IO graph $G_{\mathsf{I},\mathsf{O}}$ if the output assignment $\mathsf{O} : V \to \{0,1\}^*$ is an optimal (feasible) solution for the co-legal G and I or α-far from being an optimal solution. Let $\mathbf{x}$ be the primal variable vector encoded in the output assignment O. If O is an optimal solution for G and I, then the prover uses a sequential algorithm to generate a feasible dual variable vector $\mathbf{y}$ such that $\mathbf{x}$ and $\mathbf{y}$ meet the relaxed complementary slackness conditions with parameters β and γ (such a dual solution $\mathbf{y}$

exists as Ψ is (β, γ)-fitted). The label assignment $L : V \to \{0,1\}^*$ constructed by the prover assigns to each node $u \in V$, a label $L(u)$ that encodes the vector $\mathbf{y}(u) = \langle y_i \mid v(i) = u \rangle$ of dual variables mapped to u in the dual variable vector $\mathbf{y}$.

Consider some node $u \in V$ of the given IO graph $G_{\mathsf{I},\mathsf{O}}$. The verifier at node u extracts (i) the vector $\mathbf{x}(u) = \langle x_j \mid u \in e(j) \rangle$ of primal variables mapped to edges incident on u from the local output $\mathsf{O}(u)$; (ii) the vector $\mathbf{y}(u)$ of dual variables mapped to u from the label $L(v)$; (iii) the vector $\mathbf{y}^N(u) = \langle y_i \mid v(i) \in N(u) \rangle$ of dual variables mapped to u's neighbors from the label vector $L^N(u)$; and (iv) the vectors $\mathbf{a}(u) = \langle a_{i,j} \mid u \in e(j) \rangle$, $\mathbf{b}(u) = \langle b_i \mid v(i) = u \rangle$, and $\mathbf{c}(u) = \langle c_j \mid u \in e(j) \rangle$ of coefficients mapped to u and the edges incident on u from the local input $\mathsf{I}(u)$.

The verifier at node $u \in V$ then proceeds as follows: (1) using $\mathbf{x}(u)$, $\mathbf{a}(u)$, and $\mathbf{b}(u)$, the verifier verifies that the primal constraints that correspond to rows $i \in [k]$ such that $v(i) = u$ are satisfied; (2) using $\mathbf{y}(u)$, $\mathbf{y}^N(u)$, $\mathbf{a}(u)$, and $\mathbf{c}(u)$, the verifier verifies that the dual constraints that correspond to columns $j \in [\ell]$ such that $u \in e(j)$ are satisfied; (3) using $\mathbf{x}(u)$, $\mathbf{y}(u)$, $\mathbf{y}^N(u)$, $\mathbf{a}(u)$, and $\mathbf{c}(u)$, the verifier verifies that the primal relaxed complementary slackness conditions that correspond to primal variables x_j such that $u \in e(j)$ hold with parameter β; and (4) using $\mathbf{x}(u)$, $\mathbf{y}(u)$, $\mathbf{a}(u)$, and $\mathbf{b}(u)$, the verifier verifies that the dual relaxed complementary slackness conditions that correspond to dual variables y_i such that $v(i) = u$ hold with parameter γ. If all four conditions are satisfied, then the verifier at node u returns `True`; otherwise, it returns `False`. Put together, the verifier accepts the IO graph $G_{\mathsf{I},\mathsf{O}}$ if and only if $\mathbf{x}$ and $\mathbf{y}$ are feasible primal and dual solutions that satisfy the primal and dual relaxed complementary slackness conditions with parameters β and γ, respectively.

To establish the correctness of the α-APLS, notice first that the primal constraints are satisfied if and only if O is a feasible solution for G and I. Assuming that primal constraints are satisfied, if O is an optimal solution for G and I, then the fact that Ψ is (β, γ)-fitted implies that the verifier generates a feasible dual solution $\mathbf{y}$ such that the primal and dual relaxed complementary slackness conditions are satisfied with parameters β and γ. Conversely, If $\mathbf{y}$ is a feasible dual solution and the primal and dual relaxed complementary slackness conditions are satisfied with parameters β and γ, then $\mathbf{x}$ approximates the optimal primal (fractional) solution within an approximation bound of $\beta \cdot \gamma = \alpha$, hence O approximates $OPT_\Psi(G, \mathsf{I})$ within the same approximation bound.

The proof size of a $(\beta \cdot \gamma)$-APLS for a (β, γ)-fitted and locally verifiable OptDGP Ψ, designed by means of the primal dual method, is the maximum number of bits required to encode the vector $\mathbf{y}(u)$ of dual variables mapped to a node $u \in V$. Let r be the range of possible values assigned by the prover to a dual variable y_i. We aim for schemes that minimize r. Particularly, for OptDGPs where the number of primal constraints mapped to each node is bounded by a constant, this results in a $(\beta \cdot \gamma)$-APLS with a proof size of $O(\log r)$.

In [10] we present APLSs that are obtained using the primal dual method. We note that for all these APLSs, both the prover and verifier run in polynomial sequential time, thus yielding sequentially efficient APLSs.

3.2 The Verifiable Centralized Approximation Method

Consider some OptDGP $\Psi = \langle \Pi, f \rangle$. We say that Ψ is *identified* if the input assignment $\mathsf{I} : V \to \{0,1\}^*$ encodes a unique id represented using $O(\log n)$ bits at each node $v \in V$ (possibly among other input components) for every IO graph $G_{\mathsf{I},\mathsf{O}}$.

We say that Ψ is *decomposable* if there exists a function $\lambda : \{0,1\}^* \times \{0,1\}^* \to \mathbb{R}$, often referred to as a *decomposition function*, such that $f(G_{\mathsf{I},\mathsf{O}}) = \sum_{v \in V} \lambda(\mathsf{I}(v), \mathsf{O}(v))$ for every IO graph $G_{\mathsf{I},\mathsf{O}} \in \Pi$ (cf. the notion of semi-group functions in [26]). Given an input and

output assignments $\mathsf{I}, \mathsf{O} : V \to \{0,1\}^*$, let $\lambda(\mathsf{I}, \mathsf{O}) = \sum_{v \in V} \lambda(\mathsf{I}(v), \mathsf{O}(v))$ denote the sum of the decomposition function values $\lambda(\mathsf{I}(v), \mathsf{O}(v))$ over all nodes $v \in V$. Notice that the decomposition function is well defined for all bit string pairs; in particular, the definition of $\lambda(\mathsf{I}, \mathsf{O})$ does not require that the output assignment O is a feasible solution for the graph G and the input assignment I.

Let $\Psi = \langle \Pi, f \rangle$ be a decomposable MinDGP (resp., MaxDGP) with a decomposition function $\lambda : \{0,1\}^* \times \{0,1\}^* \to \mathbb{R}$. Given a legal input graph $G_{\mathsf{I}} \in \Pi$ and a parameter $\alpha \geq 1$, we say that a (not necessarily feasible) output assignment $\mathsf{A} : V \to \{0,1\}^*$ is a *decomposable α-approximation* for G and I if $OPT_{\Psi}(G, \mathsf{I}) \leq \lambda(\mathsf{I}, \mathsf{A}) \leq \alpha \cdot OPT_{\Psi}(G, \mathsf{I})$ (resp., $OPT_{\Psi}(G, \mathsf{I})/\alpha \leq \lambda(\mathsf{I}, \mathsf{A}) \leq OPT_{\Psi}(G, \mathsf{I})$).

Fix some identified decomposable MinDGP (resp., MaxDGP) $\Psi = \langle \Pi, f \rangle$ with a decomposition function $\lambda : \{0,1\}^* \times \{0,1\}^* \to \mathbb{R}$. The *verifiable centralized approximation (VCA)* method facilitates the design of an α-ADPLS for Ψ whose goal is to determine for a given legal input graph $G_{\mathsf{I}} \in \Pi$ and some parameter $k \in \mathbb{Z}$ if every output assignment yields an objective value of at least (resp., at most) k or if there exists an output assignment that yields an objective value smaller than k/α (resp., larger than $\alpha \cdot k$). The ADPLSs designed by means of the VCA method are composed of two verification tasks, namely, the *approximation* task and the *comparison* task, so that the verifier accepts G_{I} if and only if both verification tasks accept. The label $L(v) = \langle L_{\mathtt{approx}}(v), L_{\mathtt{comp}}(v) \rangle$ assigned by the prover to each node $v \in V$ is composed of the fields $L_{\mathtt{approx}}(v)$ and $L_{\mathtt{comp}}(v)$ serving the approximation task and the comparison task, respectively.

In the approximation task, the prover runs a centralized algorithm `ALG` that is guaranteed to produce a decomposable α-approximation $\mathsf{A} : V \to \{0,1\}^*$ for graph G and input assignment I. The $L_{\mathtt{approx}}(v)$ field of the label $L(v)$ assigned by the prover to each node $v \in V$ consists of both $\mathsf{A}(v)$ and a proof that the output assignment A is indeed the outcome of the centralized algorithm `ALG`. The correctness requirement for this task is defined so that the verifier accepts G_{I} if and only if the field $L_{\mathtt{approx}}(v)$ encodes an output assignment A that can be obtained using `ALG`.

The purpose of the comparison task is to verify that $\lambda(\mathsf{I}, \mathsf{A}) \geq k$ (resp., $\lambda(\mathsf{I}, \mathsf{A}) \leq k$), where λ is the decomposition function associated with the (decomposable) MinDGP (resp., MaxDGP) Ψ and A is the output assignment encoded in the $L_{\mathtt{approx}}(v)$ fields of the labels $L(v)$ assigned to nodes $v \in V$. This is done by means of the (λ, k)-comparison scheme (resp., the $(-\lambda, -k)$-comparison scheme) presented in Section 2.

The correctness of the α-ADPLS for the MinDGP (resp., MaxDGP) Ψ and the integer k is established as follows. If $OPT_{\Psi}(G, \mathsf{I}) \geq k$ (resp., $OPT_{\Psi}(G, \mathsf{I}) \leq k$), then the $L_{\mathtt{approx}}(v)$ field of the label $L(v)$ assigned by the prover to each node $v \in V$ encodes an output assignment $\mathsf{A} : V \to \{0,1\}^*$ generated by the algorithm `ALG`. This means that A is a decomposable α-approximation, thus $\lambda(\mathsf{I}, \mathsf{A}) \geq OPT_{\Psi}(G, \mathsf{I}) \geq k$ (resp., $\lambda(\mathsf{I}, \mathsf{A}) \leq OPT_{\Psi}(G, \mathsf{I}) \leq k$) and the verifier accepts G_{I}. On the other hand, if $OPT_{\Psi}(G, \mathsf{I}) < k/\alpha$ (resp., $OPT_{\Psi}(G, \mathsf{I}) > \alpha \cdot k$), then for any decomposable α-approximation A, it holds that $\lambda(\mathsf{I}, \mathsf{A}) \leq \alpha \cdot OPT_{\Psi}(G, \mathsf{I}) < k$ (resp., $\lambda(\mathsf{I}, \mathsf{A}) \geq OPT_{\Psi}(G, \mathsf{I})/\alpha > k$), hence the verifier rejects G_{I} for any label assignment $L : V \to \{0,1\}^*$.

The proof size of the α-ADPLS designed via the VCA method is the maximum size of a label $L(v) = \langle L_{\mathtt{approx}}(v), L_{\mathtt{comp}}(v) \rangle$ assigned by the prover for a given input graph G_{I} such that $OPT_{\Psi}(G, \mathsf{I}) \geq k$ (resp., $OPT_{\Psi}(G, \mathsf{I}) \leq k$). As discussed in Section 2, it is guaranteed that the $L_{\mathtt{comp}}(\cdot)$ fields are represented using $O(\log n + H)$ bits, where H is an upper bound on the number of bits required to represent a $\lambda(\mathsf{I}(v), \mathsf{O}(v))$ value for any $v \in V$, and A is the decomposable α-approximation generated by the prover in the approximation task. In [10], we develop ADPLSs whose $L_{\mathtt{approx}}(\cdot)$ fields are also represented using $|L_{\mathtt{approx}}(v)| = O(\log n + H)$

bits. Moreover, the OptDGPs we consider admit some fixed parameter $W \in \mathbb{Z}$ (typically an upper bound on the weights in the graph) such that $H = O(\log n + \log W)$ which results in a proof size of $O(\log n + \log W)$.

A desirable feature of the ADPLSs we develop in [10] is that the centralized algorithms ALG employed in the approximation task are efficient, hence the prover runs in polynomial time. Since the (sequential) runtime of the verifier is also polynomial, it follows that all our ADPLSs are sequentially efficient.

4 Reductions Between APLSs and ADPLSs

4.1 From an α-ADPLS to an α-APLS

Consider an identified decomposable MinDGP (resp., MaxDGP) $\Psi = \langle \Pi, f \rangle$ with a decomposition function $\lambda : \{0,1\}^* \times \{0,1\}^* \to \mathbb{R}$. Let p and r be the proof sizes of a feasibility scheme for Π and an α-ADPLS for Ψ, respectively. We establish the following lemma.

▶ **Lemma 4.1.** *There exists an α-APLS for Ψ with a proof size of $O(p + r + \log n + H)$, where H is the maximal number of bits required to represent $\lambda(\mathsf{I}(v), \mathsf{O}(v))$ for any $v \in V$.*

Proof. Observe that if O is known to be a feasible solution for G and I, then the correctness requirements of an α-APLS for the MinDGP (resp., MaxDGP) Ψ are equivalent to those of an α-ADPLS for Ψ and $k = f(G_{\mathsf{I},\mathsf{O}})$. That is, for a given IO graph $G_{\mathsf{I},\mathsf{O}} \in \Pi$, if $OPT_\Psi(G, \mathsf{I}) \geq k$ (resp., $OPT_\Psi(G, \mathsf{I}) \leq k$), then O is an optimal solution for G and I which requires the verifier of an α-APLS to accept $G_{\mathsf{I},\mathsf{O}}$; if $OPT_\Psi(G, \mathsf{I}) < k/\alpha$ (resp., $OPT_\Psi(G, \mathsf{I}) > \alpha \cdot k$), then O is at least α-far from being optimal for G and I which requires the verifier of an α-APLS to reject $G_{\mathsf{I},\mathsf{O}}$.

The design of an α-APLS for Ψ is thus enabled by taking the label assigned by the prover to each node $v \in V$ to be $L(v) = \langle L_{\texttt{feas}}(v), L_{\texttt{obj}}(v), L_{\texttt{comp}}(v), L_{\texttt{ADPLS}}(v) \rangle$, where $L_{\texttt{feas}}(v)$ is the p-bit label assigned to v by the prover of the feasibility scheme for Π; $L_{\texttt{obj}}(v) = f(G_{\mathsf{I},\mathsf{O}})$ (note that all nodes are assigned with the same $L_{\texttt{obj}}(\cdot)$ field); $L_{\texttt{comp}}(v)$ is the label constructed in the $(\lambda, L_{\texttt{obj}}(v))$-comparison scheme (resp., the $(-\lambda, -L_{\texttt{obj}}(v))$-comparison scheme) presented in Section 2; and $L_{\texttt{ADPLS}}(v)$ is the r-bit label of an α-ADPLS for Ψ and $L_{\texttt{obj}}(v)$. This label assignment allows the verifier to verify that (1) O is a feasible solution for G and I; (2) $f(G_{\mathsf{I},\mathsf{O}}) \geq L_{\texttt{obj}}(v)$ (resp., $f(G_{\mathsf{I},\mathsf{O}}) \leq L_{\texttt{obj}}(v)$) for each $v \in V$; and (3) the verifier of an α-ADPLS for Ψ and $k = f(G_{\mathsf{I},\mathsf{O}})$ accepts the input graph G_{I}. ◀

4.2 From an α-APLS to an α-ADPLS

Consider an identified, locally verifiable, and (β, γ)-fitted OptDGP Ψ with the mappings $v : [s] \to V$ and $e : [t] \to E$ that are associated with its LP relaxation P whose matrix form is given by the variable vector $\mathbf{x} = \{x_j\} \in \mathbb{R}^t$ and coefficient matrix and vectors $\mathbf{A} = \{a_{i,j}\} \in \mathbb{R}^{s \times t}$, $\mathbf{b} = \{b_i\} \in \mathbb{R}^s$, and $\mathbf{c} = \{c_j\} \in \mathbb{R}^t$. Define $D_u = \{i \mid v(i) = u\}$ for each $u \in V$ and let $d = \max_{u \in V}\{|D_u|\}$. Let $b_{\max}$ be the maximal number of bits required to represent b_i for any $i \in [s]$. Let $\alpha = \beta \cdot \gamma$ and let r be the proof size of an α-APLS for Ψ produced by the primal dual method. We obtain the following lemma.

▶ **Lemma 4.2.** *There exists an α-ADPLS for Ψ with a proof size of $O(\log n + \log d + b_{\max} + r)$.*

Proof. We construct an α-ADPLS for the MinDGP (resp., MaxDGP) Ψ by means of the VCA method. Recall that an α-APLS for Ψ established by means of the primal dual method is defined so that the labels encode a feasible dual solution $\mathbf{y} \in \mathbb{R}^s$ that satisfies $OPT_\Psi(G, \mathsf{I}) \leq \alpha \cdot \mathbf{b}^{\mathrm{T}}\mathbf{y}$ (resp., $OPT_\Psi(G, \mathsf{I}) \geq \frac{1}{\alpha} \cdot \mathbf{b}^{\mathrm{T}}\mathbf{y}$). Define $\mathsf{A}(u) = \mathbf{y}(u) = \langle y_i \mid v(i) = u \rangle$

and $\lambda(\mathsf{I}(u), \mathsf{A}(u)) = \alpha \cdot \sum_{i:v(i)=u} b_i y_i$ (resp., $\lambda(\mathsf{I}(u), \mathsf{A}(u)) = \frac{1}{\alpha} \cdot \sum_{i:v(i)=u} b_i y_i$) for each $u \in V$. The prover sets the sub-label $L_{\texttt{approx}}(u) = \mathsf{A}(u) = \mathbf{y}(u)$ associated with the approximation task for each node $u \in V$, which allows the verifier to verify that $\mathbf{y}$ is a feasible dual solution.

For the correctness of this scheme, it suffices to show that A is a decomposable α-approximation for G and I (with respect to the decomposition function λ). Note that $\mathbf{y}$ is defined so that it satisfies $OPT_{\Psi}(G, \mathsf{I}) \leq \alpha \cdot \mathbf{b}^{\mathrm{T}}\mathbf{y}$ (resp., $OPT_{\Psi}(G, \mathsf{I}) \geq \frac{1}{\alpha} \cdot \mathbf{b}^{\mathrm{T}}\mathbf{y}$); and weak duality implies that $\alpha \cdot \mathbf{b}^{\mathrm{T}}\mathbf{y} \leq \alpha \cdot OPT_{\Psi}(G, \mathsf{I})$ (resp., $\frac{1}{\alpha} \cdot \mathbf{b}^{\mathrm{T}}\mathbf{y} \geq \frac{1}{\alpha} \cdot OPT_{\Psi}(G, \mathsf{I})$). It follows that A is a decomposable α-approximation for G and I since $\lambda(\mathsf{I}, \mathsf{A}) = \sum_{u \in V} \lambda(\mathsf{I}(u), \mathsf{A}(u)) = \alpha \cdot \sum_{u \in V} \sum_{i:v(i)=u} b_i y_i = \alpha \cdot \mathbf{b}^{\mathrm{T}}\mathbf{y}$ (resp., $\lambda(\mathsf{I}, \mathsf{A}) = \sum_{u \in V} \lambda(\mathsf{I}(u), \mathsf{A}(u)) = \frac{1}{\alpha} \cdot \sum_{u \in V} \sum_{i:v(i)=u} b_i y_i = \frac{1}{\alpha} \cdot \mathbf{b}^{\mathrm{T}}\mathbf{y}$). ◀

References

1. B. Awerbuch, B. Patt-Shamir, and G. Varghese. Self-stabilization by local checking and correction. In *Proceedings 32nd Annual Symposium of Foundations of Computer Science*, pages 268–277, 1991.
2. Nir Bacrach, Keren Censor-Hillel, Michal Dory, Yuval Efron, Dean Leitersdorf, and Ami Paz. Hardness of distributed optimization. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, 2019.
3. Alkida Balliu, Gianlorenzo D'Angelo, Pierre Fraigniaud, and Dennis Olivetti. What can be verified locally? *J. Comput. Syst. Sci.*, 97:106–120, 2018.
4. Lélia Blin, Pierre Fraigniaud, and Boaz Patt-Shamir. On proof-labeling schemes versus silent self-stabilizing algorithms. In *Stabilization, Safety, and Security of Distributed Systems*, pages 18–32, 2014.
5. Keren Censor-Hillel, Ami Paz, and Mor Perry. Approximate proof-labeling schemes. *Theor. Comput. Sci.*, 811:112–124, 2020.
6. Miroslav Chlebík and Janka Chlebíková. The steiner tree problem on graphs: Inapproximability results. *Theor. Comput. Sci.*, 406(3):207–214, 2008.
7. Pierluigi Crescenzi, Pierre Fraigniaud, and Ami Paz. Trade-offs in distributed interactive proofs. In *33rd International Symposium on Distributed Computing*, pages 13:1–13:17, 2019.
8. Irit Dinur and Samuel Safra. On the hardness of approximating minimum vertex cover. *Annals of Mathematics*, 162(1):439–485, 2005.
9. Irit Dinur and David Steurer. Analytical approach to parallel repetition. In *Symposium on Theory of Computing, STOC*, pages 624–633, 2014.
10. Yuval Emek and Yuval Gil. Twenty-two new approximate proof labeling schemes (full version), 2020. `arXiv:2007.14307`.
11. Laurent Feuilloley. Introduction to local certification, 2019.
12. Laurent Feuilloley and Pierre Fraigniaud. Survey of distributed decision. *Bull. EATCS*, 119, 2016.
13. Laurent Feuilloley and Pierre Fraigniaud. Error-sensitive proof-labeling schemes. In *31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria*, volume 91 of *LIPIcs*, pages 16:1–16:15, 2017.
14. Laurent Feuilloley, Pierre Fraigniaud, Juho Hirvonen, Ami Paz, and Mor Perry. Redundancy in distributed proofs. In *32nd International Symposium on Distributed Computing, DISC*, volume 121 of *LIPIcs*, pages 24:1–24:18, 2018.
15. Pierre Fraigniaud, Amos Korman, and David Peleg. Local distributed decision. In *IEEE 52nd Annual Symposium on Foundations of Computer Science, FOCS*, pages 708–717, 2011.
16. Pierre Fraigniaud, Pedro Montealegre, Rotem Oshman, Ivan Rapaport, and Ioan Todinca. On distributed merlin-arthur decision protocols. In *Structural Information and Communication Complexity*, pages 230–245, 2019.
17. Pierre Fraigniaud, Boaz Patt-Shamir, and Mor Perry. Randomized proof-labeling schemes. *Distributed Comput.*, 32(3):217–234, 2019.

18 Mika Göös and Jukka Suomela. Locally checkable proofs in distributed computing. *THEORY OF COMPUTING*, 12:1–33, 2016.

19 Johan Håstad. Some optimal inapproximability results. *J. ACM*, 48(4):798–859, July 2001.

20 Marek Karpinski, Michael Lampis, and Richard Schmied. New inapproximability bounds for TSP. *J. Comput. Syst. Sci.*, 81(8):1665–1677, 2015.

21 Subhash Khot. On the power of unique 2-prover 1-round games. In *Proceedings of the Thiry-Fourth Annual ACM Symposium on Theory of Computing*, page 767–775, 2002.

22 Subhash Khot, Guy Kindler, Elchanan Mossel, and Ryan O'Donnell. Optimal inapproximability results for max-cut and other 2-variable csps? *SIAM Journal on Computing*, 37(1):319–357, 2007.

23 Subhash Khot and Oded Regev. Vertex cover might be hard to approximate to within 2-epsilon. *J. Comput. Syst. Sci.*, 74(3):335–349, 2008.

24 Gillat Kol, Rotem Oshman, and Raghuvansh R. Saxena. Interactive distributed proofs. In *PODC 2018 - Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, pages 255–264, July 2018.

25 Amos Korman and Shay Kutten. Distributed verification of minimum spanning trees. *Distributed Comput.*, 20(4):253–266, 2007.

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

27 E. Kushilevitz and N. Nisan. *Communication Complexity.* Cambridge University Press, 2006.

28 Moni Naor, Merav Parter, and Eylon Yogev. The power of distributed verifiers in interactive proofs. In *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 1096–115, 2020.

29 Rafail Ostrovsky, Mor Perry, and Will Rosenbaum. Space-time tradeoffs for distributed verification. In Shantanu Das and Sebastien Tixeuil, editors, *Structural Information and Communication Complexity*, pages 53–70, Cham, 2017. Springer International Publishing.

30 Boaz Patt-Shamir and Mor Perry. Proof-labeling schemes: Broadcast, unicast and in between. In *Stabilization, Safety, and Security of Distributed Systems - 19th International Symposium, SSS*, volume 10616, pages 1–17. Springer, 2017.

Distributed Constructions of Dual-Failure Fault-Tolerant Distance Preservers

Merav Parter
Weizmann Institute of Science, Rehovot, Israel
merav.parter@weizmann.ac.il

Abstract

Fault tolerant distance preservers (spanners) are sparse subgraphs that preserve (approximate) distances between given pairs of vertices under edge or vertex failures. So-far, these structures have been studied thoroughly mainly from a centralized viewpoint. Despite the fact fault tolerant preservers are mainly motivated by the error-prone nature of distributed networks, not much is known on the distributed computational aspects of these structures.

In this paper, we present distributed algorithms for constructing fault tolerant distance preservers and +2 additive spanners that are resilient to at most *two edge* faults. Prior to our work, the only non-trivial constructions known were for the *single* fault and *single source* setting by [Ghaffari and Parter SPAA'16].

Our key technical contribution is a distributed algorithm for computing distance preservers w.r.t. a subset S of source vertices, resilient to two edge faults. The output structure contains a BFS tree $BFS(s, G \setminus \{e_1, e_2\})$ for every $s \in S$ and every $e_1, e_2 \in G$. The distributed construction of this structure is based on a delicate balance between the edge congestion (formed by running multiple BFS trees simultaneously) and the sparsity of the output subgraph. No sublinear-round algorithms for constructing these structures have been known before.

2012 ACM Subject Classification Networks → Network algorithms

Keywords and phrases Fault Tolerance, Distance Preservers, CONGEST

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.21

Related Version `https://arxiv.org/abs/2010.01503`

Funding Partially funded by the ISF, grant no. 713130.

1 Introduction

Fault tolerant distance preservers are sparse subgraphs that preserve distances between given pairs of nodes under edge or vertex failures. In this paper, we present the first non-trivial distributed constructions of source-wise distance preservers and additive spanners that can handle *two* edge failures. We start by providing some background on fault-tolerant preservers from a graph-theoretical perspective, and then provide the distributed algorithmic context.

Fault-Tolerant Distance Preserves. Distances preservers are sparse subgraphs that preserve the distances between a given pairs of nodes *exactly*. As distance preservers are often computed for distributed networks where parts can spontaneously fail, a desired requirement for these applications is *fault-tolerance*. For every small set of edge failures, fault tolerant preservers are required to contain *replacement paths* around the faulted set. Formally, for a pair of vertices s and t and a subset of edge failures F, a replacement path $P(s, t, F)$ is an s-t shortest path in the surviving graph $G \setminus F$. The efficient (centralized) computation of all replacement path distances for a given s-t pair and a given source vertex s has attracted a lot of attention since the 80's [16, 15, 9, 22, 12, 24, 6, 1, 7]. Most of these works focus on the single-failure case, and relatively little is known on the complexity of distance preserving computation under multiple edge faults.

© Merav Parter;
licensed under Creative Commons License CC-BY
34th International Symposium on Distributed Computing (DISC 2020).
Editor: Hagit Attiya; Article No. 21; pp. 21:1–21:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Parter and Peleg [19] introduced the notion of FT-BFS structure for a given source vertex s. Roughly speaking, an FT-BFS structure is a subgraph of the original graph that preserves all $\{s\} \times V$ distances under a single failure of an edge or a vertex. An FT-MBFS structure is collection of FT-BFS structures with respect to a collection of sources $S \subseteq V$. For every n-vertex graph $G = (V, E)$ and a source set $S \subseteq V$, [19] presented an algorithm for computing an FT-MBFS subgraph $H \subseteq G$ with $O(\sqrt{|S|}n^{3/2})$ edges. This was also shown to be existentially tight. Parter [17] presented a construction of dual-failure FT-BFS structures with $O(n^{5/3})$ edges. Gupta and Khan [13] extended this construction to multiple sources S and provided a dual-failure FT-MBFS with $O(|S|^{1/3}n^{5/3})$ edges, which is also existentially tight [19]. For a general bound on the number of fault f, the state-of-the-art upper bound is $O(|S|^{1/2^f} n^{2-1/2^f})$ by Bodwin et al. [3], a lower bound of $\Omega(|S|^{1/(f+1)}n^{2-1/(f+1)})$ is known by [17]. Closing this gap is a major open problem.

Fault-tolerant (FT) additive spanners are sparse subgraphs that preserve distance under failure with some additive stretch. While various upper bound constructions are known [4, 2, 18], to this date no lower bounds are known for constant additive stretch. For example, one can compute +2 FT-additive spanners with $\widetilde{O}(n^{5/3})$ edges[1], but no lower-bound is known for this structure.

Distributed Constructions. Despite the fact that the key motivation for fault tolerant preservers comes from distributed networks, considerably less is known on their distributed constructions. In this paper, we consider the standard CONGEST model of distributed computing [20]. In this model, the network is abstracted as an n-node graph $G = (V, E)$, with one processor on each node. Initially, these processors only know their incident edges in the graph, and the algorithm proceeds in synchronous communication rounds over the graph $G = (V, E)$. In each round, nodes are allowed to exchange $O(\log n)$ bits with their neighbors and perform local computation. Throughout, the diameter of the graph $G = (V, E)$ is denoted by D.

Ghaffari and Parter [11] presented the first distributed constructions of fault tolerant distance preserving structures. For every n-vertex D-diameter graph $G = (V, E)$ and a source vertex $s \in V$, they gave an $\widetilde{O}(D)$-round algorithm for computing an FT-BFS structure with respect to s. Both the size bound of the output structure and the round complexity of their algorithm are nearly optimal. An additional useful property of that algorithm is that it also computes the length of all the $\{s\} \times V$ replacement paths. To the best of our knowledge, currently there are no non-trivial distributed constructions that support either multiple sources or more than a single fault. A natural extension of [11] to a subset of sources S (dual faults) might lead to a round complexity of $\Omega(|S|D)$ (resp., $\Omega(D^2)$ rounds[2]). These bounds are inefficient for graphs with a large diameter, or for supporting a large number of sources. Finally, while distributed constructions for additive spanners are known [21, 5, 8], to this date there are no known distributed constructions of fault-tolerant additive spanners.

1.1 Our Results

In this paper we overcome the single-source and single-fault distributed barriers. We present constructions of FT preservers and additive spanners resilient to two edge failures with *sublinear* round complexities.

---

[1] The notation $\widetilde{O}$ hides poly-logarithmic terms in the number of vertices n.
[2] These bounds indeed seem to be achievable by slightly adapting the algorithm [11].

Fault Tolerant Distance Preservers. Given an unweighted and undirected n-vertex graph $G = (V, E)$ and integer $f \geq 1$, a subgraph $H \subseteq G$ is an f-FT-MBFS structure w.r.t. S if:

$$\text{dist}(s, t, H \setminus F) = \text{dist}(s, t, G \setminus F), \text{ for every } s \in S, t \in V, F \subseteq E \text{ and } |F| \leq f \ .$$

When $f = 1$, we call H an FT-MBFS structure, and when $f = 2$ it is called a dual-failure FT-MBFS.

▶ **Theorem 1** (Distributed FT-MBFS). *There exists a randomized algorithm that given an n-vertex graph $G = (V, E)$, and a subset $S \subseteq V$ computes w.h.p. a subgraph $H \subseteq G$ such that H is an FT-MBFS w.r.t. S and $|E(H)| = O(\sqrt{|S|}n^{3/2})$ edges. The round complexity is $\widetilde{O}(D + \sqrt{n|S|})$.*

This improves upon the $O(|S|D)$ bound implied by the FT-BFS algorithm of [11], for $D \geq \sqrt{n/|S|}$. The size of the FT-MBFS subgraph H is existentially optimal (up to a logarithmic factor).

We also consider the dual-failure setting. In the centralized literature it has been widely noted that the dual-failure case is already considerably more involved compared to the single fault setting. Indeed, there has been no prior distributed constructions of distance preserving subgraphs that are resilient to two faults. We provide a simplified centralized algorithm for the dual failure setting which serves the basis for our distributed construction:

▶ **Theorem 2** (Distributed Dual-Failure FT-MBFS). *There exists a randomized algorithm that given an n-vertex graph $G = (V, E)$, and a subset $S \subseteq V$ computes w.h.p. a subgraph $H \subseteq G$ such that H is a dual-failure FT-MBFS w.r.t. S and contains $O(|S|^{1/8} \cdot n^{15/8})$ edges. The round complexity is $\widetilde{O}(D + n^{7/8}|S|^{1/8} + |S|^{5/4}n^{3/4})$.*

We note that the size of our subgraph is suboptimal, as there exist (centralized) constructions [13, 17] that compute dual-failure FT-MBFS subgraphs with $O(|S|^{1/3}n^{5/3})$ edges. These constructions, however, are inherently sequential, and it is unclear how to efficiently implement them in the distributed setting. Specifically, in the CONGEST model, a naive simultaneous computation of multiple BFS trees $\texttt{BFS}(s, G \setminus \{e_1, e_2\})$ for every $s \in S$ and $e_1, e_2 \in G$ might result in a very large *congestion* over the graph edges. To reduce this congestion, one needs to balance the edge congestion and the sparsity of the output subgraph. These two opposing forces lead to suboptimal constructions w.r.t. the size, but with the benefit of obtaining a sub-linear round-complexity. We also note that our algorithms solve the subgraph problem rather than the distance computation problem. That is, in contrast to the FT-BFS algorithm of [11], we compute only the FT preserving subgraph but not necessarily the FT distances.

Fault Tolerant Additive Spanners. We employ the distributed construction of FT-MBFS structures to provide the first non-trivial constructions of fault tolerant +2 additive spanners. These structures are defined as follows. Given an unweighted undirected n-vertex graph $G = (V, E)$ and integer $f \geq 1$ and a stretch parameter β, a subgraph $H \subseteq G$ is an $+\beta$ f-FT additive spanner w.r.t. S if:

$$\text{dist}(s, t, H \setminus F) \leq \text{dist}(s, t, G \setminus F) + \beta, \text{ for every } s, t \in V, F \subseteq E \text{ and } |F| \leq f \ .$$

When $f = 1$, we call H an +2 FT-additive spanner, and when $f = 2$ it is called a +2 dual-failure FT-additive spanner. By using Thm. 1 and Thm. 2 respectively, we get:

▶ **Corollary 3** (+2 Additive Spanner, Single Fault). *For every n-vertex graph $G = (V, E)$, there exists a randomized algorithm that w.h.p. computes +2 FT-additive spanner $H \subseteq G$ with $\widetilde{O}(n^{5/3})$ edges in $O(D + n^{5/6})$ rounds.*

The size of the +2 FT-additive spanner matches the state-of-the-art bound. For the dual-failure setting, our bounds are suboptimal due to the suboptimality of the dual-failure `FT-MBFS` structures.

▶ **Corollary 4** (+2 Additive Spanner, Two Faults)**.** *For every n-vertex graph $G = (V, E)$, there exists a randomized algorithm that w.h.p. computes a $+2$ dual-failure FT-additive spanner with $\widetilde{O}(n^{17/9})$ edges within $\widetilde{O}(D + n^{8/9})$ rounds.*

No sublinear round algorithms for +2 FT-additive spanners with $o(n^2)$-edges were known before.

The High-Level Approach. We provide the high-level ideas required to compute `FT-MBFS` structures w.r.t. a collection of source nodes S. This construction is later on used for computing FT-additive spanners and dual-failure `FT-MBFS` structures. By definition, an `FT-MBFS` structure for S is required to contain a BFS tree w.r.t. each $s \in S$ in every graph $G \setminus \{e\}$. Upon using any consistent tie-breaking of shortest-path distances, the union of all these trees contain $O(|S|n^{3/2})$ edges [19]. Our goal is then to compute all these BFS trees efficiently in the CONGEST model. For the single source case, the key observation made in [11] is that for every vertex t, it is sufficient to send only $O(D)$ BFS tokens throughout the computation: one token for every edge e on the shortest s-t path $\pi(s, t)$. The reason is that a failing of an edge $e \notin \pi(s, t)$ does not effect the s-t distance. Since every edge (u, t) is required to pass through only $|\pi(s, t)| = O(D)$ BFS tokens, using the random-delay approach, all these trees could be computed in dilation+congestion= $\widetilde{O}(D)$ rounds, w.h.p. Extending this idea to multiple sources, ultimately leads to a round complexity of $\Omega(D|S|)$. Indeed a-priori it is unclear how to break this barrier, as for every s and $e \in \pi(s, t)$, the s-t distance in $G \setminus \{e\}$ might be different, forcing t to receive the BFS token from $\Omega(D|S|)$ BFS algorithms. Our key idea is to define for every vertex t a smaller set of *relevant pairs* (s, e) from which it is allowed to receive the BFS tokens. This set $\{(s, e)\}$ is defined by including only edges e that are sufficiently *close* to t on its $\pi(s, t)$ path for every s. The main technical issue that arises with this idea is the inconsistency in the definition of relevant pairs between nodes on a given replacement path $P(s, t, e)$. In particular, there might be cases where (s, e) is relevant for t but it is not in the relevant set of some vertex w on the $P(s, t, e)$ path. In such a case, w might block the propagation the BFS token $BFS(s, G \setminus \{e\})$ (as (s, e) is not in its relevant set) which would prevent t from receiving it. These technical issues become more severe in the dual failure setting, due to a considerably more delicate interaction between the dual-failure replacement paths. In the very high level to mitigate this problem, we add to the output structure a collection of an (FT-) BFS trees w.r.t. a *randomly* sampled set of nodes. This edge set would compensate (in a non-trivial manner) for the lost tokens of the truncated BFS constructions.

1.2 Preliminaries

Given an unweighted n-vertex graph $G = (V, E)$, for an $s, t \in V$ and $e \in G$, the replacement path $P(s, t, e)$ is an s-t shortest path in $G \setminus \{e\}$. Throughout, we assume that the shortest-path ties are broken in a consistent manner using the vertex IDs. For a given $p \in [0, 1]$, let $\texttt{Sample}(V, p)$ be a subset of vertices obtained by sampling each vertex in V independently with probability p. Let $\texttt{BFS}(s, G)$ be a BFS tree rooted at s in G. Throughout, the (unique) shortest-path between any pair x, y in G is denoted by $\pi(x, y)$. Let $N(u)$ be the neighbors of u in G. Given a tree T and $u, v \in V(T)$, let $\pi(u, v, T)$ be the tree path between u and v. Throughout, all shortest-path ties are broken in a consistent manner, by preferring vertices

of lower IDs. That is, in every BFS computation (which defines the shortest-path) every vertex picks its parent to be the vertex of minimum ID among all its potential parents in the tree. For a given integer parameter σ, let $\pi_\sigma(u,v)$ denote the set of last $\min\{\sigma, |\pi(u,v)|\}$ edges (closest to v) on the path $\pi(u,v)$. For an edge $e=(x,y)$ and a subgraph $G' \subseteq G$, let $\text{dist}(e,t,G') = \min\{\text{dist}(x,t,G'), \text{dist}(y,t,G')\}$. For an s-t path P, let $\texttt{LastE}(P)$ be the last edge of the path (incident to t). For $a,b \in P$, let $P[a,b]$ be the sub-path segment between a and b in P.

▶ **Definition 5.** *For a given source vertex s, a subgraph H is an* FT-BFS structure *with respect to s if* $\text{dist}(s,t,H \setminus \{e\}) = \text{dist}(s,t,G \setminus \{e\})$ *for every $t \in V$ and $e \in E$. In the same manner, for a given subset $S \subseteq V$, a subgraph H is an* multi-source FT-BFS structure *with respect to S if* $\text{dist}(s,t,H \setminus \{e\}) = \text{dist}(s,t,G \setminus \{e\})$ *for every $s,t \in S \times V$ and $e \in E$.*

▶ **Fact 6** ([19]). *For every n-vertex graph $G=(V,E)$, and a subset $S \subseteq V$, let*

$$H = \bigcup_{s,t,e \in S \times V \times E} \{\texttt{LastE}(P(s,t,e))\}.$$

Then, H is an **FT-MBFS** *structure w.r.t. S and $|E(H)| = O(\sqrt{|S|} \cdot n^{3/2})$. This edge bound is existentially tight.*

The random delay Technique. Throughout, we make an extensive use of the random delay approach of [14, 10]. Specifically, we use the following theorem:

▶ **Theorem 7** ([10, Theorem 1.3]). *Let G be a graph and let $A_1, \ldots, A_m$ be m distributed algorithms in the* CONGEST *model, where each algorithm takes at most $\mathtt{d}$ rounds, and where for each edge of G, at most $\mathtt{c}$ messages need to go through it, in total over all these algorithms. Then, there is a randomized distributed algorithm (using only private randomness) that, with high probability, produces a schedule that runs all the algorithms in $O(\mathtt{c} + \mathtt{d} \cdot \log n)$ rounds, after $O(\mathtt{d} \log^2 n)$ rounds of pre-computation.*

2 Simplified Meta-Algorithms

We start by presenting simplified (centralized) constructions of FT preserving subgraphs. This would serve as a more convenient starting point for the distributed constructions of these structures.

FT-MBFS Structures. Let $T_S = \bigcup_{s \in S} T_s$ where $T_s = \texttt{BFS}(s,G)$. The $\texttt{FT-MBFS}$ subgraph H is given by the union of three subgraphs: T_S, and the subgraphs H_1 and H_2 defined by:

$$H_1 = \{\texttt{LastE}(P(s,t,e)) \mid s,t \in S \times V, e \in \pi_\sigma(s,t,T_s)\} \text{ where } \sigma = \sqrt{n/|S|},$$

and $H_2 = \bigcup_{r \in R} \texttt{BFS}(r,G)$, where $R = \texttt{Sample}(V, 10 \log n/\sigma)$.

▶ **Lemma 8.** *$E(H) = O(\sqrt{|S|} \cdot n^{3/2} \log n)$ and H is an* **FT-MBFS** *with respect to S.*

Proof. The size analysis follows by noting that $|T_S| = O(|S| \cdot n)$, and in addition each vertex t adds at most $\sigma = \sqrt{n/|S|}$ edges to H_1 for every source $s \in S$. Thus, $|H_1| = O(\sigma \cdot |S| n) = O(n\sqrt{n|S|})$. Turning to H_2, by the Chernoff bound, w.h.p., $|R| = O(n \log n/\sigma)$ and thus $|H_2| = O(\sqrt{|S|} \cdot n^{3/2} \log n)$.

We next show that H is an $\texttt{FT-MBFS}$ with respect to S. By Fact 6, it is sufficient to show that H contains the last edge of the replacement path $P(s,t,e)$ for every $s,t,e \in S \times V \times E$.

Fix a source $s \in S$ and a vertex $t \in V$. If $\text{dist}(s,t,G) \leq \sigma$, then $H_1 \cup T_S$ contain the last edge of $P(s,t,e)$ for every edge e. This is because $\texttt{LastE}(P(s,t,e))$ is added to H_1 for every $e \in \pi(s,t,T_s)$ and $P(s,t,e) = \pi(s,t,T_s)$ for every $e \notin \pi(s,t,T_s)$.

Thus, assume that $\text{dist}(s,t,G) \geq \sigma$ and specifically, consider an edge $e \in \pi(s,t,T_s) \setminus \pi_\sigma(s,t,T_s)$. Since $\text{dist}(s,t,G) \geq \sigma$, it also holds that $|P(s,t,e)| \geq \sigma$. Thus by the Chernoff bound, w.h.p. there is at least one vertex in R that lies in the $(\sigma/2)$-length suffix of $P(s,t,e)$. That is, w.h.p., there is a vertex $r \in V(P(s,t,e)) \cap R$ such that $\text{dist}(r,t,P(s,t,e)) \leq \sigma/2$. We next claim that there is *no* r-t shortest path in G that contains the failing edge e. This holds as $\text{dist}(r,t,G) \leq \text{dist}(r,t,G \setminus \{e\}) \leq \sigma/2$, but by the definition of the edge e, $\text{dist}(e,t,G) \geq \sigma$. By the uniqueness of the replacement paths, we have that $P(s,t,e)[r,t] = \pi(r,t,T_r)$ where $T_r = \texttt{BFS}(r,G)$, and thus $\texttt{LastE}(P(s,t,e)) \in H_2$. The claim follows. ◀

FT-MBFS Structures for 2 Faults. We next describe a simplified centralized construction of dual-failure FT-MBFS structures, this serves the basis for the distributed implementation. As we will see later on, computing these structures in the distributed setting is considerably more involved. To balance between edge congestion and sparsity of the structure, the final size of the FT-MBFS structures computed in distributed setting is larger compared to the centralized setting. For every $s,t \in V$ and $e_1,e_2 \in E$, let $P(s,t,\{e_1,e_2\})$ be the s-t shortest path in $G \setminus \{e_1,e_2\}$.

▶ **Fact 9** ([17]). *For every n-vertex graph $G=(V,E)$, and a subset $S \subseteq V$, let*

$$H = \bigcup_{s,t \in S \times V, e_1,e_2 \in E} \{\mathit{LastE}(P(s,t,\{e_1,e_2\}))\}.$$

Then H is a dual-failure FT-MBFS w.r.t. S.

Let S be the set of sources. Let R_1 be a random sample of $O(\sqrt{n|S|}\log n)$ vertices, and let R_2 be a random sample of $O(|S|^{1/4} \cdot n^{3/4} \log n)$ vertices. Let $H_1 = \bigcup_{r \in R_1} \texttt{FT-MBFS}(r,G)$ and $H_2 = \bigcup_{r \in R_2} \texttt{BFS}(r,G)$. The dual-failure FT-BFS structure w.r.t. S denoted by $\texttt{FT-BFS}_2(S)$ contains H_1, H_2 and the a subset of last edges of certain replacement paths. Let $\sigma_1 = \sqrt{n/|S|}$ and $\sigma_2 = (n/|S|)^{1/4}$. For every path P and integer σ, let P_σ be the σ-length suffix of P (when $\sigma \geq |P|$, then P_σ is simply P). Every vertex t, define the edge set E_t as

$$E_t = \bigcup_{s \in S} \bigcup_{e_1 \in \pi_{\sigma_1}(s,t)} \bigcup_{e_2 \in P_{\sigma_2}(s,t,e_1)} \{\texttt{LastE}(P(s,t,\{e_1,e_2\}))\} .$$

The final dual-failure FT-BFS structure is given by:

$$\texttt{FT-BFS}_2(S) = H_1 \cup H_2 \cup \bigcup_t E_t .$$

▶ **Lemma 10.** *$\texttt{FT-BFS}_2(S)$ is a dual-failure FT-BFS of S and contains $\widetilde{O}(|S|^{1/4} \cdot n^{7/4})$ edges.*

Proof. By the definition of the E_t sets, it remains to show that H contains the last edge of an replacement path $P(s,t,\{e_1,e_2\})$ such that either (i) $e_1 \in \pi(s,t) \setminus \pi_{\sigma_1}(s,t)$ or (ii) $e_1 \in \pi_{\sigma_1}(s,t)$ but $e_2 \in P(s,t,e_1) \setminus P_{\sigma_2}(s,t,e_1)$. We begin with (i). Since $e_1 \in \pi(s,t) \setminus \pi_{\sigma_1}(s,t)$, we have that $|P(s,t,e_1)| \geq \sqrt{n/|S|}$. Since we sample each vertex into R_1 with probability of $10\log n \cdot \sqrt{|S|/n}$, w.h.p. the $\sigma_1/2$-length suffix of $P(s,t,\{e_1,e_2\})$ contains a vertex, say r, in R_1. Since $\text{dist}(r,t,G) \leq \sigma_1/2$, we have that $e_1 \notin \pi(r,t,G)$, and by the uniqueness of the shortest paths, we have that $P(s,t,\{e_1,e_2\}) = P(s,t,\{e_1,e_2\})[s,r] \circ P(r,t,\{e_2\})$. Since H_2 contains

the FT-BFS w.r.t. r, it contains the path $P(r,t,\{e_2\})$ and thus $\texttt{LastE}(P(s,t,\{e_1,e_2\}))$ is in $\texttt{FT-BFS}_2(S)$. We proceed with (ii). Since $e_2 \in P(s,t,e_1) \setminus P_{\sigma_2}(s,t,e_1)$, we have that $|P(s,t,\{e_1,e_2\})| \geq \sigma_2$. Since we sample each vertex into R_2 with probability of $10\log n/\sigma_2$, w.h.p. the $\sigma_2/2$-length suffix of $P(s,t,\{e_1,e_2\})$ contains a vertex, say r', in R_2. Since $\text{dist}(r,t,G) \leq \sigma_2/2$, we have that $e_1, e_2 \notin \pi(r,t,G)$. By the uniqueness of the shortest paths, we have that $P(s,t,\{e_1,e_2\}) = P(s,t,\{e_1,e_2\})[s,r] \circ \pi(r,t,G)$. The claim follows as H_1 contains the BFS tree rooted at r, and concluding that $\texttt{LastE}(P(s,t,\{e_1,e_2\}))$ is in $\texttt{FT-BFS}_2(S)$. The size bound follows by noting that $|E(H_1)| = O(|\sqrt{|R_1|}n^{3/2})$ and $|E(H_2)| = O(|R_2|n)$. In addition, since each vertex t adds the last edges of $O(|S| \cdot (n/|S|)^{3/4})$ replacement paths, we get that $|E_t| = O(|S|^{1/4}n^{3/4})$. The lemma follows. ◀

Comparison to Bodwin et al. [3]. A simplified algorithm for computing sparse `FT-MBFS` structures (of suboptimal size) has been also provided by [3]. Their algorithm iterates over the vertices where for every vertex t the algorithm defines a small set of edges incident to t that should be added to the output subgraph H. For every *vertex* t, the algorithm reduces the task of computing `FT-MBFS` structure with respect to S sources and supporting f faults[3] into the computation of an `FT-MBFS` structure to support S' sources and $f-1$ faults, where $|S'| = O(\sqrt{|S|n})$. The main limitation in implementing this algorithm in the distributed setting is that for each vertex t the algorithm defines a *distinct* set of sources. For $f = 1$ for example, our simplified algorithm computes BFS trees w.r.t. a subset of sources S'. In contrast, in the algorithm of [3], a BFS tree is computed w.r.t. a distinct set of sources S_t for every vertex t, the union of all these S_t sets might be very large (leading to a large round complexity).

3 Distributed Construction of `FT-MBFS` Structures

In this section we prove Thm. 1 and present our main algorithm for computing sparse `FT-MBFS` structures with respect to S sources. This structure becomes useful both for the construction of FT-additive spanners, and for the computation of the dual-failure FT preservers.

3.1 The algorithm

Set $\sigma = \lceil \sqrt{n/|S|} \rceil$ and $\sigma' = 3\sigma$. The algorithm has two main steps. In the first step, a subset R of $O(n\log n/\sigma)$ vertices is uniformly sampled, and a BFS trees $T_s = \texttt{BFS}(s,G)$ is computed for every vertex $s \in S \cup R$. Let $T_S = \bigcup_{s\in S} T_s$ and $T_R = \bigcup_{r\in R} T_r$. All the edges of $T_R \cup T_S$ are added to the output subgraph H, by their corresponding endpoints.

In the second step, the algorithm computes a special subset of replacement paths, and the last edges of these replacement paths will be added to H. To define this subset, we need the following definition. For every $s,t \in S \times V$, each vertex t defines a set of *relevant edge-list* $\pi_{\sigma'}(s,v)$ that consists of the last σ' edges of its $\pi(s,v)$ paths. It also defines a shorter prefix $\pi_\sigma(s,v)$ that contains the last σ edges of this path.

The algorithm first lets each vertex t learn its relevant edge-list $\pi_{\sigma'}(s,t)$ for every $s \in S$. This can be done within $O(|S| \cdot \sigma' + D) = O(\sqrt{n|S|} + D)$ rounds by applying a simple pipeline strategy. From now on, the algorithm divides the time into phases of $\ell = O(\log n)$ rounds. Every $\texttt{BFS}(s, G \setminus \{e\})$ algorithm then starts in phase $\tau_{s,e}$, where $\tau_{s,e}$ is a random

[3] The task is to pick the edges of t that should be added to such a structure

variable with a uniform distribution in $[1, \sigma' \cdot |S|]$. Specifically, using the notion of k-wise independence, similarly to [11], all vertices can learn a random seed of $\mathcal{SR}$ of $O(\log n)$ bits. Using the seed $\mathcal{SR}$ and the IDs of the edge e and the source s, all vertices can compute the starting phase $\tau_{s,e}$ of each BFS construction $\texttt{BFS}(s, G \setminus \{e\})$. In the analysis section, we show that due to these random starting points, w.h.p., each edge $e' = (u, v)$ is required to send as most ℓ edges in every phase. In a standard application of a BFS computation with delay $\tau_{s,e}$, every vertex t is supposed to receive a $\texttt{BFS}(s, G \setminus \{e\})$-token (for the first time) in phase $\text{dist}_{G\setminus\{e\}}(s, t) + \tau_{s,e}$. In our case, the algorithm cannot afford to compute the entire BFS trees, but rather only certain fragments of them. Specifically, the BFS tokens are initiated and propagated following certain rules whose goal is to keep the congestion over the edges small. In the high-level, every vertex t would send its neighbor $u \in N(t)$ (such that $(u, t) \neq e$) a BFS token $\texttt{BFS}(s, G \setminus \{e\})$ only if $e \in \pi_{\sigma'}(s, u)$. In the special case where $e \notin \pi(s, t)$, the vertex t will initiate the BFS token to u in phase $\text{dist}_{G\setminus\{e\}}(s, t) + \tau_{s,e}$. In the remaining case where $e \in \pi(s, t)$, t will send u the token $\texttt{BFS}(s, G \setminus \{e\})$ to u in phase i iff (i) $e \in \pi_{\sigma'}(s, u)$ and t received the token $\texttt{BFS}(s, G \setminus \{e\})$ for the first time in phase $i - 1$.

As we will see in the analysis section, even-though each vertex t sends the BFS tokens $\texttt{BFS}(s, G \setminus \{e\})$ to neighbors u provided that $e \in \pi_{\sigma'}(s, u)$, it might be the case that a vertex u would not get the BFS token for each of its edges in $\pi_{\sigma'}(s, u)$. This might happen when the path $P(s, u, e)$ contains intermediate vertices w for which $e \notin \pi_{\sigma'}(s, w)$, which would block the propagation of the token. Fortunately, a more careful look reveals that in all the cases where $\texttt{LastE}(P(s, u, e)) \notin T_R$, the BFS token of $\texttt{BFS}(s, G \setminus \{e\})$ would complete its propagation over the entire $P(s, u, e)$ for every $e \in \pi_\sigma(s, u) \subseteq \pi_{\sigma'}(s, u)$. As we will see, this would be sufficient for the correctness of the `FT-MBFS` structure.

Second-Order Implementation Details. For the generation of the shared random seed, we use the same construction of [11] which is based on the notion of k-wise independence hash functions.

▶ **Lemma 11** ([11]). *The string of shared randomness SR can be generated and delivered to all vertices in $O(D + \log n)$ rounds.*

We argue that each vertex v by knowing the sets $\bigcup_{u \in N(v)\cup\{v\}} \pi_{\sigma'}(s, u)$, can locally compute the edges in $\pi_{\sigma'}(s, u) \setminus \pi(s, v)$ for every $u \in N(v)$.

▶ **Lemma 12.** *For every vertex v, neighbor $u \in N(v)$ and an edge $e \in \pi_{\sigma'}(s, u)$, v can locally decide if $e \in \pi(s, v)$ or not.*

Proof. Let $e \in \pi_{\sigma'}(s, u) \cap \pi(s, v)$. We will show that e can locally recognize that $e \in \pi(s, v)$. If $e \in \pi_{\sigma'}(s, v)$, then v clearly knows that $e \in \pi(s, v)$. Otherwise, if $e = (x, y) \in \pi_{\sigma'}(s, u) \setminus \pi_{\sigma'}(s, v)$, we show that y must be the endpoint of the first edge in $\pi_{\sigma'}(s, v)$. To see this, assume towards contradiction that y has no incident edge in $\pi_{\sigma'}(s, v)$. Since $e \in \pi_{\sigma'}(s, u)$, we have that $\text{dist}(x, u, G) \leq 3\sigma$. However, by the assumption, $\text{dist}(x, v, G) \geq 3\sigma + 2$, in contradiction as (u, v) are neighbors. As $y \in V(\pi_{\sigma'}(s, v))$ and $e = (x, y) \in \pi_{\sigma'}(s, u)$, v can deduce that x is the parent of y in T_s, and consequently that $(x, y) \in \pi(s, v)$ as well. ◀

Note that vertex u receives messages from all its potential parents in $\texttt{BFS}(s, G \setminus \{e\})$ at the same time, namely, at phase $\text{dist}(s, u, G \setminus \{e\}) + \tau_{s,e}$. It selects as its parent the vertex of minimum ID, which would guarantee that the shortest path ties are broken an a consistent manner, leading to a sparse structure.

Algorithm 1 The Distributed FT-MBFS Algorithm

1. Sample a set $R \subset V$ of $O(n \log n/\sigma)$ vertices uniformly at random from V.
2. Construct a BFS tree $T_s = \texttt{BFS}(G, s)$ rooted at s for every $s \in S \cup R$, and add these trees to H.
3. Number the edges of $T_S = \bigcup_{s \in S} T_s$ by numbers 1 to $|S|(n-1)$, where each edge $e \in T_s$ has a distinct number for every s containing $e \in T_s$.
4. Make each vertex v know the numbers of the edges on the σ'-length suffix $\pi_{\sigma'}(s, v)$ for every $s \in S$.
5. Let each vertex v send to each of its neighbors $u \in N(v)$ the numbers of the edges on $\bigcup_{s \in S} \pi_{\sigma'}(s, v)$.
6. Broadcast a string $\mathcal{SR}$ of $O(\log n)$ random bits.
7. For every $s \in S$, and $e \in T_s$, let $\tau_{s,e}$ be picked uniformly at random from $\{1, 2, \ldots, \sigma' \cdot |S|\}$ by setting it equal to $\mathcal{SR}[i]$ where i is the edge-number of e in T_s. Since $\mathcal{SR}$ is publicly known, given the ID of an edge e, a vertex can compute $\tau_{s,e}$ for every s.
8. Divide time into phases of $\ell = \Theta(\log n)$ rounds each.
9. Run each $\texttt{BFS}(G \setminus \{e\}, s)$ for every e and $s \in S$ at a speed of one hop per phase, following these rules for every vertex v:
   - For every edge $e \in \pi_{\sigma'}(s, u) \setminus \pi(s, v)$, and[4] every neighbor $u \in N(v)$, v sends u the BFS token $\texttt{BFS}(s, G \setminus \{e\})$ in phase $\text{dist}(s, v, G) + \tau_{s,e}$.
   - For every BFS token $BFS(s, G \setminus \{e\})$ received for the first time at phase i at v from a non-empty subset of neighbors $N'(v) \subseteq N(v)$, v does the following:
     - If $e \in \pi_\sigma(s, v)$, then v adds the edge (w, v) to H where w is the vertex of minimum-ID in $N'(u)$.
     - v sends the BFS token $\texttt{BFS}(s, G \setminus \{e\})$ in phase $i+1$ to every neighbor $u \in N(v) \setminus N'(v)$ satisfying that $e \in \pi_{\sigma'}(s, u)$.

Correctness. Unlike the single-source case, where the correctness of the algorithm was immediate by construction, here the correctness argument is more delicate. Specifically, in the `FT-BFS` construction of [11], for every vertex v, the BFS token $\texttt{BFS}(s, G \setminus \{e\})$ reached every vertex t for which $e \in \pi(s, t)$. In contrast, in our setting, only a subset of the replacement paths are fully constructed which poses a challenge for showing the correctness.

To show that the output subgraph H is indeed an `FT-MBFS` w.r.t. S, throughout, we fix a source $s \in S$, target $t \in V$ and an edge $e = (x, y)$. We need the following definitions. Let T_s be a BFS tree rooted at s for every $s \in S$. For a vertex y and a tree T_s, let $T_s(y)$ be the subtree rooted at y in T_s. A vertex w is said to be *sensitive* to an edge $e \in T_s$, if $e \in \pi(s, w)$. Observe that for every edge $e = (x, y) \in T_s$, where x is closer to s, the set of sensitive vertices to e are those that belong to $T_s(y)$.

▶ **Definition 13** (sensitive-detour). *For a given replacement path $P(s, t, e)$ let w be the first vertex on the path (closest to s) that is sensitive to e. We denote the segment $SD(s, t, e) = P(s, t, e)[w, t]$ by the* sensitive-detour *of $P(s, t, e)$.*

▶ **Observation 14.** *For every $s, t \in S \times V$ and $e = (x, y) \in G$, it holds that: (i) $SD(s, t, e) \subseteq T_s(y)$ and (ii) $P(s, t, e) = \pi(s, w') \circ (w', w) \circ SD(s, t, e)$ for a unique pair $w, w' \in P(s, t, e)$.*

Proof. (i) Let w be the first vertex in $T_s(y) \cap P(s, t, e)$, thus $SD(s, t, e) = P(s, t, e)[w, t]$. Assume towards contradiction that there exists $w' \in SD(s, t, e)$ such that $w' \notin T_s(y)$. Since

the shortest-paths are computed in a consistent manner, and $e \notin \pi(s, w')$, we get that $P(s,t,e)[s,w'] = \pi(s,w')$. Thus, $w \in \pi(s,w')$, contradiction as $w \in T_s(y)$.

(ii) Let w' be the neighbor of w (defined as above) on $P(s,t,e)$ that is closer to s. By definition, $w' \notin T_s(y)$ and thus by the uniqueness of the shortest-paths, we have $P(s,t,e) = \pi(s,w') \circ (w',w) \circ SD(s,t,e)$. ◀

▷ Claim 15. If $e \in \pi_{\sigma'}(s,w')$ for every $w' \in SD(s,t,e)$, then $\texttt{LastE}(P(s,t,e)) \in H$.

Proof. Let w be the first vertex on $SD(s,t,e)$ and let q be the preceding neighbor of w (not in $SD(s,t,e)$). Since $e \in \pi_{\sigma'}(s,w)$, the vertex q can locally detect that $e \notin \pi(s,q)$ (using Lemma 12). Note that since $q \notin SD(s,t,e)$, it holds $\text{dist}(s,q,G \setminus \{e\}) = \text{dist}(s,q,G)$. Thus, q send to w the BFS token $\texttt{BFS}(s, G \setminus \{e\})$ in phase $\text{dist}(s,q,G \setminus \{e\}) + 1 + \tau_{s,e}$. The token propagates over the $SD(s,t,e)$ segment at a speed of one hop per phase as for each $w' \in SD(s,t,e)$, $e \in \pi_{\sigma'}(s,w')$. ◁

▶ **Lemma 16.** *For every $s,t \in S \times V$ and $e \in G$, we have that $\texttt{LastE}(P(s,t,e)) \in H$.*

Proof. Fix a replacement path $P(s,t,e)$ where $e = (x,y)$. We consider the following cases.

Case (1): $e \in \pi(s,t) \setminus \pi_\sigma(s,t)$. In this case, $|P(s,t,e)| \geq \text{dist}(s,t,G) \geq \sigma$ and thus w.h.p. the $\sigma/2$-length suffix of the path contains at least one sampled vertex in R, say r. Since $\text{dist}(r,t,G) \leq \sigma/2$ but $\text{dist}(x,t,G) \geq \sigma$, the edge e does not appear on any r-t shortest path. As the shortest-path ties are broken in a consistent manner, we have that $P(s,t,e)[r,t] = \pi(r,t)$. Since the algorithm adds the BFS trees w.r.t. all vertices in R, we have that $\texttt{LastE}(\pi(r,t)) \in H$.

Case (2): $e \in \pi_\sigma(s,t)$ but $|SD(s,t,e)| \geq \sigma$. The proof for this case follows by noting that for every two vertices $u,v \in T_s(y)$, there is no u-v shortest path that go through the edge e. Assume towards contradiction that there is a u-v shortest path P that goes through e, since $\pi(x,u) \subset \pi(s,u)$, $\pi(x,v) \subset \pi(s,v)$, it holds that $e \in \pi(x,u), \pi(x,v)$, and thus:

$$|P| = \text{dist}_G(u,x) + \text{dist}(x,v) = 1 + \text{dist}_G(y,u) + 1 + \text{dist}_G(y,v) = 2 + \text{dist}(u,v) \ ,$$

contradiction that P is a u-v shortest path. Since $|SD(s,t,e)| \geq \sigma$, w.h.p., it contains at least one sampled vertex $r \in R$. As both $r,t \in T_s(y)$, $\pi(r,t)$ is free of failed edge e. Thus $P(s,t,e)[r,t] = \pi(r,t)$, concluding that $\texttt{LastE}(P(s,t,e)) \in H$. We note that this is the only case where the proof would not work for the case of a single vertex (rather than edge) fault.

Case (3): $e \in \pi_\sigma(s,t)$ but $|SD(s,t,e)| < \sigma$. This is the most interesting case as the last edge of the path $P(s,t,e)$ is not necessarily in $\bigcup_{r \in R} T_r$. We need to show that the suffix of the path $P(s,t,e)$ is computed by the algorithm, and that its last edge is added to H. Since $|SD(s,t,e)| < \sigma$, it holds that $\text{dist}(w,t,G \setminus \{e\}) \leq \sigma$ where w is the first vertex on the $SD(s,t,e)$ segment. Since $e \in \pi_\sigma(s,t)$, it holds that $\text{dist}(e,w,G \setminus \{e\}) \leq 2\sigma$. Finally, as $w \in SD(s,t,e)$ it implies that $e \in \pi_{\sigma'}(s,w)$. The claim then follows by Claim 15. ◀

Size. The first part adds the BFS trees w.r.t. $|R| = O(\sqrt{|S|n})$ vertices. In addition, in the gradual BFS constructions, for every edge $e \in \bigcup_{s \in S} \pi_\sigma(s,v)$, the vertex v adds at most one edge to H (corresponding to the last edge of $P(s,v,e)$). Since $|\bigcup_{s \in S} \pi_\sigma(s,v)| = O(\sqrt{|S|n})$, this adds $O(\sqrt{|S|}n^{3/2})$ edges.

Round Complexity.

▷ Claim 17. Each vertex t can learn the relevant edge set $\bigcup_{s\in S} \pi_{\sigma'}(s,t)$ within $O(\sqrt{n|S|})$ rounds.

Proof. For every $s \in S$, each edge e in T_s propagates down the tree for σ' time steps (i.e., until reaching all vertices at distance σ' from e). Focusing on a single-source s, each edge $e' = (x,y)$ needs to pass at most σ' messages, corresponding to the last σ' edges on the $\pi(s,y)$ path. Since there are $|S|$ sources, the total number of messages passing through a single edge is $|S| \cdot \sigma'$. Using pipeline all these messages can arrive in $O(\sqrt{n|S|})$ rounds. ◁

▶ **Lemma 18.** *W.h.p., at most $\ell = O(\log n)$ BFS tokens need to go through each edge, per phase.*

Proof. We show that w.h.p., in each phase number τ and for each edge $e' = (v,u)$, at most $O(\log n)$ BFS tokens will need to go through e' from v to u in phase τ. Note that the only BFS tokens passing over the edge $e' = (v,u)$ correspond to the BFS algorithms of $\texttt{BFS}(s, G \setminus \{e\})$ for $e \in \pi_{\sigma'}(s,u) \cup \pi_{\sigma'}(s,v)$. Thus each edge passes $O(|S| \cdot \sigma)$ tokens.

Each of the $O(|S| \cdot \sigma)$ permitted tokens passing through e' from v to u in phase τ satisfies that $\text{dist}(s,v,G \setminus \{e\}) + \tau_{s,e} = \tau$. Assuming that the starting phase $\tau_{s,e}$ is chosen uniformly at random from a range of size $3\sigma|S|$, the probability of this event is at most $1/(3\sigma \cdot |S|)$. Hence, over the set $3\sigma \cdot |S|$ permitted tokens, only 1 token, in expectation, is scheduled to go through from v to u in phase τ. If the random delay values $\tau_{s,e}$ were completely independent, by an application of the Chernoff bound, we would have that this number is at most $O(\log n)$, w.h.p. This will not be exactly true in our case, as we produce $\mathcal{SR}$ using a pseudo-random generators, but using k-wise independence on the generated string, for $k = \Theta(\log n)$ and from a result of Schmidt et al.[23], it is known that for this application of the Chernoff bound, it suffices to have k-wise independence between the random values, for $k = \Theta(\log n)$. ◀

We are now ready to complete the round complexity argument. By Claim 17, each vertex t computes its relevant edge set $\pi_{\sigma'}(s,t)$ within $O(D + \sqrt{|S|n})$ rounds. Within additional $O(\sqrt{|S|n})$ rounds, each vertex t can also learn the relevant edge sets of its neighbors. By Lemma 18, the computation of all BFS trees is implemented within $\widetilde{O}(D + \sigma \cdot |S|) = \widetilde{O}(D + \sqrt{|S|n})$ rounds. This completes the proof of Theorem 1.

4 Distributed Construction of Dual Failure Distance Preservers

In this section, we extend the construction of $\texttt{FT-MBFS}$ structures to support two edge failures. Throughout, For every $s,t \in S \times V$, and $e_1, e_2 \in G$, recall that $P(s,t,\{e_1,e_2\})$ is the unique s-t path in $G \setminus \{e_1,e_2\}$ chosen based on a consistent tie-breaking scheme (based on vertex IDs). For a given parameter σ, let $P_\sigma(s,t,F)$ be the σ-length suffix (ending at t) of the path. When $|P_\sigma(s,t,F)| \leq \sigma$, $P_\sigma(s,t,F)$ is simply $P(s,t,F)$. Set

$$\sigma_1 = (n/|S|)^{5/8} \text{ and } \sigma_2 = (n/|S|)^{1/4} .$$

We start by describing the algorithm based on the assumption that every vertex t has the following information:

- (I1) The distance $\text{dist}(s,t,e)$ for every $s \in S$ and every $e \in \pi_{2\sigma_2}(s,t)$.
- (I2) The path segment $P_{\sigma_2}(s,t,e)$ for every $s \in S$ and every $e \in \pi_{2\sigma_2}(s,t)$.

Note that in contrast to the FT-BFS construction of [11], the FT-MBFS algorithm of Theorem 1 computes the structure but not necessarily the distances. We therefore need to augment the algorithm by a procedure that computes the information (I1,I2) for all near faults (at distance σ_2 from t).

▶ **Lemma 19.** *There is a randomized algorithm that w.h.p. computes the information (I1,I2) for every vertex t within $\widetilde{O}((n/\sigma_1) \cdot \sigma_2 + (n/\sigma_1)^2 + D)$ rounds.*

In Subsec. 4.1 we describe the key construction. Then in Subsec. 4.2, we prove Lemma 19.

4.1 Distributed Alg. for Dual Failure FT-MBFS Structure (Under Assumption)

Before explaining the algorithm, we need the following definition, which extends Def. 13 to the dual failure setting.

▶ **Definition 20** (sensitive-detour of a Dual-Fault Replacement Path)**.** *A vertex t is* sensitive *to the triplet (s, e_1, e_2) if $P(s, t, \{e_1, e_2\}) \notin \{P(s, t, e_1), P(s, t, e_2)\}$. This necessarily implies that for a sensitive vertex it holds that $e_2 \in P(s, t, e_1)$* and *$e_1 \in P(s, t, e_2)$. For a given $P(s, t, \{e_1, e_2\})$ path, let w be the first vertex (closest to s) that is sensitive to (s, e_1, e_2). The sensitive detour $SD(s, t, \{e_1, e_2\})$ correspond to the segment $P(s, t, \{e_1, e_2\})[w, t]$.*

Set $\sigma = \sigma_2 = (n/|S|)^{1/4}$. The first step of the algorithm computes an FT-MBFS subgraph FT-MBFS$(R \cup S)$ where R is a randomly sampled set of $O(n \log n/\sigma)$ vertices. By Thm. 1, this can be done in $\widetilde{O}(\sqrt{|R|n} + D)$ rounds. The second step computes a subset of dual-failure replacement paths $\{P(s, t, \{e_1, e_2\}), s \in S, t \in V, e_1, e_2 \in E\}$ that satisfy certain properties. As in the single failure case, the computation of many of the replacement paths might be incomplete. The guarantee, however, would be that any $P(s, t, \{e_1, e_2\})$ replacement path whose last edge is not in FT-MBFS$(R \cup S)$ is fully computed by the algorithm. The set of replacement paths which the algorithm attempts to compute is defined by:

$$Q_t = \{(s, e_1, e_2) \mid e_1 \in P_\sigma(s, t, e_2) \text{ and } e_2 \in P_\sigma(s, t, e_1),\ s \in S\}.$$

By the assumption (I1,I2), each vertex t knows the last 2σ edges of the path $P(s, t, e)$ for every $s \in S$ and every $e \in \pi_{2\sigma}(s, t)$, it can compute Q_t (see Claim 22). Observe that $|Q_t| = |S| \cdot \sigma^2$. The algorithm starts by letting each vertex exchange its Q_t set with its neighbors. The BFS tokens BFS$(s, G \setminus \{e_1, e_2\})$ are permitted to pass from a vertex u to a vertex v only if $(s, e_1, e_2) \in Q_v$. To control the congestion due to the simultaneous constructions of multiple BFS trees, the vertices share a random string $\mathcal{SR}$. Each BFS algorithm BFS$(s, G \setminus \{e_1, e_2\})$ for every $e_1, e_2 \in G$ and $s \in S$ starts in phase τ_{s,e_1,e_2} chosen uniformly at random in the range $\{1, \ldots, \Theta(|S| \cdot \sigma^2)\}$. Using the seed $\mathcal{SR}$ and the IDs of s, e_1, e_2, each vertex can compute τ_{s,e_1,e_2}. These τ_{s,e_1,e_2} values are $O(\log n)$-wise independent.

Each BFS algorithm BFS$(s, G \setminus \{e_1, e_2\})$ then starts in phase τ_{s,e_1,e_2}, and proceeds in a speed of one hop per phase. Each phase consists of $\ell = \Theta(\log n)$ rounds. The rules for passing the BFS tokens of BFS$(s, G \setminus \{e_1, e_2\})$ are as follows:

- Each vertex v that is not sensitive[5] to e_1, e_2 sends the token BFS$(s, G \setminus \{e_1, e_2\})$ in round $\tau_{s,e_1,e_2} + \text{dist}(s, v, G \setminus \{e_1, e_2\})$ to every neighbor $u \in N(v)$ satisfying that $(s, e_1, e_2) \in Q_u$.

[5] In the analysis, we show that in the case where there is $u \in N(v)$ for which $(s, e_1, e_2) \in Q_u$, v can indeed detect that it is not sensitive.

- Every vertex v that is sensitive to (s, e_1, e_2) upon receiving the first BFS token $\texttt{BFS}(s, G \setminus \{e_1, e_2\})$ in phase i does as follows:
  - Let w be the minimum-ID vertex in $N(v)$ from which v has received the BFS token in that phase. Then, v adds the edge (w, v) to the output structure H.
  - v sends the token $\texttt{BFS}(s, G \setminus \{e_1, e_2\})$ in phase $i + 1$ to every neighbor $u \in N(v)$ satisfying that $(s, e_1, e_2) \in Q_u$.

This completes the description of the algorithm.

Analysis. Let $Q'_t \subset Q_t$ be defined by $Q'_t = \{(s, e_1, e_2) \mid e_1 \in P_{\sigma/2}(s, t, e_2) \text{ and } e_2 \in P_{\sigma/2}(s, t, e_1),\ s \in S\}$. Let w be the first sensitive vertex (see Def. 20) w.r.t. (s, e_1, e_2) on the replacement path $P(s, t, \{e_1, e_2\})$. Recall that $SD(s, t, \{e_1, e_2\}) = P(s, t, \{e_1, e_2\})[w, t]$ is the sensitive detour of $P(s, t, \{e_1, e_2\})$.

▶ **Observation 21.** *Any vertex $w' \in SD(s, t, \{e_1, e_2\})$ is sensitive to the two edges e_1, e_2.*

Proof. Recall that w is the first sensitive vertex on $P(s, t, \{e_1, e_2\})$, and thus the first vertex of the sensitive detour. Assume towards contradiction, that there exists a vertex $w' \in SD(s, t, \{e_1, e_2\})$ that is not sensitive to (s, e_1, e_2). Let $P \in \{P(s, w', e_1), P(s, w', e_2)\}$ be such that $P = P(s, w', \{e_1, e_2\})$. By the uniqueness of the shortest paths, we have that $P(s, t, \{e_1, e_2\})[s, w'] = P \circ P(s, t, \{e_1, e_2\})[w', t]$. We then have that $w \in P$ and thus $P[s, w] = P(s, w, \{e_1, e_2\})$ contradiction that w is the first sensitive vertex on $P(s, t, \{e_1, e_2\})$. ◀

▷ Claim 22. By knowing (I1) and (I2), each vertex t can compute the set Q_t.

To prove the correctness of the output structure, by Fact 9 we need to show that $\texttt{LastE}(P(s, t, \{e_1, e_2\}))$ is in H for every $s \in S$ and every $e_1, e_2 \in E$. Throughout, we consider a fixed replacement path $P(s, t, \{e_1, e_2\})$ and assume w.l.o.g. that $e_1 \in \pi(s, t)$ and $e_2 \in P(s, t, e_1)$.

▶ **Lemma 23.** *For every $(s, e_1, e_2) \notin Q'_t$ it holds that $\texttt{LastE}(P(s, t, \{e_1, e_2\})) \in \textit{FT-MBFS}(R)$.*

To complete the correctness argument, it remains to show that $\texttt{LastE}(P(s, t, \{e_1, e_2\})) \in H$ for every $(s, e_1, e_2) \in Q'_t$. We do it in two steps, depending on the length of the sensitive-detour.

▷ Claim 24. Let $(s, e_1, e_2) \in Q'_t$. If $|SD(s, t, \{e_1, e_2\})| \geq \sigma/3$, then $\texttt{LastE}(P(s, t, \{e_1, e_2\})) \in H$.

For now on, we consider $P(s, t, \{e_1, e_2\})$ paths such that $(s, e_1, e_2) \in Q'_t$ and with a short sensitive detour, i.e., $|SD(s, t, \{e_1, e_2\})| \leq \sigma/3$. We first show the following.

▷ Claim 25. Let $(s, e_1, e_2) \in Q'_t$ and $|SD(s, t, \{e_1, e_2\})| \leq \sigma/3$. Then, $(s, e_1, e_2) \in Q_{w'}$ for every $w' \in SD(s, t, \{e_1, e_2\})$.

Proof. Fix $w' \in SD(s, t, \{e_1, e_2\})$. Since the detour is short it holds that $\text{dist}(w', t, G \setminus \{e_1, e_2\}) \leq \sigma/3$ for every $w' \in SD(s, t, \{e_1, e_2\})$. In addition, since $e_2 \in P_{\sigma/2}(s, t, e_1)$, we have that

$$\text{dist}(e_2, w', G \setminus \{e_1\}) \leq \text{dist}(e_2, t, G \setminus \{e_1\}) + \text{dist}(t, w', G \setminus \{e_1, e_2\}) \leq \sigma\ . \tag{1}$$

As w' is sensitive, it holds that $e_2 \in P(s, w', e_1)$, and combining with Eq. (1) we have that $e_2 \in P_\sigma(s, w', e_1)$. In the same manner, since $e_1 \in P_{\sigma/2}(s, t, e_2)$ and $e_1 \in P(s, w', e_2)$, by the same reasoning we have that $e_1 \in P_\sigma(s, w', e_2)$. We conclude that $(s, e_1, e_2) \in Q_{w'}$. ◁

We next show that the BFS token $\mathtt{BFS}(s, G\setminus\{e_1,e_2\})$ arrives each vertex $w' \in SD(s,t,\{e_1,e_2\})$ in phase $\text{dist}(s,w',G\setminus\{e_1,e_2\}) + \tau_{s,e_1,e_2}$. Since for every vertex $w' \in SD(s,t,\{e_1,e_2\})$ it holds that $(s,e_1,e_2) \in Q_{w'}$, it is guaranteed that the BFS token $\mathtt{BFS}(s, G\setminus\{e_1,e_2\})$ arriving w' in $SD(s,t,\{e_1,e_2\})$ in phase i is sent to the next hop $w'' \in SD(s,t,\{e_1,e_2\})$ in phase $i+1$. Therefore it is sufficient to show that the first vertex, say w, on the sensitive-detour $SD(s,t,\{e_1,e_2\})$ receives the token $\mathtt{BFS}(s, G\setminus\{e_1,e_2\})$ in phase $\text{dist}(s,w',G\setminus\{e_1,e_2\}) + \tau_{s,\{e_1,e_2\}}$. Let q be the neighbor of w on $P(s,t,\{e_1,e_2\})$ not in $SD(s,t,\{e_1,e_2\})$.

▷ Claim 26. q sends to w (first vertex on the sensitive-detour) the BFS token $\mathtt{BFS}(s, G\setminus\{e_1,e_2\})$ in phase $\text{dist}(s,w,G\setminus\{e_1,e_2\}) + \tau_{s,e_1,e_2}$.

▶ **Corollary 27.** *For every path $P(s,t,\{e_1,e_2\})$ satisfying that (i) $(s,e_1,e_2) \in Q'_t$ and (ii) $|SD(s,t,\{e_1,e_2\})| \leq \sigma/3$, it holds that the detour $SD(s,t,\{e_1,e_2\})$ is fully computed by the algorithm (i.e., the BFS token propagates through all the vertices on the sensitive detour). Consequently, $\mathtt{LastE}(P(s,t,\{e_1,e_2\})) \in H$.*

Round Complexity. We next analyze the round complexity. The computation of the structure $\mathtt{FT\text{-}MBFS}(R \cup S)$ takes $O(\sqrt{(|R|+|S|)n} + D) = \widetilde{O}(n^{7/8} \cdot |S|^{1/8})$ rounds. Running the partially computed BFS trees $\mathtt{BFS}(s, G\setminus\{e_1,e_2\})$ takes in total $\widetilde{O}(D + (\sigma_2)^2 \cdot |S|)$. Combining with the round complexity of Lemma 19 yields the desired bound of $\widetilde{O}(D + |S|^{5/4}n^{3/4} + |S|^{1/8}n^{7/8})$.

Size. The total number of edges in $\mathtt{FT\text{-}MBFS}(R \cup S)$ is bounded by $O(\sqrt{|R|+|S|} \cdot n^{3/2})$. In addition, each vertex t adds at most $|Q_t| = O(|S| \cdot \sigma^2)$ edges to H. Plugging $\sigma = (n/|S|)^{1/4}$ and $|R| = O(n \log n/\sigma)$ yields the desired edge bound of $\widetilde{O}(|S|^{1/8}n^{15/8})$.

4.2 Learning Distances and Short RP Segments of Near Faults

In this subsection we fill in the missing piece of the algorithm by proving Lemma 19, and thus establishing Theorem 2. The computation of the information (I1,I2) for every vertex t is done in two key steps depending on the structure of the $P(s,t,e)$ path.

A replacement-path $P(s,t,e)$ for $e \in \pi_{\sigma_2}(s,t)$ is said to be *easy* if $|SD(s,t,e)| \leq \sigma_1$. Otherwise, the path $P(s,t,e)$ for $e \in \pi_{\sigma_2}(s,t)$ is *hard*.

Computing the information for easy replacement paths. We will present a somewhat stronger algorithm that computes (I1,I2) for every $P(s,t,e)$ paths satisfying that $e \in \pi_{\sigma_1}(s,t)$ (rather than just $e \in \pi_{\sigma_2}(s,t)$). The algorithm simply applied the second step of the single-failure $\mathtt{FT\text{-}MBFS}$ algorithm with parameter $\sigma = 8\sigma_1$. Recall that in this phase, a partial collection of replacement paths is computed which is characterized by the given parameter σ. By the proof of Lemma 16 (Case (3)), we have that each t knows $\text{dist}(s,t,e)$ for every *easy* replacement path. It therefore remains for it to learn also the σ_2-length suffix of these paths. We next show that this can be done by a simple extension of the algorithm.

▷ Claim 28. Within extra $\widetilde{O}(D + \sigma_1 \cdot \sigma_2 \cdot |S|)$ rounds, every vertex t can learn the σ_2-length suffix of every easy replacement path $P(s,t,e)$, $e \in \pi_{\sigma_1}(s,t)$ for every $s \in S$.

Computing the information for hard replacement paths. It remains to consider the hard replacement paths $P(s,t,e)$. I.e., paths for which $e \in \pi_{\sigma_2}(s,t)$ and their sensitive-detour is of length at least σ_1. (Unlike the previous algorithm, here we might not learn the distances $\text{dist}(s,t,G\setminus\{e\})$ for edges $e \in \pi_{\sigma_1}(s,t) \setminus \pi_{\sigma_2}(s,t)$.) We assume here that this step is applied already computing the information for the easy replacement paths.

Let R be a random sample of $O(n \log n/\sigma_1)$ vertices. The algorithm computes BFS trees $T_r = \texttt{BFS}(r, G)$ for every $r \in R$. In addition, each vertex also learns its last σ_2 edges on each $\pi(r, t, T_r)$ paths. Using the random delay approach, this can be done in $\widetilde{O}(D+(n \log n/\sigma_1)\cdot\sigma_2)$ rounds.

▶ **Lemma 29.** *One can compute LCA (Least Common Ancestor) labels in each BFS tree T_s, $s \in S$ in total time $\widetilde{O}(D + S)$. The size of each LCA label is $O(\log^2 n)$ bits (per tree T_s).*

Consider an hard replacement-path $P(s, t, e)$ and let q be the neighbor before w on the path, where w is the first sensitive vertex on $P(s, t, e)$. Let $e = (x, y)$. We claim the following, see Fig. 1 for an illustration.

▷ Claim 30. For every hard replacement path $P(s, t, e)$, there must be two vertices r_1, r_2 such that (i) $\text{dist}(r_1, r_2, G) \le \sigma_1/16$, (ii) r_1 is not sensitive to e and r_2 is sensitive to e and (iii) $e \notin \pi(r_1, r_2)$.

Proof. We claim that for every vertex w' appearing on the $(\sigma_1/8)$-length prefix of $SD(s, t, e)$ it holds that $e \notin \pi_{\sigma_1/8}(s, w')$. Assume towards contradiction otherwise, since $e \in \pi_{\sigma_1/8}(s, w')$ and $e \in \pi_{\sigma_2}(s, t)$ (and $\sigma_2 << \sigma_1$), the tree path between w' and t in T_s is free from e and has length at most $\sigma_1/4$. As $\text{dist}(w', t, G \setminus \{e\}) = |SD(s, t, e)[w', t'] \ge \sigma_1/2$, we end with a contradiction.

Next, let w be the first vertex on $SD(s, t, e)$ and let q be the vertex that appears just before w on $P(s, t, e)$. By the uniqueness of the shortest-path, $P(s, t, e) = \pi(s, q) \circ P(s, t, e)[q, t]$. We now claim that $|\text{dist}(s, q, G)| \ge \sigma_1/8-1$. Since $e \notin \pi_{\sigma_1/8}(s, w)$, it implies that $|\text{dist}(s, w, G)| \ge \sigma_1/8$ concluding that $|\text{dist}(s, q, G)| \ge \sigma_1/8 - 1$.

Therefore the $\sigma_1/32$ suffix of $\pi(s, q)$ contains a vertex $r_1 \in R$ that is not sensitive to e. The $\sigma_1/32$ prefix of the sensitive detour $SD(s, t, e)$ contains a vertex $r_2 \in R$ that is sensitive to e. Since the distance between r_1, r_2 on $P(s, t, e)$ is at most $\sigma_1/16$ and since $\text{dist}(e, r_2, G) \ge \sigma_1/8$, we conclude that $\text{dist}(r_1, r_2, G) = \text{dist}(r_1, r_2, G \setminus \{e\})$. ◁

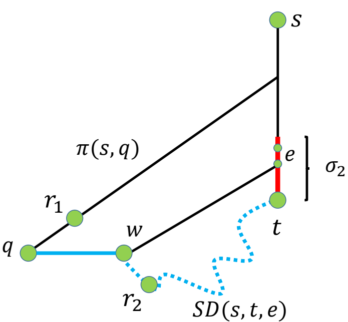


Figure 1 An illustration for the proof of Claim 30. Shown in an hard $P(s, t, e)$ path where $e = (x, y) \in \pi_{\sigma_2}(s, t)$. The vertex w is the first vertex on the sensitive detour, thus the entire $P(s, t, e)[w, t]$ is contained in the vertex set of $T_s(y)$, where is the subtree of T_s rooted at y. Dashed edges correspond to the path segment $SD(s, t, e)$. Since e is very close to t, but e is somewhat far from the vertices on the prefix of the sensitive detour, there are two vertices r_1, r_2 that satisfy the properties of the claim.

The algorithm then lets each vertex r in R send to all vertices in the graph the following:

- The list of the distances $\text{dist}(r, r', G)$ for every r' in R.
- The $\widetilde{O}(1)$-length bit LCA label of r in each tree T_s.

Overall, the total information sent is $\widetilde{O}(|R|^2+|S|\cdot|R|)$. This can be done in $\widetilde{O}(|R|^2+|S|\cdot|R|+D)$ rounds by a simple pipeline.

Now every vertex t is doing the following calculations for every edge $e \in \pi_{\sigma_1}(s,t)$ for which it did not receive a BFS token $\texttt{BFS}(s,t,G\setminus\{e\})$ in the first phase of the algorithm (of handling the easy replacement paths). Using the LCAs of all vertices in R with respect to T_s, it computes the set R_e^+ and R_e^- where $R_e^- = \{r \in R \mid e \notin \pi(s,r)\}$ and $R_e^+ = R \setminus R_e^-$. Note that $e \in \pi(s,r)$ only if the LCA of r and t is *below* the failing edge e. Since t has the $2\sigma_1$-length suffix of its $\pi(s,t)$ path, it can detect if the LCA is below the edge e. Let

$$\mathrm{dist}(s,t,G\setminus\{e\}) = \min_{r_1\in R_e^-} \min_{r_2\in R_e^+,\mathrm{dist}(r_1,r_2,G)\le\sigma_1/16} \mathrm{dist}(s,r_1,G)+\mathrm{dist}(r_1,r_2,G)+\mathrm{dist}(r_2,t,G)\ .$$

Let $r_1^* \in R_e^-$ and $r_2^* \in R_e^+$ be the vertices that minimize the $\mathrm{dist}(s,t,G\setminus\{e\})$. Then, the t lets $P_{\sigma_2}(s,t,e) = \pi_{\sigma_2}(r_2^*,t)$. This completes the description of the algorithm, the complete proof of Lemma 19, Cor. 3 and 4 are deferred to the full version.

References

1. Noga Alon, Shiri Chechik, and Sarel Cohen. Deterministic combinatorial replacement paths and distance sensitivity oracles. In *46th International Colloquium on Automata, Languages, and Programming, ICALP 2019, July 9-12, 2019, Patras, Greece*, pages 12:1–12:14, 2019.
2. Davide Bilò, Fabrizio Grandoni, Luciano Gualà, Stefano Leucci, and Guido Proietti. Improved purely additive fault-tolerant spanners. In *Algorithms-ESA 2015*, pages 167–178. Springer, 2015.
3. Greg Bodwin, Fabrizio Grandoni, Merav Parter, and Virginia Vassilevska Williams. Preserving distances in very faulty graphs. In *44th International Colloquium on Automata, Languages, and Programming (ICALP 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.
4. Gilad Braunschvig, Shiri Chechik, David Peleg, and Adam Sealfon. Fault tolerant additive and (μ, α)-spanners. *Theor. Comput. Sci.*, 580:94–100, 2015.
5. Keren Censor-Hillel, Telikepalli Kavitha, Ami Paz, and Amir Yehudayoff. Distributed construction of purely additive spanners. *Distributed Computing*, 31(3):223–240, 2018.
6. Shiri Chechik and Sarel Cohen. Near optimal algorithms for the single source replacement paths problem. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 2090–2109, 2019.
7. Shiri Chechik and Ofer Magen. Near optimal algorithm for the directed single source replacement paths problem. *CoRR*, abs/2004.13673, 2020.
8. Michael Elkin and Shaked Matar. Near-additive spanners in low polynomial deterministic CONGEST time. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*, pages 531–540, 2019.
9. Yuval Emek, David Peleg, and Liam Roditty. A near-linear-time algorithm for computing replacement paths in planar directed graphs. *ACM Transactions on Algorithms (TALG)*, 6(4):1–13, 2010.
10. Mohsen Ghaffari. Near-optimal scheduling of distributed algorithms. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC*, pages 3–12, 2015.
11. Mohsen Ghaffari and Merav Parter. Near-optimal distributed algorithms for fault-tolerant tree structures. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2016, Asilomar State Beach/Pacific Grove, CA, USA, July 11-13, 2016*, pages 387–396, 2016.
12. Fabrizio Grandoni and Virginia Vassilevska Williams. Improved distance sensitivity oracles via fast single-source replacement paths. In *2012 IEEE 53rd Annual Symposium on Foundations of Computer Science*, pages 748–757. IEEE, 2012.

13 Manoj Gupta and Shahbaz Khan. Multiple source dual fault tolerant bfs trees. In *44th International Colloquium on Automata, Languages, and Programming (ICALP 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

14 Frank Thomson Leighton, Bruce M Maggs, and Satish B Rao. Packet routing and job-shop scheduling ino (congestion+ dilation) steps. *Combinatorica*, 14(2):167–186, 1994.

15 Enrico Nardelli, Guido Proietti, and Peter Widmayer. Finding the most vital node of a shortest path. *Theoretical computer science*, 296(1):167–177, 2003.

16 Enrico Nardelli, Ulrike Stege, and Peter Widmayer. Low-cost fault-tolerant spanning graphs for point aets in the euclidean plane, 1997.

17 Merav Parter. Dual failure resilient bfs structure. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, pages 481–490, 2015.

18 Merav Parter. Vertex fault tolerant additive spanners. *Distributed Computing*, 30(5):357–372, 2017.

19 Merav Parter and David Peleg. Sparse fault-tolerant BFS structures. *ACM Trans. Algorithms*, 13(1):11:1–11:24, 2016.

20 David Peleg. *Distributed Computing: A Locality-sensitive Approach.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

21 Seth Pettie. Distributed algorithms for ultrasparse spanners and linear size skeletons. In *the Proc. of the Int'l Symp. on Princ. of Dist. Comp. (PODC)*, pages 253–262, 2008.

22 Liam Roditty and Uri Zwick. Replacement paths and k simple shortest paths in unweighted directed graphs. *ACM Transactions on Algorithms (TALG)*, 8(4):1–11, 2012.

23 Jeanette P Schmidt, Alan Siegel, and Aravind Srinivasan. Chernoff-hoeffding bounds for applications with limited independence. *SIAM Journal on Discrete Mathematics*, 8(2):223–250, 1995.

24 Oren Weimann and Raphael Yuster. Replacement paths and distance sensitivity oracles via fast matrix multiplication. *ACM Transactions on Algorithms (TALG)*, 9(2):1–13, 2013.

Singularly Optimal Randomized Leader Election

Shay Kutten
Faculty of Industrial Engineering and Management,
Technion – Israel Institute of Technology, Haifa, Israel
kutten@technion.ac.il

William K. Moses Jr.
Faculty of Industrial Engineering and Management,
Technion – Israel Institute of Technology, Haifa, Israel
wkmjr3@gmail.com

Gopal Pandurangan
Department of Computer Science, University of Houston, TX, USA
gopal@cs.uh.edu

David Peleg
Department of Computer Science and Applied Mathematics,
Weizmann Institute of Science, Rehovot, Israel
david.peleg@weizmann.ac.il

Abstract

This paper concerns designing distributed algorithms that are *singularly optimal*, i.e., algorithms that are *simultaneously* time and message *optimal*, for the fundamental leader election problem in networks. Our main result is a randomized distributed leader election algorithm for *asynchronous complete* networks that is essentially (up to a polylogarithmic factor) singularly optimal. Our algorithm uses $O(n)$ messages with high probability[1] and runs in $O(\log^2 n)$ time (with high probability) to elect a unique leader. The $O(n)$ message complexity should be contrasted with the $\Omega(n \log n)$ lower bounds for the deterministic message complexity of leader election algorithms (regardless of time), proven by Korach, Moran, and Zaks (TCS, 1989) for asynchronous algorithms and by Afek and Gafni (SIAM J. Comput., 1991) for synchronous networks. Hence, our result also separates the message complexities of randomized and deterministic leader election. More importantly, our (randomized) time complexity of $O(\log^2 n)$ for obtaining the optimal $O(n)$ message complexity is significantly smaller than the long-standing $\tilde{\Theta}(n)$ time complexity obtained by Afek and Gafni and by Singh (SIAM J. Comput., 1997) for message optimal (deterministic) election in asynchronous networks. Afek and Gafni also conjectured that $\tilde{\Theta}(n)$ time would be optimal for message-optimal asynchronous algorithms. Our result shows that randomized algorithms are significantly faster.

Turning to *synchronous* complete networks, Afek and Gafni showed an essentially singularly optimal deterministic algorithm with $O(\log n)$ time and $O(n \log n)$ messages. Ramanathan et al. (Distrib. Comput. 2007) used randomization to improve the message complexity, and showed a randomized algorithm with $O(n)$ messages but still with $O(\log n)$ time (with failure probability $O(1/\log^{\Omega(1)} n)$). Our second result shows that synchronous complete networks admit a *tightly* singularly optimal randomized algorithm, with $O(1)$ time and $O(n)$ messages (both bounds are optimal). Moreover, our algorithm's time bound holds with certainty, and its message bound holds with high probability, i.e., $1 - 1/n^c$ for constant c.

Our results demonstrate that leader election can be solved in a simultaneously message and time-efficient manner in asynchronous complete networks using randomization. It is open whether this is possible in asynchronous general networks.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Mathematics of computing → Probabilistic algorithms; Mathematics of computing → Discrete mathematics

Keywords and phrases Leader election, Asynchronous systems, Randomized algorithms, Singularly optimal, Complete networks

[1] Throughout, "with high probability" means with probability at least $1 - 1/n^c$, for a constant $c > 0$.

© Shay Kutten, William K. Moses Jr., Gopal Pandurangan, and David Peleg;
licensed under Creative Commons License CC-BY

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.22

Related Version A full version of the paper is available at [27], https://arxiv.org/abs/2008.02782.

Funding *Shay Kutten*: This work was supported in part by the Bi-national Science Foundation (BSF) grant 2016419.
William K. Moses Jr.: This work was supported in part by the BSF grant 2016419 and in part by a Technion fellowship.
Gopal Pandurangan: G. Pandurangan was supported, in part, by NSF grants CCF-1527867, CCF-1540512, IIS-1633720, CCF-1717075, and BSF grant 2016419.
David Peleg: Supported in part by the US-Israel Binational Science Foundation grant 2016732.

1 Introduction

Leader election is a classical and fundamental problem in distributed computing with numerous applications; see e.g., [31, 12, 1, 18, 23, 25, 26, 29, 48, 28, 32, 33, 47, 20, 34, 39, 46, 50, 24, 42, 3, 43, 16, 30, 2]. The goal is to select a unique node, called the *leader*, from a set of nodes. An *arbitrary* subset of nodes can *wake up spontaneously at arbitrary times* and start the election algorithm by sending messages over the network. When the algorithm terminates, a *unique* node v must be elected as leader and be *known to all nodes*.

Election is important both theoretically and because of the multiple applications such as implementing databases and data centers [19, 9, 49], locks [7], file servers [14, 10], broadcast and multicast [41, 8], and virtually every global task [5]. In many of these applications, the network is treated as being virtually complete. With the advent of large-scale and resource-constrained networks such as peer-to-peer systems [22, 21, 4, 44, 45, 52] and ad hoc and sensor networks (e.g., [13, 51]), it is often desirable to achieve low cost and scalable leader election. Leader election has been studied extensively over the years in various distributed computing models starting with the standard CONGEST model of networks (in particular, ring, complete networks, and arbitrary networks), radio networks, peer-to-peer networks, population protocols, and programmable matter to name a few.

Our goal in this paper is to design leader election algorithms in distributed networks such that each is efficient with respect to *both of* the two fundamental complexity measures, namely *messages* and *time*. Unfortunately, designing distributed network algorithms that are *simultaneously* time- and message-efficient has proved to be a challenging task. Consequently, research in the last three decades has focused mainly on optimizing either one of the two measures separately, typically at the cost of neglecting the other. There has been significant recent progress in obtaining algorithms that are essentially optimal in both measures (or at least work well under both measures to the extent possible) for various problems such as leader election, minimum spanning tree and shortest paths [28, 38, 11, 17]. In particular, as defined in [38] (see also [15]), the following two notions are of interest for any given problem:

- **Singular optimality:** A problem enjoys *singular optimality* if it has a distributed algorithm that is optimal (or at least optimal up to a polylogarithmic factor) with respect to both measures simultaneously.
- **Time-message trade-off:** A problem exhibits a *time-message trade-off* if it fails to admit a singularly optimal solution, namely, algorithms of better time complexity for it necessarily incur higher message complexity and vice versa.

The singularly optimal results mentioned earlier for leader election, minimum spanning tree, and shortest paths [28, 38, 11, 17], crucially apply only to *synchronous* networks. A main motivation for this work is to study whether leader election admits singularly optimal

algorithms or exhibits a time-message tradeoff in *asynchronous* networks. Unlike synchronous networks which admit a leader election algorithm which is singularly optimal (up to a logarithmic factor) [28], it is not known whether asynchronous networks admit such an algorithm (see "Background and Prior Work").

In this paper, we focus on the classical problem of leader election in *complete* networks, which itself has been studied extensively for nearly four decades. In such a network, every pair of nodes is connected by a bidirectional communication link and in the beginning, no node has any knowledge of any other nodes, including their identities (if any). Like many prior works (e.g., [1, 47, 18, 12, 2] and others), this paper focuses on the more challenging *asynchronous* model and provides the first-known singularly optimal algorithm.

It is clear that the best possible time bound for leader election in complete networks is *constant* if there is no restriction on the message complexity. The situation with respect to the message complexity is more nuanced. (The message lower bound is trivially $\Omega(n)$ since the leader has to be known to all nodes.) For *deterministic* algorithms, there is a well-known lower bound of $\Omega(n \log n)$ messages (even in the synchronous setting when an adversary decides which nodes to wake up and when) [1, 26, 25] where n is the number of network nodes. Moreover, Afek and Gafni [1] show that any deterministic algorithm that is message optimal (i.e., takes $\Theta(n \log n)$ messages) needs $\Omega(\log n)$ time; they present such a deterministic algorithm that takes $O(n \log n)$ messages and $O(\log n)$ time in *synchronous* networks. Hence for synchronous networks, there exists a deterministic algorithm that is essentially singularly optimal (up to a logarithmic factor).

The situation is less clear in the asynchronous setting and for randomized solutions. First, the $\Omega(n \log n)$ message lower bound was shown for deterministic algorithms, and it was not clear whether by using randomization one can breach this bound. Second, Afek and Gafni [1] present an *asynchronous* leader election algorithm that takes $O(n \log n)$ messages (which is message optimal), but takes $O(n)$ time. They conjecture that this is the best possible time bound for message-optimal asynchronous algorithms. Singh [47] (mildly) disproves this conjecture by presenting an algorithm that runs in $O(n/\log n)$ time and is also message optimal ($O(n \log n)$ messages). This is still a far cry from being singularly optimal, as the time bound is essentially linear. When it comes to the use of randomness, Afek and Matias [2] develop a randomized algorithm that succeeds with probability $1 - \varepsilon$ and runs in $O(\log n)$ time using $O((n/\varepsilon)\log^2(1/\varepsilon))$ messages.[2] If we allow for a constant probability of success, i.e., if we set ε to a fixed constant, then the algorithm runs in $O(\log n)$ rounds and uses $O(n)$ messages. However, for randomized algorithms, it is typically desired that algorithms succeed with high probability. In such a scenario if we set ε to $1/n^c$ for some constant $c > 0$, the message complexity significantly increases to $O(n^{1+c}\log^2 n)$.

The results of the current paper (detailed later on) *break* the long-standing barrier of $\Omega(n \log n)$ messages for deterministic algorithms and show that there exist singularly optimal randomized asynchronous algorithms that succeed with high probability.

Distributed Computing Model. We consider a system represented as an undirected complete graph $G = (V, E)$, $|V| = n$, similar to the models of [1, 18, 23, 25, 26], except that processors can access *private unbiased coins*. Our upper bounds do not require unique

[2] In the paper, it is claimed that they develop an algorithm with termination detection that runs in $O(\log n)$ time for complete graphs. However, it is unclear if indeed the running time is $O(\log n)$ for an algorithm with termination detection or rather $O(n)$ time when termination detection is required.

identities, and in particular, nodes can be anonymous. If nodes do have unique identifiers, then we assume, as in prior works on complete networks (see [1]), that nodes initially do not know the unique identifiers of other nodes.[3]

In the synchronous communication setting, the computation is divided into discrete lockstep time units called *rounds*, and every message sent over an edge arrives at the receiver after a fixed delay of one time unit, namely, at the end of the current round. In contrast, in the *asynchronous* communication setting, messages sent on edges incur unpredictable but finite delay, in an error-free and FIFO manner (i.e., messages will arrive in sequence). Nevertheless, for the sake of time analysis it is assumed that a message takes *at most one time unit* to be delivered. For both modes, we make the usual (and here, very reasonable) assumption that local computation within a node is instantaneous and free. We assume the standard $\mathcal{CONGEST}$ model [40], where a node can send at most one $O(\log n)$ bit message on each edge in each round.

Following the standard assumption in asynchronous protocols (see [1, 12, 47]), nodes are initially asleep. A node enters the execution when it is woken up by the environment (at most once), or upon receiving messages from awakened neighbors. In the asynchronous setting, once a node enters execution, it performs all the computations required of it by the algorithm, sends out messages to neighbors as specified by the algorithm. In the synchronous setting, we assume the above *adversarial* wake up assumption as well.[4]

The message complexity of an algorithm is the worst-case total number of $O(\log n)$ bit messages sent during its execution. The time complexity is the worst-case total number of time units since the first node is woken up to the last message transmission due to the algorithm. Note that a time unit in the synchronous model corresponds to one round, whereas in the asynchronous model it is an upper bound on the transmission time of a message over an edge. Hence, it is generally more difficult to design efficient algorithms in asynchronous systems than in synchronous systems.

Following the standard approach to modeling distributed networks (see [1]), we represent the environmental uncertainties by means of an *adversary* controlling some of the execution parameters. Specifically, we assume an *adversarial wake up* mode, where node wake-up is scheduled by an adversary (who may decide to keep some nodes dormant). A node can also be woken up by receiving messages from other nodes. In addition to the wake-up schedule, the adversary also decides for how long to delay each message. These decisions are done *adaptively*, i.e., when the adversary makes a decision to wake up a node or delay a message, it has access to the results of all previous coin flips. Finally, recalling that initially the nodes are unaware of which neighbor is connected to each of their outgoing edges, the adversary also controls the graph structure (i.e., the mapping of outgoing edges to neighbors). Here, we consider the adversary to be *oblivious*, i.e., if nodes use randomness to choose an outgoing edge, the adversary must choose endpoints for all such outgoing edges prior to the first use of randomness in this way.[5]

---

[3] Otherwise (in the KT_1 model [6], where each node knows the unique identifiers of other nodes), leader election is trivial in complete networks.

[4] This should be contrasted with the *simultaneous wake up* model, where all nodes are assumed to be awake at the beginning of computation; this is typically assumed in design of synchronous protocols (see e.g., [1, 29, 28]).

[5] To appreciate the implications of allowing the adversary to construct the graph *after* seeing the random choices, see the lower bound proof techniques for message complexity in [26, 1]. These techniques are for deterministic algorithms, but will work when the adversary has adaptive edge mapping abilities.

Background and additional Prior Work. The complexity of the leader election problem and (especially deterministic) algorithms for it have been very well-studied in distributed networks. Various algorithms and lower bounds are known in different models with synchronous/asynchronous communication and in networks of varying network topologies such as a cycle, a complete graph, or some arbitrary topology (e.g., see [20, 23, 28, 29, 34, 39, 46, 50] and the references therein). The problem was first studied in context of a ring network by Le Lann [31] and discussed for general graphs in the influential paper of Gallager, Humblet, and Spira [12]. Kutten et al. [28] presented a singularly optimal (up to a logarithmic factor) randomized leader election algorithm for general *synchronous* networks that ran in $O(D)$ time and used $O(m \log n)$ messages (where D, m, and n are the network diameter, number of edges, and the number of nodes respectively). We note that $\Omega(D)$ and $\Omega(m)$ are lower bounds for time and messages for leader election even for randomized algorithms [28]. It is not known whether similar bounds can be achieved for general *asynchronous* networks, although one can obtain algorithms that are separately time optimal [39] and message optimal [12].

Leader election in the class of *complete networks* – which is the focus of this paper – has come to occupy a special position of its own and has been extensively studied [1, 2, 18, 23, 25, 26, 29, 48]; see also [32, 33, 47] for leader election in complete networks where nodes have a sense of direction. While $\Omega(n)$ is an obvious lower bound on the message complexity of leader election when the leader's identity should be known for all nodes, an $\tilde{O}(\sqrt{n})$ (i.e., sublinear) message complexity can be obtained for a related but different problem in *synchronous* complete networks with *simultaneous wake up* [29] where we *do not* require that the nodes not elected know who is the leader (nor which of their ports lead to it).[6] The above result crucially uses randomization to break the linear message complexity bound that applies for deterministic algorithms.

The study of leader election algorithms is usually concerned with both message and time complexity. Korach et al. [24], Humblet [18], Peterson [42] and Afek and Gafni [1] presented $O(n \log n)$ message algorithms for *asynchronous* complete networks. Korach, Kutten, and Moran [23] presented a general method plus applications to various classes of graphs including complete networks. Afek and Matias [2] similarly presented a general method with application to a complete graph.

Afek and Gafni (as a part of presenting a tradeoff between time and message complexity) showed that the time complexity of a message optimal *synchronous* algorithm was $\Theta(\log n)$, while for a message optimal *asynchronous* algorithm, they only demonstrated an $O(n)$ time upper bound (improving previous time bounds for message optimal algorithms). They conjectured that *in the asynchronous case, time complexity of any message optimal algorithm is* $\Omega(n)$. Singh [48] (as a part of presenting a different tradeoff), presented a somewhat better ($O(n/\log n)$) time for message optimal asynchronous algorithms, but still posed as an important open problem the question whether $\Omega(n/\log n)$ time was optimal for such algorithms. The current paper demonstrates that this is not the case, at least for randomized algorithms that succeed w.h.p., by giving a $O(\log^2 n)$ time bound for message optimal $O(n)$ algorithm.

The above mentioned papers on "*sense of direction*" demonstrated that when the nodes possessed additional knowledge on the topology, it was possible to reduce the number of messages to $O(n)$. In the same vein, note that for *deterministic* algorithms in *synchronous* networks and under the strong assumption of *simultaneous wake up*, there exists an $O(n)$

[6] This variant of leader election is sometimes called "implicit", as opposed to the version studied here, where all nodes need to know the identity of the leader.

messages algorithm [28]. In contrast, an $\Omega(n \log n)$ message lower bound in synchronous networks was shown by Afek and Gafni [1] under the more common assumption of adversarial wake up. The message complexity in the current paper is $O(n)$ without using sense of direction or simultaneous wakeup assumptions (but using randomized algorithms).

For anonymous networks under some reasonable assumptions, deterministic leader election was shown to be impossible, using symmetry arguments [3]. Randomization comes to the rescue in this case; random rank assignment is often used to assign unique identifiers, as done herein. Randomization also allows us to beat the lower bounds for deterministic algorithms, albeit at the risk of a small chance of error. For example, Afek and Matias [2] developed a randomized algorithm that succeeded with probability $1 - \varepsilon$ and ran in $O(\log n)$ time using $O((n/\varepsilon) \log^2(1/\varepsilon))$ messages. It should be noted that Singh's [47] deterministic algorithm allowed a trade-off between time and memory such that it was possible to achieve a running time as low as $O(\log n)$ in exchange for more messages ($O(n^2/\log n)$). By setting ε to $1/n^c$, for a constant $c > 0$, we see that Afek and Matias's algorithm succeeds with high probability and takes less messages ($O(n^{1+c} \log^2 n)$) for the same running time.

Turning to *synchronous* complete networks, Afek and Gafni showed an essentially singularly optimal deterministic algorithm with $O(\log n)$ time and $O(n \log n)$ messages. Ramanathan et al. used randomization to improve the message complexity, and showed a randomized leader election algorithm for synchronous networks that could err with probability $O(1/\log^{\Omega(1)} n)$ with time $O(\log n)$ and $O(n)$ messages[7] [43]. That paper also extends the synchronous algorithm to work for *partially* synchronous networks, where message delays are bounded. It also surveys some related papers about randomized algorithms in other models that use more messages for performing leader election [16] or related tasks (e.g., quorum systems, Malkhi et al. [35]). In the context of self-stabilization, a randomized algorithm with $O(n \log n)$ messages and $O(\log n)$ time until stabilization was presented in [30].

Table 1 Comparison of previous upper bound results for leader election in complete networks of n nodes along with our contributions. The variables $2 \leqslant c \leqslant n$, $\log n \leqslant k \leqslant n$, and $0 < \varepsilon < 1$ are parameters provided to the respective algorithms.

Paper	Message Complexity	Time Complexity	Communication Mode	Type of Solution
[24]	$O(n \log n)$	$O(n \log n)$	Asynchronous	Deterministic
[1]	$O(n \log n)$	$O(n)$	Asynchronous	Deterministic
[47]	$O(nk)$	$O(n/k)$	Asynchronous	Deterministic
[2]	$O(n \log^2(1/\varepsilon)/\varepsilon)$	$O(\log n)$	Asynchronous	Randomized, success prob. $1 - \varepsilon$
This paper	$O(n)$ w.h.p.	$O(\log^2 n)$ w.h.p.	Asynchronous	Randomized, success w.h.p.*
[1]	$O(cn \log_c n)$	$O(\log_c n)$	Synchronous	Deterministic
[43]	$O(n)$	$O(\log n)$	Synchronous	Randomized, success prob. $1 - O(1/\log^{\Omega(1)} n)$
This paper	$O(n)$ w.h.p.	$O(1)$	Synchronous	Randomized, success w.h.p.*
*The algorithm always succeeds when nodes have unique identifiers (as opposed to anonymous networks).				

Our Main Results. The main focus of this paper is on studying how randomization can help in designing singularly optimal algorithms for leader election in asynchronous as well as synchronous networks. Our results are summarized in Table 1. Our main result is a randomized asynchronous leader election algorithm for complete networks that runs in

[7] In contrast, the synchronous algorithm presented in the current paper succeeds with high probability, its time complexity is constant, and its complexity bounds hold with probability $1 - O(1/n^{\Omega(1)})$.

$O(\log^2 n)$ time and uses only $O(n)$ messages to elect a unique leader with high probability (Section 2). This is a significant improvement over the $\Omega(n \log n)$ messages needed for any deterministic algorithm, and an even larger improvement over the time complexity of previous message optimal ($O(n \log n)$) deterministic algorithms for asynchronous networks, which required $O(n)$ or $O(n/\log n)$ time [1, 47]. In addition, we show that for the synchronous setting too, we can obtain a singularly optimal algorithm that tightly matches the best possible asymptotic time and message lower bounds. We present a randomized algorithm in synchronous networks that takes $O(1)$ time and $O(n)$ messages with high probability (see Section 3). Note that our algorithms succeed w.h.p. in anonymous networks and always succeed when each node has a unique identifier.

Our results, while providing near-optimal message and time bounds for complete asynchronous networks, are a step towards designing singularly optimal algorithms for *general* asynchronous networks.

2 A Randomized Algorithm for Asynchronous Networks & Analysis

In this section, we first give some high level intuition of how we use randomization to overcome the barriers of time and message complexity posed to deterministic asynchronous algorithms. Subsequently, we present a randomized algorithm that solves leader election in $O(\log^2 n)$ time with high probability using $O(n)$ messages with high probability.

High level intuition behind the use of randomness. In [1], an $\Omega(n \log n)$ message lower bound is presented. A key idea in the proof is that the adversary is able to control the destination of messages sent by a node. It can do this for a deterministic algorithm because the adversary may predict which edges will be used in the course of the algorithm and construct the initial graph accordingly. Intuitively, if multiple clusters of nodes are active, a leader cannot be determined until they interact. If the adversary can consistently control the destination of messages over unknown links, the adversary may blow up the number of messages until the clusters interact. We bypass this issue by having nodes choose the edges they use to communicate at random, similar to the idea used in [29]. Since the adversary cannot predict which edge will be used, it cannot initially construct an undesirable graph. Our second use of randomness is to have each node, once awake, choose its $RANK$ uniformly at random from $[1, n^4]$. This helps us achieve a significantly smaller running time compared to the deterministic case. If $RANK$s are not randomly chosen but deterministically assigned, then the adversary may wake up nodes such that it takes $O(n)$ time from the time the first node is woken up until the algorithm terminates. This is because, in our algorithm, a node's $RANK$ plays an important role in deciding if it will go on to become the leader when interacting with another node or if it will stop trying to be the leader.

The Randomized algorithm. We now describe the algorithm. Each awake node may play up to two roles in the algorithm, namely (i) a candidate and (ii) a referee. During the process, candidates attempt to progress towards becoming the leader. The position of a candidate u in this process is represented by the pair $\langle RANK_u, PH_u \rangle$, where $RANK_u$ is u's *rank* and PH_u is its current *phase*. Intuitively, we say that candidate v is *ahead of* candidate u, and u is *behind* v, if v has a higher phase number, or, in the case of a tie, v has a higher RANK. Formally, we say that $v \gg u$ if either (i) $PH_v > PH_u$, or (ii) $PH_v = PH_u$ and $RANK_v > RANK_u$.[8]

[8] If nodes have unique identifiers, then in the case of two nodes in the same phase with the same rank, their unique identifiers can be used to break ties. Furthermore, a node will append its unique identifier

During the process, candidates gradually either *progress* or *retire*, eventually resulting in a single un-retired candidate who then becomes the leader. Essentially, a candidate u retires upon encountering another candidate v that is ahead of it. Such an encounter occurs when a referee learns of both candidates, and it is the referee's task to make one of the candidates retire. (The referee is typically a third party, but may also be u or v.) Hence, candidates fundamentally drive the algorithm forward, while referees serve a complementary role of guardians, assisting candidates in deciding whether to retire or to proceed. Each node is in one of three candidate states Candidate, Non-elected, Elected, stored in the variable CAND-STATE, corresponding to whether the node is still a candidate in the running to become a leader, is retired, or is elected, respectively.

If a node is woken up by the adversary, then it plays the role of a candidate and may eventually become the leader. If a node is woken up by a message from a neighbor, then it is immediately considered to be a retired candidate. When a node receives a message, depending on the type of message received, the node may either act in the role of a candidate or in the role of a referee, with an appropriate procedure called to handle the message. Thus, during the execution, a node either acts as just a referee or as both a candidate and a referee.

Each node u, once woken up, checks if it was woken by the adversary (i.e., spontaneously wakes up itself) or by a message from a neighbor. If u was woken by a neighbor, u sets its candidate state to Non-elected. If u was woken by the adversary, u sets its candidate state to Candidate, chooses an integer in $[1, n^4]$ uniformly at random as its ID_u and attempts to progress. The initialization is described in Algorithm 1 in the full version.

A candidate u progresses through at most $K = \lceil \log \sqrt{4n \log n} \rceil + 1$ phases until it either changes its candidate state to Non-elected or completes phase K and stays in candidate state Candidate, in which case it declares itself leader. At the beginning of each phase $i = 1$ to $K - 1$, a candidate node u chooses a set $\mathcal{S}$ of $\min\{10 \cdot 2^i, \lceil \sqrt{4n \log n} \rceil\}$ nodes uniformly at random designated as its *referees* and seeks their approval to progress.[9] In phase $i = K$, u sets $\mathcal{S} \leftarrow V$, i.e. all nodes of the network form set $\mathcal{S}$. Node u sends a *request* message $\langle RANK_u, PH_u, \text{REQUEST} \rangle$ to each referee in $\mathcal{S}$.

When node u receives replies from all nodes in $\mathcal{S}$, it checks if any of those replies is a decline of the form $\langle RANK_u, PH_u, \text{DECLINED} \rangle$. If so, u changes its candidate state to Non-elected.[10] In case u received no declines, it may proceed. In particular, if $i < K$, then u increases PH_u to $i + 1$ and starts the next phase, and if $i = K$ and u still retains candidate state Candidate, then u declares itself as leader, namely, it changes its candidate state to Elected, broadcasts the final announcement $\langle RANK_u, 0, \text{LEADER} \rangle$ and terminates. The above process is described in Procedure 2 in the full version.

Each referee r helps candidates retire (until only one is left) by comparing candidate pairs, keeping the more advanced one, and instructing the other to retire. Referee r can be in one of four referee states stored in the variable REF-STATE.

- REF-STATE $= C0$ holds when r has not been approached by any candidate yet.
- REF-STATE $= C1$ holds when r has one approved candidate, referred to as its *chosen* candidate, and has declined every other candidate that approached it so far. A record containing the position of the chosen candidate v, namely, $\mathcal{P}(v) = \langle RANK_v, PH_v \rangle$, is kept in the variable Chosen.

---

to its $RANK$ to avoid ambiguity.

[9] We assume that node u treats itself as a referee as well in addition to the nodes of $\mathcal{S}$.

[10] The candidate state of u can also be changed to Non-elected as a result of processing a DECIDE message, described later.

- REF-STATE $= C2$ holds when r currently keeps track of two candidates: the chosen v (in the variable Chosen) and a contender w (in the variable Contender), such that $w \gg v$, and all other candidates that approached r so far were declined. Moreover, a *dispute* is currently in progress between v and w. This state is typically reached when r has a chosen candidate v that was approved by it, and later it gets a request from another candidate w such that $w \gg v$. In this situation, r cannot decline w (since it is ahead of its current chosen), but at the same time it cannot approve w, since it may be that v has progressed in the meantime, and it is now ahead of w. To resolve this uncertainty, r declares a dispute, and sends a DECIDE message containing w's position $\langle RANK_w, PH_w \rangle$ to v, asking it to make a comparison between w and itself (based on its current phase PH_v). While waiting for v's response, r keeps a record containing the details of w, namely, $\mathcal{P}(w) = \langle RANK_w, PH_w \rangle$, in the variable Contender.
- REF-STATE $= C3$ is similar to $C2$, i.e., it holds when r currently keeps track of a chosen v and a contender w, all other candidates that approached r so far were declined, and a dispute is currently in progress. The difference, however, is that the on-going dispute does not involve w. Rather, it is between v and some *previous* contender z such that $w \gg z \gg v$. This state is typically reached when a new candidate w approaches r while r is in referee state $C2$ with a dispute in progress between a chosen candidate v and a contender $z \gg v$, and r discovers that $w \gg z$. This allows r to decline the current contender z immediately (since even if the outcome of the dispute favors z over v, the new candidate w is ahead of z, so z must retire). Now w takes z's place as the contender.

If r receives a message $\langle RANK_u, PH_u, \text{REQUEST} \rangle$ from node u over some edge e, it responds as follows.

- If r is in referee state $C0$, then it registers u as its chosen candidate, sends back an approval message, and switches its referee state to $C1$.
- If r is in referee state $C1$, then the following sub-cases may apply: If the current chosen is also u (from an earlier phase), then r updates the record stored in Chosen. If it is another node v such that $v \gg u$, then r sends u a decline message. Otherwise (i.e., if $u \gg v$), r registers u as the contender and initiates a dispute by sending a DECIDE message to v, requesting it to compare its current position with that of u. It also switches its referee state to $C2$.
- If r is in referee state $C2$, signifying that a dispute is in progress, then r compares the new candidate u with the current contender w. If $u \ll w$ then r declines u's request (and thus retires u). Otherwise (i.e., if $u \gg w$), r retires w, registers u as the new contender, and switches its referee state to $C3$.
- If r is in referee state $C3$, then it does the same as in referee state $C2$.

The pseudocode for the referee's actions on receiving a request is given in Procedure 3 in the full version.

When a node v which is currently the chosen candidate of the referee r (but may have possibly retired since the time it was approved by r) receives from r an $\langle RANK_u, PH_u, \text{DECIDE} \rangle$ message, it must decide whether to end its candidacy (if it is still a candidate) or that of the other candidate. To do so, it compares its own current position with that of the contender u.

- If v is in candidate state Non-elected or $v \ll u$, then v returns the message $\langle RANK_u, PH_u, \text{WINS} \rangle \circ \langle RANK_v, PH_v, \text{LOSES} \rangle$.
- Otherwise (v is still in candidate state Candidate and is ahead of u), it returns the message $\langle RANK_u, PH_u, \text{LOSES} \rangle \circ \langle RANK_v, PH_v, \text{WINS} \rangle$.

Pseudocode for the chosen's actions on receiving a DECIDE message from a referee is given in Procedure 4 in the full version.

Once the chosen's reply message is received by r, it is processed as follows.

- If v replied that the contender u has won the dispute, r sends an approval message to the current contender (which is u if the referee state is $C2$, and another candidate if the referee state is $C3$), makes it the chosen, and switches to referee state $C1$.
- If v replied that it has progressed beyond the contender u, so u has lost the dispute and must retire, then there are two sub-cases to consider. If the referee state is $C2$, r sends a decline message to the contender u, and switches to state $C1$. Now suppose the referee state is $C3$ and the current contender is some candidate w. If v's current position is such that $v \gg w$, then r sends a decline message to w and switches to referee state $C1$. Otherwise ($w \gg v$), a new dispute is required, this time between v and w.

The referee's actions on receiving a reply from the chosen about an ongoing dispute are given in Procedure 5 in the full version.

Analysis of the algorithm. We establish the following theorem.

▶ **Theorem 1.** *Consider a complete anonymous network G of n nodes in the $\mathcal{CONGEST}$ model. Assume communication is asynchronous with adversarial wakeup. Then there is a randomized algorithm to solve leader election with high probability in $O(\log^2 n)$ time with high probability using $O(n)$ messages with high probability. If each node has a unique identifier, then the algorithm always succeeds. All nodes terminate at the end of the algorithm.*

We show three properties: exactly one leader is chosen w.h.p. for anonymous networks and always when nodes have unique identifiers and all nodes subsequently terminate, the time complexity is $O(\log^2 n)$ w.h.p., and the message complexity is $O(n)$ w.h.p.

Before we continue with the proof, we make an important observation.

▶ **Observation 1.** *For each node u that participates in the algorithm, if u begins phase i, then u will finish phase i.*

Observation 1 is used implicitly in the remaining proof whenever a node is mentioned to finish some phase and possibly perform some calculation as a result of completing that phase.

▶ **Lemma 2.** *By the end of the algorithm, exactly one node is in candidate state* Elected *w.h.p. for anonymous networks and always when nodes have unique identifiers and the remaining nodes are in candidate state* Non-elected. *Furthermore, all nodes eventually terminate.*

Proof. Let us consider a network where each node has a unique identifier. We show that exactly one leader is always chosen by arguing that exactly one $RANK$ is chosen as the leader. It is clear to see that if we then consider an anonymous network where each node chooses its $RANK$ from $[1, n^4]$ and nodes do not have access to unique identifiers to break ties, the $RANK$ chosen as the leader will belong to not more than one node w.h.p.

We prove by induction that at least one node is a candidate at the end of phase i for $1 \leqslant i \leqslant K$. For the induction basis, consider, for the sake of the proof, that when a node wakes up, it is in phase 0, but it immediately moves to phase 1. Clearly, this does not change the outcome of the algorithm. Now the lemma holds for phase zero since at least one node is woken up by the adversary initially, and the induction is proven with phase zero as base case.

Assume that the claim holds true up to the end of some phase $k < K$. Let u be the candidate with largest RANK in phase $k+1$. If u is not retired by any other node in phase $k+1$, then it will be in candidate state Candidate at the end of phase $k+1$. Hence the claim holds for $k+1$. Otherwise, if u is retired in phase $k+1$, then necessarily either u came into contact with (i) some candidate u' in a higher phase $PH_{u'} > k+1$ or a referee to such

a candidate, (ii) a candidate u'' with higher rank $RANK_{u''} > RANK_u$ in the same phase $k+1$ or a referee to such a candidate, or (iii) a candidate v that completed all phases and is still in candidate state Candidate or a message from such a candidate v. In all cases, the induction claim still holds true for $k+1$.

We subsequently show that exactly one node will change its candidate state to Elected. By the previous claim, at least one candidate completes phase $K-1$. Let u_{max} be the node with the largest RANK that is still a candidate at the beginning of phase K. In phase K, u_{max} will approach all other nodes, receive an approval from each of them, and subsequently change its candidate state to Elected. Any other candidate v will approach u_{max}, receive a DECLINED message from it, and thus change its candidate state to Non-elected. Thus exactly one node will change its candidate state to Elected and broadcast a $\langle \cdot, 0, \text{LEADER} \rangle$ message.

Finally, we argue that all nodes terminate. It is clear that if exactly one node changes its candidate state to Elected and broadcasts a $\langle \cdot, 0, \text{LEADER}\}$ message, then that node terminates the algorithm. All other nodes change their candidate state to Non-elected and terminate the algorithm upon receiving this message. Thus, the proof is complete. ◀

We now prove the time complexity bound. First, we prove the following useful lemma about the amount of time one phase for a candidate takes.

▶ **Lemma 3.** *If a node u starts some phase i as a candidate, then it exits the phase (either by increasing its phase to $i+1$ or by retiring) within $O(1)$ time.*

Proof. For any candidate u, for each phase i, the largest delay until that phase ends occurs when candidate u sends a message to a referee r, which in turn sends a message to its current chosen v. Subsequently, v sends a reply to r, which in turn sends a reply to u. If the congestion on both these edges is $O(1)$ messages in the time interval from start to end of phase i, then the total time duration of the phase is $O(1)$. Let $e_{u,r}$ and $e_{r,v}$ denote edges between u and r and between r and v resp. We show at most $O(1)$ messages need to be transmitted on both $e_{u,r}$ and $e_{r,v}$ during phase i, thus allowing each phase to take $O(1)$ time.

Consider edge $e_{u,r}$. In phase i, u can send an invite to r and receive a reply to the invite from r. Furthermore, u can also be the chosen for r and receive a DECIDE message from r and subsequently send back a reply.[11] Additionally, r may also be a candidate with u as its referee or r may be the chosen for u, resulting in at most a doubling of messages over the edge. Thus at most $O(1)$ messages are sent across that edge for phase i of u. Now consider edge $e_{r,v}$. A similar analysis as above renders $O(1)$ an upper bound on the messages on the edge. Thus, we see that each phase takes $O(1)$ time units. ◀

In order to bound the running time, we also make use of the following useful property from [37], denoted in the reference as Problem C.2 in Appendix C.6. We slightly modify the statement to suit our needs.

▶ **Lemma 4.** *(Problem C.2 in [37]) Let $a_1, a_2, \ldots, a_n$ be a sequence of n values. Each value a_i is independently and randomly chosen from a fixed distribution $\mathcal{D}$. Let $m_i = \max\{a_1, a_2, \ldots, a_i\}$, i.e., the maximum of the first i values. Let random variable Y denote the number of times the maximum value is updated, i.e., the number of times $m_i \neq m_{i+1}$. Then $E[Y] = O(\log n)$.*

[11] Recall that once some node u, in phase i, is the chosen of some node r and receives a DECIDE message, either u is retired or r learns that u is in phase i and will not send any further messages from other candidates in phase $< i$. If another DECIDE message is subsequently sent to u from r, then either u wins (implying it is in a higher phase), or else u is retired and no further messages are sent along the edge.

▶ **Lemma 5.** *The running time of the algorithm is $O(\log^2 n)$ with high probability.*

Proof. For any given execution of the algorithm, eventually one candidate, C_ℓ, woken up by the adversary, becomes the leader. We show two properties for any such C_ℓ. First, once C_ℓ is woken up by the adversary, the algorithm takes an additional at most $O(K)$ time until C_ℓ broadcasts that it is the leader and the algorithm terminates for every node. Second, we bound the time from when the first node is woken up by the adversary to when candidate C_ℓ is woken up as $O(K \cdot \log n)$ with high probability.

Once C_ℓ is woken up, it must complete K phases before it broadcasts that it is the leader. By Lemma 3, each of its phases takes $O(1)$ time to complete. Thus, once C_ℓ is woken up by the adversary, it takes at most $O(K)$ time until all nodes terminate.

Now, we bound the time until C_ℓ is woken up. For this proof, we say that two candidates u and v *encounter* each other at time t when a referee (which could be either u or v itself) is made aware of both u and v for the first time. If, as a result of the encounter, v is retired, then we call u the *winner* of the encounter and v the *loser*. Note the following observation.

▶ **Observation 2.** *Consider some execution of the algorithm where a candidate u while in phase k retires candidate w. Subsequently, u reaches phase i and is retired by another candidate v in phase j. When u and v encounter each other, then either $j > i$ or v has a larger $RANK$ than u and is also in phase i. Furthermore, from the time u retired w, at most $O(i - k)$ time units have passed until the encounter between u and v (by Lemma 3) and an additional $O(1)$ time units pass until u retires.*

Observation 2 has the following implications. Either the winner of an encounter has a larger $RANK$ than the loser or else the phase number of the winner is larger than that of the loser when they encounter one another. Furthermore, the amount of time that passes between two encounters involving a common candidate can be bounded by the difference in phases of that candidate plus $O(1)$ time units.

We now construct a new graph where the nodes are the subset of the nodes that are woken up by the adversary (and which become candidates). There is an edge from candidate C_i to C_j if candidate C_i encounters and subsequently is retired by C_j. Notice that this graph is a directed acyclic graph since once a candidate C_i is retired by some other node C_j, C_i cannot go on to retire C_j or any candidate that retires C_j. Notice also that a candidate may encounter multiple other candidates, resulting in nodes with an in degree or out degree (or both) that is > 1. Finally, note that candidates that do not retire any other candidates are sources and C_ℓ is a sink. Consider a path in the graph from a source to C_ℓ such that the node first woken up by the adversary is one of the nodes in this path. If there are multiple such paths, choose one arbitrarily. Label the nodes from the source to the node just before C_ℓ as $C_1, C_2, \ldots, C_w$. Let C_1 be in some phase p at the time of its encounter with C_2. Let R be the time that pass from the first encounter until the final encounter in the chain. Then, the total time from when the first node was woken up until C_ℓ was woken up is upper bounded by $O(p + R) = O(K + R)$. We now bound the value of R.

With each of the candidates C_i, $1 \leqslant i \leqslant w$, associate a bit b_i which indicates how C_i was retired. Specifically, b_i is set to 0 if C_{i+1} was in a higher phase than C_i at the time of the encounter.[12] Bit b_i is set to 1 if C_{i+1} has a higher $RANK$ than C_i and is in the same phase as it at the time of the encounter. By Observation 2 and its implications, we see that every bit b_i, $1 \leqslant i \leqslant w$ is set to 0 or 1.

[12] Note that we are referring to the actual phases C_i and C_{i+1} are in at this time, not necessarily the phase numbers stored in the referee when it was first aware of both of them, as one of these values may be outdated. Also note that if $i = w$, then C_{i+1} refers to C_ℓ.

Consider the bit string $B = b_1 b_2 \ldots b_w$. We will show that R is upper bounded $O(K+|B|)$. In order to bound the size of B, we first bound the number of 0's that can be present in B, and then we bound the number of 1's that can be present between any two 0's in B.

Denote by P_i the phase of C_i when it was retired. Observe that, regardless of whether $b_i = 0$ or 1, the phase P_{i+1} of the subsequent candidate C_{i+1} can never be less than that of the candidate C_i it just retired. Furthermore, when $b_i = 0$, $P_{i+1} > P_i$. Thus, there can be at most K bits set to 0 in B. We now bound the number of bits set to 1 between any two bits that are set to 0.

Between any two bits set to 0, at most $n-1$ nodes can be woken up by the adversary to trigger an encounter leading to a bit being set to 1. An encounter where a bit is set to 1 involves the $RANK$ of the awakened node being higher than that of the currently considered candidate, resulting in the higher $RANK$ node becoming the currently considered candidate. By Lemma 4, we see that on expectation, $O(\log n)$ such encounters can thus be triggered. Applying a simple Chernoff bound, we see that $O(\log n)$ is in fact a high probability upper bound on the number of such encounters, and by extension the number of bits set to 1 between any two bits set to 0. Since there are at most K bits set to 0, $|B| = O(K \cdot \log n)$.

Each encounter of two candidates contributes $O(1)$ time towards the running time. Furthermore, we must account for the time between encounters as well. Between any two encounters, the phase of a candidate may increase. We have already seen that if candidate C_i is in phase P_i, then for subsequent candidates C_j where $j > i$, $P_j \geqslant P_i$. From Lemma 3, we see that for any candidate a phase takes $O(1)$ time. Therefore, the total number of time units between all encounters is at most $O(K)$ time units. Thus, R can be upper bounded by $O(K + |B|) = O(K \cdot \log n)$ and the total running time is $O(K + R) = O(\log^2 n)$ time with high probability. ◀

We now prove that the message complexity of the randomized algorithm is $O(n)$ with high probability. We first show that the number of candidates that can participate in every phase is reduced by a factor of four from one phase to the next up to phase $\rho = K - \lceil \log \log n \rceil - 5$ with high probability.

▶ **Lemma 6.** *The total number of candidates that participate in phase i of the algorithm is at most $\lceil n/4^{i-1} \rceil$ with high probability for $1 \leqslant i \leqslant \rho$.*

Proof. We prove the claim by induction. Initially, even if the adversary wakes up all nodes, at most n candidates participate in phase 1. Assume that the claim holds true until some phase $k < \rho$. We prove that the claim holds in phase $k+1$ when the number of candidates participating in phase k is upper bounded by $\lceil n/4^{k-1} \rceil$.

Notice that if $\leqslant \lceil n/4^k \rceil$ candidates participate in phase k, then the claim holds immediately for phase $k+1$. Now, let us assume that the number of candidates participating in phase k lies in $(\lceil n/4^k \rceil, \lceil n/4^{k-1} \rceil]$.

Organize the candidates in increasing order of their $RANK$s. Group the top one-sixteenth of these candidates into the set *top* and group the remaining candidates into the set *bottom*. We ignore the retirement of candidates from *top* in phase k by assuming that none of them are retired in this phase and show that a sufficient number of candidates from *bottom* are retired for the claim to hold in phase $k+1$.[13] In this phase, there are at least $1/16 \cdot \lceil n/4^k \rceil$

[13] Notice that the claim is an upper bound on the number of candidates in each phase. By showing that the claim holds when none of the candidates from *top* is retired, it is easy to see that the claim also holds when at least one candidate from *top* is retired.

candidates in *top*, each of which makes $10 \cdot 2^k$ requests.[14] Thus, altogether, candidates from *top* make at least $m_0 = (\lceil n/4^k \rceil/16) \cdot 10 \cdot 2^k \geqslant 5n/2^{k+3}$ requests uniformly at random.[15]

Consider a candidate u from *bottom*. Let r be some specific referee approached by u.

Call r *top-free* if none of the candidates from *top* has approached r and r itself is not in *top*. Then the probability that r is top-free is at most $p_0 = (1-1/n)^{m_0} \cdot 15/16 \leqslant (1-1/n)^{m_0}$. Note that u will be retired if even one of its requests is to a non-top-free referee. Therefore, the probability that u is not retired in this phase is at most $p_0^{10 \cdot 2^k} \leqslant (1-1/n)^{(5n/2^{k+3}) \cdot 10 \cdot 2^k} \leqslant e^{-25/4}$. Thus, denoting the number of candidates from *bottom* that are not retired by NR_B, we have $\mathbb{E}[NR_B] \leqslant (15/16) \cdot \lceil n/4^{k-1} \rceil \cdot e^{-25/4} \leqslant (15/2)e^{-25/4} \cdot n/4^k \leqslant (1/16) \cdot n/4^k$. As NR_B is the sum of independent Bernoulli trials, we can apply a Chernoff bound (second bound of Theorem 4.4 in [36]). When $k < \rho$, the probability that NR_B exceeds twice the expected value is upper bounded by $1/n^3$. Thus the total number of candidates that can participate in phase $k+1$ with high probability is at most $(1/16) \cdot \lceil n/4^{k-1} \rceil + (2/16) \cdot (n/4^k) \leqslant \lceil n/4^k \rceil$.

Note that for each phase, the upper bound on the number of candidates who are not retired holds with probability $1 - 1/n^3$, assuming that the bound held in the previous phase. Applying a union bound over all phases of the induction, we see that this upper bound holds for phase ρ with probability $1 - O(1/n^2)$, i.e. w.h.p., and for each previous phase with a larger probability. Thus the claim is true for all $1 \leqslant i \leqslant \rho$ with high probability. ◀

We next show that in the final phase of the algorithm, only a single candidate will not be retired with high probability.

▶ **Lemma 7.** *At the end of phase $K-1$, exactly one node will be in candidate state* Candidate *with high probability.*

Proof. Consider the candidate with the largest RANK, u_{max}, in phase $K-1$. We show that for each candidate $v \neq u_{max}$ in this phase, the intersection of referees approached by v and u_{max} is non-zero with high probability. Thus u_{max} will retire each such candidate v with high probability. Notice that $10 \cdot 2^K \geqslant \lceil \sqrt{4n \log n} \rceil$ and so each candidate in this phase makes exactly $\lceil \sqrt{4n \log n} \rceil$ requests. Then

$$\mathbb{P}[v \text{ is not retired by } u_{max}] = (1-1/n)^{\lceil \sqrt{4n \log n} \rceil \cdot \lceil \sqrt{4n \log n} \rceil} \leqslant 1/n^4.$$

By Lemma 6, there are at most $\lceil 2^{13} \log n \rceil$ candidates participating in phase ρ. As the number of candidates participating in subsequent phases cannot increase beyond this value, it acts as an upper bound for the number of candidates participating in phase $K-1$.

We see that the probability that any of these candidates is not retired is at most $1/n^3$ through the use of a union bound. Thus, w.h.p. u_{max} retires every other candidate in this phase. Thus only u_{max} ends the phase in candidate state Candidate w.h.p. ◀

▶ **Lemma 8.** *The message complexity of the algorithm is $O(n)$ messages with high probability.*

Proof. Let $\psi = \lfloor \log(1/10 \cdot \lceil \sqrt{4n \log n} \rceil) \rfloor$. We start by analysing the message complexity that can be "charged" to candidates in phases 1 to ψ. In each phase $i \leqslant \psi$, each candidate u generates and receives up to $2 \cdot 10 \cdot 2^i$ messages from its referees. Furthermore, for each of its

[14] Notice that for all phases i that the claim applies to, $10 \cdot 2^i \leqslant \lceil \sqrt{4n \log n} \rceil$. Thus, any candidate in one of these phases will make $10 \cdot 2^i$ requests.

[15] Note that the requests by a candidate in a single phase are made without repetition, i.e. a candidate does not send more than one request along the same edge in a single phase. During the subsequent analysis, we consider them to be made with repetition so as to simplify the analysis. This assumption only decreases the probability of a candidate from *bottom* inviting a referee that was approached by one of the *top* candidates and thus this simplifying assumption is acceptable.

referees in a given phase i and from all previous phases, a candidate may receive a message to decide candidacy, resulting in at most $\sum_{j=1}^{i}\sum_{k=1}^{j} 2\cdot 10\cdot 2^k \leqslant \sum_{j=1}^{i} 2\cdot 10\cdot 2^{j+1} = 5\cdot 2^{i+4}$ additional messages received and generated by the candidate in this phase. (For a candidate u and a referee r chosen by u in phase i or a previous one, only one DECIDE message will be sent from r to to u.)

In each phase $1 \leqslant i \leqslant \rho$, there are at most $\lceil n/4^{i-1}\rceil$ candidates w.h.p. by Lemma 6. As calculated earlier, we have $5\cdot 2^{i+2} + 5\cdot 2^{i+4} = 25\cdot 2^{i+2}$ messages per candidate in each phase i, resulting in a total of $O(n)$ messages w.h.p. generated across all candidates in all phases.

In phases $\rho+1 \leqslant i \leqslant \psi$, there are at most $\lceil 2^{13}\log n\rceil$ candidates in each phase with high probability by Lemma 6, each making $10\cdot 2^i$ requests in that phase. Thus, the total number of messages over these phases is $\sum_{i=\rho+1}^{\psi}\lceil 2^{13}\log n\rceil 25\cdot 2^{i+2} = O(\sqrt{n}\log^{3/2} n)$ w.h.p.

Now we analyze the message complexity attributed to candidates in phases ψ and above. In each phase $i > \psi$, each candidate u generates and receives $2\cdot\lceil\sqrt{4n\log n}\rceil$ messages from its referees. Furthermore, for each of its referees in a given phase i and from all previous phases, u may receive a message to decide candidacy, resulting in at most $2\lceil\sqrt{4n\log n}\rceil(i-\psi)$ additional messages due to DECIDE messages from referees of phases $j > \psi$, plus $5\cdot 2^{\psi+4}$ additional messages due to DECIDE messages from referees of phases $j \leqslant \psi$. Thus, at most $2\cdot(i+5-\psi)\cdot\lceil\sqrt{4n\log n}\rceil$ messages are received and generated by u in this phase.

In phases $i > \psi$, there are most $\lceil 2^{13}\log n\rceil$ candidates in each phase w.h.p. by Lemma 6. Thus, the total number of messages across all these phases is at most $\sum_{i=\psi+1}^{K}\lceil 2^{13}\log n\rceil\cdot 2\cdot(i+5-\psi)\cdot\lceil\sqrt{4n\log n}\rceil = O(\sqrt{n}\log^{5/2} n)$ w.h.p.

By Lemma 7, exactly one candidate will complete all the first $K-1$ phases and remain in candidate state Candidate w.h.p. This candidate will generate $O(n)$ messages in phase K.

Thus, over all phases of the algorithm, there are totally $O(n)$ messages w.h.p. ◀

3 A (Tightly) Singularly Optimal Synchronous Algorithm

We also develop a message and time optimal algorithm for the synchronous setting, i.e., an algorithm that takes $O(1)$ time and $O(n)$ messages with high probability. Note that these upper bounds are tight (hence the term “tightly” in the section title).

The following theorem captures the properties of the algorithm. Due to space constraints, we refer the reader to the full version of the paper for the description of the algorithm and the proof of the theorem.

▶ **Theorem 9.** *Consider a complete anonymous network G of n nodes in the $\mathcal{CONGEST}$ model, with synchronous communication and adversarial wakeup. There is a randomized algorithm to solve leader election w.h.p. using $O(n)$ messages w.h.p. in 9 rounds (deterministically). (If nodes have unique identifiers, then the algorithm always elects a leader.)*

4 Conclusion

In the asynchronous setting, no singularly optimal *deterministic* leader election algorithm is known to exist. In contrast, we have shown that using randomization, a singularly optimal algorithm (whose message complexity is asymptotically optimal and whose time complexity is optimal up to a polylogarithmic factor) can be obtained.

One open question is whether we can improve the time complexity of our message-optimal ($O(n)$) randomized asynchronous algorithm from the current $O(\log^2 n)$ time, to say, $O(\log n)$ time. Also, can we obtain such a (almost) singularly optimal *deterministic* algorithm?

Another important question is to find out whether (and when) one can construct time and message-efficient asynchronous algorithms also for *general* graphs, in particular algorithms that are (essentially) singularly optimal. In general graphs, that would mean algorithms with $O(m)$ (or, at least, $\tilde{O}(m)$) messages and $\tilde{O}(D)$ (or, at least, $\tilde{O}(D)$) time. This was shown for the case of synchronous networks in [28].

References

1 Yehuda Afek and Eli Gafni. Time and message bounds for election in synchronous and asynchronous complete networks. *SICOMP*, 20(2):376–394, 1991.

2 Yehuda Afek and Yossi Matias. Elections in anonymous networks. *Information and Computation*, 113(2):312–330, 1994.

3 Dana Angluin. Local and global properties in networks of processors (extended abstract). In *STOC*, pages 82–93, 1980. doi:10.1145/800141.804655.

4 John Augustine, Gopal Pandurangan, Peter Robinson, and Eli Upfal. Towards robust and efficient distributed computation in dynamic peer-to-peer networks. In *SODA*, pages 551–569, 2012.

5 Baruch Awerbuch, Israel Cidon, and Shay Kutten. Communication-optimal maintenance of replicated information. In *Proceedings [1990] 31st Annual Symposium on Foundations of Computer Science*, pages 492–502. IEEE, 1990.

6 Baruch Awerbuch, Oded Goldreich, Ronen Vainish, and David Peleg. A trade-off between information and communication in broadcast protocols. *J. ACM*, 37:238–256, 1990.

7 Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

8 Miguel Castro, Peter Druschel, A-M Kermarrec, and Antony IT Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in communications*, 20(8):1489–1499, 2002.

9 Tushar D Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407, 2007.

10 Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):1–26, 2008.

11 Michael Elkin. A simple deterministic distributed mst algorithm, with near-optimal time and message complexities. In *PODC*, pages 157–163, 2017.

12 Robert G. Gallager, Pierre A. Humblet, and Philip M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. Programming Languages & systems (TOPLAS)*, 5(1):66–77, January 1983. doi:10.1145/357195.357200.

13 Saurabh Ganeriwal, Ram Kumar, and Mani B. Srivastava. Timing-sync protocol for sensor networks. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems, SenSys 2003, Los Angeles, California, USA, November 5-7, 2003*, pages 138–149, 2003.

14 Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, 2003.

15 Robert Gmyr and Gopal Pandurangan. Time-message trade-offs in distributed algorithms. In *32nd International Symposium on Distributed Computing, DISC 2018, New Orleans, LA, USA, October 15-19, 2018*, pages 32:1–32:18, 2018.

16 Indranil Gupta, Robbert van Renesse, and Kenneth P. Birman. A probabilistically correct leader election protocol for large groups. In *Proceedings of the 14th International Conference on Distributed Computing*, DISC '00, pages 89–103, 2000. URL: http://dl.acm.org/citation.cfm?id=645957.675964.

17 Bernhard Haeupler, D Ellis Hershkowitz, and David Wajc. Round-and message-optimal distributed graph algorithms. In *PODC*, pages 119–128, 2018.

18 Pierre A. Humblet. Selecting a leader in a clique in $O(n \log n)$ messages. Memo, Lab. for Information & Decision Systems, MIT, 1984.

19 Michael Isard. Autopilot: automatic data center management. *ACM SIGOPS Operating Systems Review*, 41(2):60–67, 2007.

20 Maleq Khan, Fabian Kuhn, Dahlia Malkhi, Gopal Pandurangan, and Kunal Talwar. Efficient distributed approximation algorithms via probabilistic tree embeddings. In *Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*, PODC '08, pages 263–272, New York, NY, USA, 2008. ACM. `doi:10.1145/1400751.1400787`.

21 Valerie King, Jared Saia, Vishal Sanwalani, and Erik Vee. Scalable leader election. In *SODA*, pages 990–999, 2006.

22 Valerie King, Jared Saia, Vishal Sanwalani, and Erik Vee. Towards secure and scalable computation in peer-to-peer networks. In *FOCS*, pages 87–98, 2006. `doi:10.1109/FOCS.2006.77`.

23 Ephraim Korach, Shay Kutten, and Shlomo Moran. A modular technique for the design of efficient distributed leader finding algorithms. *ACM Trans. Programming Languages & Systems (TOPLAS)*, 12(1):84–101, January 1990. `doi:10.1145/77606.77610`.

24 Ephraim Korach, Shlomo Moran, and Shmuel Zaks. Tight lower and upper bounds for some distributed algorithms for a complete network of processors. In *Proceedings of the third annual ACM symposium on Principles of distributed computing*, pages 199–207, New York, NY, USA, 1984. ACM. `doi:10.1145/800222.806747`.

25 Ephraim Korach, Shlomo Moran, and Shmuel Zaks. The optimality of distributive constructions of minimum weight and degree restricted spanning trees in a complete network of processors. *SIAM J. Computing*, 16(2):231–236, 1987. `doi:10.1137/0216019`.

26 Ephraim Korach, Shlomo Moran, and Shmuel Zaks. Optimal lower bounds for some distributed algorithms for a complete network of processors. *Theoretical Computer Science*, 64(1):125–132, 1989. `doi:10.1016/0304-3975(89)90103-5`.

27 Shay Kutten, William K. Moses Jr., Gopal Pandurangan, and David Peleg. Singularly optimal randomized leader election. *CoRR*, abs/2008.02782, 2020. `arXiv:2008.02782`.

28 Shay Kutten, Gopal Pandurangan, David Peleg, Peter Robinson, and Amitabh Trehan. On the complexity of universal leader election. *J. ACM*, 62:7, 2015.

29 Shay Kutten, Gopal Pandurangan, David Peleg, Peter Robinson, and Amitabh Trehan. Sublinear bounds for randomized leader election. *Theoretical Computer Science*, 561:134–143, 2015.

30 Shay Kutten and Dmitry Zinenko. Low communication self-stabilization through randomization. In *DISC*, pages 465–479. Springer, 2010.

31 Gérard Le Lann. Distributed systems - towards a formal approach. In *IFIP Congress*, pages 155–160, 1977.

32 Michael C. Loui, Teresa A. Matsushita, and Douglas B. West. Election in a complete network with a sense of direction. *IPL*, 22(4):185–187, 1986.

33 Michael C. Loui, Teresa A. Matsushita, and Douglas B. West. Election in a complete network with a sense of direction. *IPL*, 28(6):327, 1988. `doi:10.1016/0020-0190(88)90181-0`.

34 Nancy Lynch. *Distributed Algorithms.* Morgan Kaufman Publishers, Inc., San Francisco, USA, 1996.

35 Dahlia Malkhi, Michael Reiter, and Rebecca Wright. Probabilistic quorum systems. In *PODC 1997*, pages 267–273, New York, NY, USA, 1997. ACM. `doi:10.1145/259380.259458`.

36 Michael Mitzenmacher and Eli Upfal. *Probability and computing: randomization and probabilistic techniques in algorithms and data analysis.* Cambridge university press, 2017.

37 Gopal Pandurangan. Distributed network algorithms. `https://sites.google.com/site/gopalpandurangan/dna`. Accessed: 2020-02-14.

38 Gopal Pandurangan, Peter Robinson, and Michele Scquizzato. A time-and message-optimal distributed algorithm for minimum spanning trees. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, pages 743–756. ACM, 2017.

39 David Peleg. Time-optimal leader election in general networks. *J. Parallel & Distributed Computing*, 8(1):96–99, 1990. doi:10.1016/0743-7315(90)90074-Y.

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

41 Radia Perlman. *Interconnections: bridges, routers, switches, and internetworking protocols.* Addison-Wesley Professional, 2000.

42 Gary Peterson. Efficient algorithms for elections in meshes and complete graphs. Technical report, TR 140, Dept. of CS, Univ. Rochester, 1985.

43 Murali Krishna Ramanathan, Ronaldo A. Ferreira, Suresh Jagannathan, Ananth Grama, and Wojciech Szpankowski. Randomized leader election. *Distributed Computing*, pages 403–418, 2007.

44 Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *SIGCOMM 2001*, pages 161–172, New York, NY, USA, 2001. ACM. doi:10.1145/383059.383072.

45 Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, Middleware '01, pages 329–350. Springer-Verlag, 2001. URL: http://dl.acm.org/citation.cfm?id=646591.697650.

46 Nicola Santoro. *Design and Analysis of Distributed Algorithms (Wiley Series on Parallel and Distributed Computing).* Wiley-Interscience, 2006.

47 Gurdip Singh. Efficient leader election using sense of direction. *Distributed Computing*, 10(3):159–165, 1997. doi:10.1007/s004460050033.

48 Gurdip Singh. Leader election in complete networks. *SIAM J. Comput.*, 26(3):772–785, 1997.

49 Andrew S. Tanenbaum and Maarten Van Steen. *Distributed systems: principles and paradigms.* Prentice-Hall, 2007.

50 Gerard Tel. *Introduction to distributed algorithms.* Cambridge University Press, New York, NY, USA, 1994.

51 Yong Yao and Johannes Gehrke. The cougar approach to in-network query processing in sensor networks. *SIGMOD Record*, 31(3):9–18, 2002.

52 Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D. Kubiatowicz. Tapestry: a resilient global-scale overlay for service deployment. *IEEE J. Selected Areas in Communications*, 22(1):41–53, January 2004. doi:10.1109/JSAC.2003.818784.

Making Byzantine Consensus Live

Manuel Bravo
IMDEA Software Institute, Madrid, Spain

Gregory Chockler
University of Surrey, UK

Alexey Gotsman
IMDEA Software Institute, Madrid, Spain

Abstract

Partially synchronous Byzantine consensus protocols typically structure their execution into a sequence of *views*, each with a designated leader process. The key to guaranteeing liveness in these protocols is to ensure that all correct processes eventually overlap in a view with a correct leader for long enough to reach a decision. We propose a simple *view synchronizer* abstraction that encapsulates the corresponding functionality for Byzantine consensus protocols, thus simplifying their design. We present a formal specification of a view synchronizer and its implementation under partial synchrony, which runs in bounded space despite tolerating message loss during asynchronous periods. We show that our synchronizer specification is strong enough to guarantee liveness for single-shot versions of several well-known Byzantine consensus protocols, including HotStuff, Tendermint, PBFT and SBFT. We furthermore give precise latency bounds for these protocols when using our synchronizer. By factoring out the functionality of view synchronization we are able to specify and analyze the protocols in a uniform framework, which allows comparing them and highlights trade-offs.

2012 ACM Subject Classification Theory of computation → Distributed computing models

Keywords and phrases Byzantine consensus, blockchain, partial synchrony, liveness

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.23

Related Version An extended version is available at `https://arxiv.org/abs/2008.04167`.

Funding This work was partially supported by an ERC Starting Grant RACCOON.

Acknowledgements We want to thank Giuliano Losa, Dahlia Malkhi, Dragos-Adrian Seredinschi, Lacramioara Astefanoaei and Eugen Zalinescu for comments that helped improve the paper.

1 Introduction

The popularity of blockchains has renewed interest in Byzantine consensus protocols, which allow a set of processes to reach an agreement on a value despite a fraction of the processes being malicious. Unlike proof-of-work or proof-of-stake protocols underlying many blockchains, classic Byzantine consensus assumes a fixed set of processes, but can in exchange provide hard guarantees on the finality of decisions. Byzantine consensus protocols are now used in blockchains with both closed membership [9, 31] and open one [15, 16, 30], in the latter case by running Byzantine consensus inside a committee elected among blockchain participants. These use cases have motivated a wave of new algorithms [15, 31, 41] that improve on classical solutions, such as DLS [27] and PBFT [20].

Designing Byzantine consensus protocols is challenging, as witnessed by a number of bugs found in recent protocols [1, 4, 7, 18]. Historically, researchers have paid more attention to safety of these protocols rather than liveness: e.g., while PBFT came with a safety proof [19], the nontrivial mechanism used to guarantee its liveness has never had one. However, achieving liveness of Byzantine consensus is no less challenging than its safety. The seminal FLP result shows that guaranteeing both properties is impossible when the network is asynchronous [28].

© Manuel Bravo, Gregory Chockler, and Alexey Gotsman;
licensed under Creative Commons License CC-BY
34th International Symposium on Distributed Computing (DISC 2020).
Editor: Hagit Attiya; Article No. 23; pp. 23:1–23:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Hence, consensus protocols aim to guarantee safety under all circumstances and liveness only when the network is synchronous. The expected network behavior is formalized by the *partial synchrony* model [27]. In one of its more general formulations [22], the model guarantees that after some unknown *Global Stabilization Time (GST)* the system becomes synchronous, with message delays bounded by an unknown constant δ and process clocks tracking real time. Before GST, however, messages can be lost or arbitrarily delayed, and clocks at different processes can drift apart without bound. This behavior reflects real-world phenomena: in practice, the space for buffering unacknowledged messages in the communication layer is bounded, and messages will be dropped if this space overflows; also, clocks are synchronized by exchanging messages (e.g., using NTP), so network asynchrony will make clocks diverge.

Byzantine consensus protocols usually achieve liveness under partial synchrony by dividing execution into *views* (aka rounds), each with a designated leader process responsible for driving the protocol towards a decision. If a view does not reach a decision (e.g., because its leader is faulty), processes switch to the next one. To ensure liveness, the protocol needs to guarantee that all correct processes will eventually enter the same view with a correct leader and stay there long enough to complete the communication required for a decision. Achieving such *view synchronization* is nontrivial, because before GST, clocks that could measure the duration of a view can diverge, and messages that could be used to bring processes into the same view can get lost or delayed. Thus, by GST processes may end up in wildly different views, and the protocol has to bring them back together, despite any disruption caused by Byzantine processes. Some of the Byzantine consensus protocols integrate the functionality required for view synchronization with the core consensus protocol, which complicates their design [15, 20]. In contrast, both the seminal DLS work on consensus under partial synchrony [27] and some of the more recent work [3, 38, 41] suggest separating the complex functionality required for view synchronization into a distinct component – *view synchronizer*, or simply *synchronizer*. This approach allows designing Byzantine protocols modularly, with mechanisms for ensuring liveness reused among different protocols.

However, to date there has been no rigorous analysis showing which properties of a synchronizer would be sufficient for modern Byzantine consensus protocols. Furthermore, the existing implementations of synchronizer-like abstractions are either expensive or do not handle partial synchrony in its full generality. In particular, DLS [27] implements view synchronization by constructing clocks from program counters of processes. Since these counters drift apart on every step, processes need to frequently synchronize their local clocks. This results in prohibitive communication overheads and makes this solution impractical. Abraham et al. [3] address this inefficiency by assuming hardware clocks with a bounded drift, but only give a solution for a synchronous system. Finally, recent synchronizers by Naor et al. [38] only handle a simplified variant of partial synchrony which disallows clock drift and message loss before GST.

In this paper we make several contributions that address the above limitations:

- We propose a simple and precise specification of a synchronizer abstraction sufficient for single-shot consensus (§3). The specification ensures that from some point on after GST, all correct processes go through the same sequence of views, overlapping for some time in each one of them. It precisely characterizes the duration of the overlap and gives bounds on how quickly correct processes switch between views.
- We propose a synchronizer implementation, called FastSync, and rigorously prove that it satisfies our specification. FastSync handles the general version of the partial synchrony model [27], allowing for an unknown δ and – before GST – unbounded clock drift and message loss (§3.1). Despite the latter, the synchronizer runs in bounded space – a key

feature under Byzantine failures, because the absence of a bound on the required memory opens the system to denial-of-service attacks. Our synchronizer also does not use digital signatures, relying only on authenticated point-to-point links.

- We show that our synchronizer specification is strong enough to guarantee liveness under partial synchrony for single-shot versions of a number of Byzantine consensus protocols. All of these protocols can thus achieve liveness using a single synchronizer – FastSync. In the paper we consider in detail HotStuff [41] (§4.1) and its two-phase version similar to Tendermint [15] (§4.2); in an extended version [14, §B] we also analyze PBFT [20], SBFT [31] and Tendermint itself. The precise guarantees about the timing of view switches provided by our specification are key to handle such a wide range of protocols.
- We provide a precise latency analysis of FastSync, showing that it quickly converges to a synchronized view (§3.2). Building on this analysis, we prove worst-case latency bounds for the above consensus protocols when using FastSync. Our bounds consider both favorable and unfavorable conditions: if the protocol executes during a synchronous period, they determine how quickly all correct processes decide; and if the protocol starts during an asynchronous period, how quickly the processes decide after GST.
- Most of the protocols we consider were originally presented in a form optimized for solving consensus repeatedly. By specializing them to the standard single-shot consensus problem and factoring out the functionality required for view synchronization, we are able to succinctly capture their core ideas in a uniform framework. This allows us to easily compare the protocols and to shed light on trade-offs between them.

2 System Model

We assume a system of $n = 3f + 1$ processes, out of which at most f can be Byzantine, i.e., can behave arbitrarily. In the latter case the process is *faulty*; otherwise it is *correct*. We call a set Q of $2f + 1$ processes a *quorum* and write $\mathsf{quorum}(Q)$ in this case. Processes communicate using authenticated point-to-point links and, when needed, can sign messages using digital signatures. We denote by $\langle m \rangle_i$ a message m signed by process p_i. We sometimes use a cryptographic hash function $\mathsf{hash}()$, which must be collision-resistant: the probability of an adversary producing inputs m and m' such that $\mathsf{hash}(m) = \mathsf{hash}(m')$ is negligible. Processes are equipped with clocks to measure timeouts. We denote the set of time points by Time (ranged over by t) and assume that local message processing takes zero time.

We consider a generalized *partial synchrony* model [22, 27], where after some time GST message delays between correct processes are bounded by a constant δ, and both GST and δ are unknown to the protocol. Before GST messages can get arbitrarily delayed or lost (although for simplicity we assume that self-addressed messages are never lost). Assuming that both GST and δ are unknown to the protocol (as in [22]) reflects the requirements of practical systems, whose designers cannot accurately predict when network problems leading to asynchrony will stop and what the latency will be during the following synchronous period. We also assume that the processes are equipped with hardware clocks that can drift unboundedly from real time before GST, but do not drift thereafter (our results can be trivially adjusted to handle bounded clock drift after GST, but we omit this for conciseness).

3 Synchronizer Specification and Implementation

We now define a *view synchronizer* interface sufficient for single-shot Byzantine consensus, and present its specification and implementation. Let $\mathsf{View} = \{1, 2, \dots\}$ be the set of *views*, ranged over by v; we sometimes use 0 to denote an invalid view. The job of the synchronizer

is to produce notifications `new_view`(v) at each correct process, telling it to *enter* view v. A process can ensure that the synchronizer has started operating by calling a special `start`() function. We assume that each correct process eventually calls `start`().

For a consensus protocol to terminate, its processes need to stay in the same view for long enough to complete the message exchange leading to a decision. Since the message delay δ after GST is unknown to the protocol, we need to increase the view duration until it is long enough. To this end, the synchronizer is parameterized by a function defining this duration – $F : \mathsf{View} \cup \{0\} \to \mathsf{Time}$, which is monotone, satisfies $F(0) = 0$, and increases unboundedly:

$$\forall \theta.\, \exists v.\, \forall v'.\, v' \geq v \implies F(v') > \theta. \tag{1}$$

The properties on the left of Figure 1 define our synchronizer specification (ignore the properties on the right for the time being). The specification strikes a balance between usability and implementability. On one hand, it is sufficient to prove the liveness of a range of consensus protocols (as we show in §4). On the other hand, it can be efficiently implemented under partial synchrony by our FastSync synchronizer (§3.1).

Ideally, a synchronizer should ensure that all correct processes overlap in each view v for a duration determined by $F(v)$. However, achieving this before GST is impossible due to network and clock asynchrony. Therefore, we require a synchronizer to provide nontrivial guarantees only after GST and starting from some view $\mathcal{V}$. To formulate the guarantees we use the following notation. Given a view v that was entered by a correct process p_i, we denote by $E_i(v)$ the time when this happens; we let $E_{\text{first}}(v)$ and $E_{\text{last}}(v)$ denote respectively the earliest and the latest time when some correct process enters v. We let S_{first} and S_{last} be respectively the earliest and the latest time when some correct process calls `start`(), and S_k the earliest time by which k correct processes do so. Thus, a synchronizer must guarantee that views may only increase at a given process (Property 1), and ensure view synchronization starting from some view $\mathcal{V}$, entered after GST (Property 2). Starting from $\mathcal{V}$, correct processes do not skip any views (Property 3), enter each view $v \geq \mathcal{V}$ within at most d of each other (Property 4) and stay there for a determined amount of time: until $F(v)$ after the first process enters v (Property 5). Our FastSync implementation satisfies Property 4 for $d = 2\delta$. Properties 4 and 5 imply a lower bound on the overlap between the time intervals during which all correct processes execute in view v:

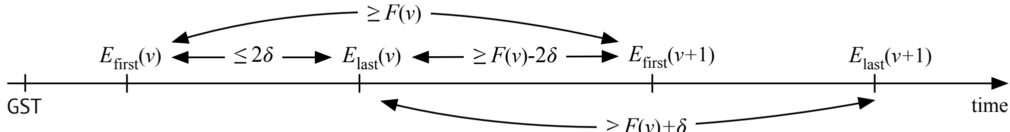
$$\forall v \geq \mathcal{V}.\, E_{\text{first}}(v+1) - E_{\text{last}}(v) \geq (E_{\text{first}}(v) + F(v)) - (E_{\text{first}}(v) + d) = F(v) - d. \tag{2}$$

Due to (1), the overlap increases unboundedly as processes keep switching views. Byzantine consensus protocols are often leader-driven, with leaders rotating round-robin across views. Hence, (2) allows us to prove their liveness by showing that there will eventually be a view with a correct leader (due to Property 3) where all correct processes will overlap for long enough. Having separate Properties 4 and 5 instead of a single property in (2) is required to prove the liveness of some protocols, e.g., two-phase HotStuff (§4.2) and Tendermint (§4.3).

3.1 FastSync: a Bounded-Space Synchronizer for Partial Synchrony

In Figure 2 we present our FastSync synchronizer, which satisfies the synchronizer specification on the left of Figure 1 for $d = 2\delta$. Despite tolerating message loss before GST, FastSync only requires bounded space; it also does not rely on digital signatures.

FastSync measures view duration using a timer timer_view: when the synchronizer tells the process to enter a view v, it sets the timer for the duration $F(v)$. When the timer expires, the synchronizer does not immediately move to the next view v'; instead, it



1. $\forall i, v, v'. (E_i(v)$ and $E_i(v')$ are defined) $\wedge\, v < v' \implies E_i(v) < E_i(v')$
2. $E_{\text{first}}(\mathcal{V}) \geq \mathsf{GST}$
3. $\forall i. \forall v \geq \mathcal{V}. p_i$ is correct $\implies p_i$ enters v
4. $\forall v \geq \mathcal{V}. E_{\text{last}}(v) \leq E_{\text{first}}(v) + d$
5. $\forall v \geq \mathcal{V}. E_{\text{first}}(v+1) \geq E_{\text{first}}(v) + F(v)$

A. $\forall v \geq \mathcal{V}. E_{\text{last}}(v+1) \leq E_{\text{last}}(v) + F(v) + \delta$

B. $S_{\text{first}} \geq \mathsf{GST} \wedge F(1) > 2\delta \implies \mathcal{V} = 1 \wedge E_{\text{last}}(1) \leq S_{\text{last}} + \delta$

C. $F(\mathsf{GV}(\mathsf{GST}+\rho)+1) > 2\delta \wedge S_{f+1} \leq \mathsf{GST} + \rho \implies \mathcal{V} = \mathsf{GV}(\mathsf{GST}+\rho) + 1 \wedge E_{\text{last}}(\mathcal{V}) \leq \mathsf{GST} + \rho + F(\mathcal{V}-1) + 3\delta$

Figure 1 Synchronizer properties (holding for some $\mathcal{V} \in \mathsf{View}$) and their visual illustration. Properties on the left specify the synchronizer abstraction, sufficient to ensure consensus liveness. Properties on the right give latency bounds specific to our FastSync synchronizer (§3.1). The latter satisfies Property 4 for $d = 2\delta$. The parameter ρ is the retransmission interval used by FastSync.

disseminates a special WISH(v') message, announcing its intention. Each process maintains an array max_views : $\{1, \ldots, n\} \to \mathsf{View} \cup \{0\}$, whose j-th entry stores the maximal view received in a WISH message from process p_j (initially 0, updated in line 13). Keeping track of only the maximal views allows the synchronizer to run in bounded space. The process also maintains two variables, view and view$^+$, derived from max_views (initially 0, updated in lines 14 and 15): view$^+$ (respectively, view) is equal to the maximal view such that at least $f+1$ processes (respectively, $2f+1$ processes) wish to switch to a view no lower than this. The two variables monotonically increase and we always have view $\leq$ view$^+$.

The process enters the view determined by the view variable (line 19) when the latter increases (view $>$ *prev_v* in line 16; we explain the extra condition later). At this point the process also resets its timer_view (line 18). Thus, a process enters a view only if it receives a quorum of WISHes for this view or higher, and a process may be forced to switch views even if its timer_view has not yet expired. The latter helps lagging processes to catch up, but poses another challenge. Byzantine processes may equivocate, sending WISH messages to some processes but not others. In particular, they may send WISHes for views $\geq v$ to some correct process, helping it to form a quorum of WISHes sufficient for entering v. But they may withhold the same WISHes from another correct process, so that it fails to form a quorum for entering v, as necessary, e.g., for Property 4. To deal with this, when a process receives a WISH that makes its view$^+$ increase, the process sends WISH(view$^+$) (line 21). By the definition of view$^+$, at least one correct process has wished to move to a view no lower than view$^+$. The WISH(view$^+$) message replaces those that may have been omitted by Byzantine processes and helps all correct processes to quickly form the necessary quorums of WISHes.

An additional guard on entering a view is view$^+$ = view in line 16, which ensures that a process does not enter a "stale" view such that another correct process already wishes to enter a higher one. Similarly, when the timer of the current view expires (line 4), the process sends a WISH for the maximum of view + 1 and view$^+$. In other words, if view = view$^+$, so that the values of the two variables have not changed since the process entered the current view, then the process sends a WISH for the the next view (view + 1). Otherwise, view $<$ view$^+$, and the process sends a WISH for the higher view view$^+$.

```
1  function start()
2    if view⁺ = 0 then
3      send WISH(1) to all;

4  when timer_view expires
5    send WISH(max(view + 1, view⁺))
       to all;

6  periodically
7    if timer_view is enabled then
8      send WISH(view⁺) to all;
9    else if max_views[i] > 0 then
10     send WISH(max(view + 1, view⁺))
         to all;

11 when received WISH(v) from p_j
12   prev_v, prev_v⁺ ← view, view⁺;
13   if v > max_views[j] then max_views[j] ← v;
14   view  ← max{v | ∃k. max_views[k] = v ∧
                |{j | max_views[j] ≥ v}| ≥ 2f + 1};
15   view⁺ ← max{v | ∃k. max_views[k] = v ∧
                |{j | max_views[j] ≥ v}| ≥ f + 1};
16   if view⁺ = view ∧ view > prev_v then
17     stop_timer(timer_view);
18     start_timer(timer_view, F(view));
19     trigger new_view(view);
20   if view⁺ > prev_v⁺ then
21     send WISH(view⁺) to all;
```

Figure 2 The FastSync synchronizer. The periodic handler is invoked every ρ units of time.

To deal with message loss before GST, a process retransmits the highest WISH it sent every ρ units of time, according to its local clock (line 6). Depending on whether timer_view is enabled, the WISH is computed as in lines 21 or 5. Finally, the **start** function ensures that the synchronizer has started operating at the process by sending WISH(1), unless the process has already done so in line 21 due to receiving $f+1$ WISHes from other processes.

Discussion. FastSync requires only $O(n)$ variables for storing views. When proving its correctness, we establish that every view is entered by some correct process [14, §A, Lemma 18], and eventually, correct processes do not skip views (Property 3). These two properties limit the power of the adversary to exhaust the value space for views, similarly to [11].

The basic mechanisms we use in our synchronizer – entering views supported by $2f+1$ WISHes and relaying views supported by $f+1$ WISHes – are similar to the ones used in Bracha's algorithm for reliable Byzantine broadcast [13]. However, Bracha's algorithm only makes a step upon receiving a set of *identical* messages. Thus, its naive application to view synchronization [38, §A.2] requires unbounded space to store the views v for which the number of received copies of WISH(v) still falls below the threshold required for delivery or relay. Moreover, tolerating message loss would require a process to retain a copy of every message it has broadcast, to enable retransmissions. FastSync can be viewed as specializing the mechanisms of Bracha broadcast to take advantage of the particular semantics of WISH messages, by keeping track of only the highest WISH received from each process and by acting on sets of WISHes for non-identical views. This allows tolerating message loss before GST in bounded space and without compromising liveness, as illustrated by the following example.

We first show that, before GST, we may end up in the situation where processes are split as follows: a set P_1 of f correct processes entered v_1, a set P_2 of f correct processes entered $v_2 > v_1$, a correct process p_i entered v_2+1, and f processes are faulty. To reach this state, assume that all correct processes manage to enter view v_1 and then all messages between P_1 and $P_2 \cup \{p_i\}$ start getting lost. The f faulty processes help the processes in $P_2 \cup \{p_i\}$ to enter all views between v_1 and v_2, by providing the required WISHes (line 16), while the processes in P_1 get stuck in v_1. After the processes in $P_2 \cup \{p_i\}$ time out on v_2, they start sending WISH(v_2+1) (line 5), but all messages directed to processes other than p_i get lost, so that the processes in P_2 get stuck in v_2. The faulty processes then help p_i gather $2f+1$ messages WISH(v_2+1) and enter v_2+1 (line 16).

Assume now that GST occurs, the faulty processes go silent and the correct processes time out on the views they are in. Thus, the f processes in P_1 send WISH(v_1+1), the f

processes in P_2 send $\mathtt{WISH}(v_2+1)$, and p_i sends $\mathtt{WISH}(v_2+2)$ (line 10). The processes in P_1 eventually receive the WISHes from $P_2 \cup \{p_i\}$, so that they set $\mathsf{view}^+ = v_2+1$ and send $\mathtt{WISH}(v_2+1)$ (line 21). Note that here processes act on $f+1$ mismatching WISHes, unlike in Bracha broadcast. Eventually, the processes in $P_1 \cup P_2$ receive $2f$ copies of $\mathtt{WISH}(v_2+1)$ and one $\mathtt{WISH}(v_2+2)$, which causes them to set $\mathsf{view} = v_2+1$ and enter v_2+1 (line 16). Note that here processes act on $2f+1$ mismatching WISHes, again unlike in Bracha broadcast. Finally, the processes $P_1 \cup P_2$ time out and send $\mathtt{WISH}(v_2+2)$ (line 5), which allows all correct processes to enter v_2+2. Acting on sets of mismatching WISHes is crucial for liveness in this example: if processes only accepted matching sets, like in Bracha broadcast, message loss before GST would cause them to get stuck, and they would never converge to the same view.

3.2 Correctness and Latency Bounds of FastSync

As we demonstrate shortly, the synchronizer specification given by Properties 1-5 in Figure 1 serves to prove that consensus *eventually* reaches a decision. However, FastSync also satisfies some additional properties that allow us to quantify *how quickly* this happens under both favorable and unfavorable conditions. We list these properties on the right of Figure 1.

▶ **Theorem 1.** FastSync *satisfies all properties in Figure 1 for* $d = 2\delta$.

Due to space constraints, we defer the proof to [14, §A]. Property A allows us to quantify the cost of switching between several views (e.g., due to faulty leaders). This is formalized by the following proposition, easily proved using Property A by induction on v'.

▶ **Proposition 2.** $\forall v, v'. \mathcal{V} \le v \le v' \implies E_{\text{last}}(v') \le E_{\text{last}}(v) + \sum_{k=v}^{v'-1}(F(k)+\delta)$.

Property B guarantees that, when the synchronizer starts after GST ($S_{\text{first}} \ge \mathsf{GST}$) and the initial timeout is long enough ($F(1) > 2\delta$), processes synchronize in the very first view ($\mathcal{V} = 1$) and enter it within δ of the last correct process calling $\mathtt{start}()$.

Let the *global view* at time t, denoted $\mathsf{GV}(t)$, be the maximum view entered by a correct process at or before t, or 0 if no view was entered by a correct process. Property C quantifies the latency of view synchronization in a more general case when the synchronizer may be started before GST. The property depends on the interval ρ at which the synchronizer periodically retransmits its internal messages to deal with possible message loss. The property considers the highest view $\mathsf{GV}(\mathsf{GST}+\rho)$ a correct process has at time $\mathsf{GST}+\rho$ and ensures that all correct processes synchronize in the immediately following view within at most $\rho + F(\mathcal{V}-1) + 3\delta$ after GST. This is guaranteed under an assumption that the timeout of this view exceeds 2δ and $f+1$ correct processes call $\mathtt{start}()$ early enough. Since GST can be arbitrary, in principle, so can be the view $\mathcal{V}$ and, thus due to (1), the timeout $F(\mathcal{V}-1)$. However, practical implementations usually stop increasing timeouts when they exceed a reasonable value. Hence, Property C guarantees that to reach $\mathcal{V}$, processes need to wait for at most a single maximal timeout.

4 Liveness and Latency of Byzantine Consensus Protocols

We show that our synchronizer abstraction allows ensuring liveness and establishing latency bounds for several consensus protocols. The protocols solve a variant of Byzantine consensus problem that relies on an application-specific $\mathsf{valid}()$ predicate to indicate whether a value is valid [17, 24]. In the context of blockchain systems a value represents a block, which may be

invalid if it does not include correct signatures authorizing its transactions. Assuming that each correct process proposes a valid value, each of them has to decide on a value so that:

- **Agreement.** No two correct processes decide on different values.
- **Validity.** A correct process decides on a valid value, i.e., satisfying valid().
- **Termination.** Every correct process eventually decides on a value.

4.1 Single-Shot HotStuff

We first consider the HotStuff protocol [41], underlying the upcoming Libra cryptocurrency [2]. The protocol was originally presented as solving an inherently multi-shot problem, agreeing on a hash-chain of blocks. In Figure 3 we present its single-shot version that concisely expresses the key idea and allows comparing the protocol with others. For brevity, we eschew the use of threshold signatures, which makes the communication complexity of a leader change $O(n^2)$ rather than $O(n)$, like in the original HotStuff. This complexity is still better than that of PBFT, which is $O(n^3)$. We handle linear versions of the protocols we consider in [14, §C]. HotStuff delegated view synchronization to a separate component [41], but did not provide its practical implementation or analyze how view synchronization affects the protocol latency. We show that our single-shot version of HotStuff is live when used with a synchronizer satisfying the specification in §3 and give precise bounds on its latency. We also show that the protocol requires only bounded space when using our synchronizer FastSync.

The protocol in Figure 3 works in a succession of views produced by the synchronizer. Each view v has a fixed leader $\mathsf{leader}(v) = p_{((v-1) \bmod n)+1}$ that is responsible for proposing a value to the other processes, which vote on the proposal. A correct leader needs to choose its proposal carefully so that, if a value was decided in a previous view, the leader will propose the same value. To enable the leader to do this, when a process receives a notification to move to a view v (line 1), it sends a NEWLEADER message to the leader of v with information about the latest value it accepted in a previous view (as described in the following). The process also stores the view v in a variable curr_view, and sets a flag voted to FALSE, to record that it has not yet received any proposal from the leader in the current view. The leader computes its proposal (as described in the following) based on a quorum of NEWLEADER messages (line 5) and sends the proposal, along with some supporting information, in a PROPOSE message to all processes (for uniformity, including itself).

The leader's proposal is processed in three phases. A process receiving a proposal x from the leader of its view v (line 11) first checks that voted is FALSE, so that it has not yet accepted a proposal in v. It also checks that x satisfies a SafeProposal predicate (explained later), which ensures that a faulty leader cannot reverse decisions reached in previous views. The process then sets voted to TRUE and stores x in curr_val.

Since a faulty leader may send different proposals to different processes, the process next communicates with others to check that they received the same proposal. To this end, the process disseminates a PREPARED message with the hash of the proposal it received. The process then waits until it gathers a set C of PREPARED messages from a quorum with a hash matching the proposal (line 16); we call this set of messages a *prepared certificate* for the value and check it using the prepared predicate. The process stores the proposal in prepared_val, the view in which it formed the prepared certificate in prepared_view, and the certificate itself in cert. At this point we say that the process *prepared* the value. Since a certificate consists of at least $2f + 1$ PREPARED messages and there are $3f + 1$ replicas in total, it is impossible to prepare different values in the same view: this would require some correct process to send two PREPARED messages with different values in the same view, which is impossible due to

```
1  upon new_view(v)
2  │ curr_view ← v;
3  │ voted ← FALSE;
4  │ send ⟨NEWLEADER(curr_view, prepared_view,
   │     prepared_val, cert)⟩_i to leader(curr_view);

5  when received {⟨NEWLEADER(v, view_j, val_j,
       cert_j)⟩_j | p_j ∈ Q} = M for a quorum Q
6  │ pre: curr_view = v ∧ p_i = leader(v) ∧
   │        (∀m ∈ M. ValidNewLeader(m));
7  │ if ∃j. view_j = max{view_k | p_k ∈ Q} ≠ 0 then
8  │ │ send ⟨PROPOSE(v, val_j, cert_j)⟩_i to all;
9  │ else
10 │ │ send ⟨PROPOSE(v, myval(), ⊥)⟩_i to all;

11 when received ⟨PROPOSE(v, x, _)⟩_j = m
12 │ pre: curr_view = v ∧ voted = FALSE ∧
   │        SafeProposal(m);
13 │ curr_val ← x;
14 │ voted ← TRUE;
15 │ send ⟨PREPARED(v, hash(curr_val))⟩_i to all;

16 when received {⟨PREPARED(v, h)⟩_j |
       p_j ∈ Q} = C for a quorum Q
17 │ pre: curr_view = v ∧ voted = TRUE
   │        ∧ hash(curr_val) = h;
18 │ prepared_val ← curr_val;
19 │ prepared_view ← curr_view;
20 │ cert ← C;
21 │ send ⟨PRECOMMITTED(v, h)⟩_i to all;

22 when received {⟨PRECOMMITTED(v, h)⟩_j
       | p_j ∈ Q} for a quorum Q
23 │ pre: curr_view = prepared_view = v ∧
   │        hash(curr_val) = h;
24 │ locked_view ← prepared_view;
25 │ send ⟨COMMITTED(v, h)⟩_i to all;

26 when received {⟨COMMITTED(v, h)⟩_j |
       p_j ∈ Q} for a quorum Q
27 │ pre: curr_view = locked_view = v ∧
   │        hash(curr_val) = h;
28 │ decide(curr_val);
```

$$\mathsf{prepared}(C, v, h) \iff \exists Q.\, \mathsf{quorum}(Q) \wedge C = \{\langle \mathtt{PREPARED}(v, h)\rangle_j \mid p_j \in Q\}$$

$$\mathsf{ValidNewLeader}(\langle \mathtt{NEWLEADER}(v', v, x, C)\rangle_{_}) \iff v < v' \wedge (v \neq 0 \implies \mathsf{prepared}(C, v, \mathsf{hash}(x)))$$

$$\mathsf{SafeProposal}(\langle \mathtt{PROPOSE}(v, x, C)\rangle_i) \iff p_i = \mathsf{leader}(v) \wedge \mathsf{valid}(x) \wedge$$
$$(\mathsf{locked_view} \neq 0 \implies x = \mathsf{prepared_val} \vee (\exists v'.\, v > v' > \mathsf{locked_view} \wedge \mathsf{prepared}(C, v', \mathsf{hash}(x))))$$

Figure 3 Single-shot HotStuff. All variables storing views are initially set to 0 and others to $\perp$.

the check on the voted flag in line 12. Formally, let us write $\mathsf{wf}(C)$ (for *well-formed*) if the set of correctly signed messages C have been sent in the execution of the protocol.

▶ **Proposition 3.** $\forall v, C, C', x, x'.\, \mathsf{prepared}(C, v, \mathsf{hash}(x)) \wedge \mathsf{prepared}(C', v, \mathsf{hash}(x')) \wedge \mathsf{wf}(C) \wedge \mathsf{wf}(C') \implies x = x'$.

Preparing a value is a prerequisite for deciding on it. Hence, by Proposition 3 a prepared certificate for a value x and a view v guarantees that x is the only value that can be possibly decided in v. For this reason, it is this certificate, together with the corresponding value and view, that the process sends upon a view change to the new leader in a NEWLEADER message (line 4). The leader makes its proposal based on a quorum of NEWLEADER messages with prepared certificates formed in lower views than the one it is in (line 5), as checked by ValidNewLeader. Similarly to Paxos [34] and PBFT [20], the leader selects as its proposal the value prepared in the highest view, or, if there are no such values, its own proposal given by myval(). In the former case, the leader sends the corresponding certificate in its PROPOSE message, to justify its choice; in the latter case this is replaced by $\perp$.

Once a process prepares a value x, it participates in the next message exchange: it disseminates a PRECOMMITTED message with the hash of the value and waits until it gathers a quorum of PRECOMMITTED messages matching the prepared value (line 22). This ensures that at least $f + 1$ correct processes have prepared the value x. Since the leader of the next view will gather prepared commands from at least $2f + 1$ processes, at least one correct process will tell the leader about the value x, and thus the leader will be aware of this value as a potential decision in the current view.

Having gathered a quorum of PRECOMMITTED messages for a value, the process becomes *locked* on this value, which is recorded by setting a special variable locked_view to the current view. From this point on, the process will not accept a proposal of a different value from a leader of a future view, unless the leader can convince the process that no decision was reached in the current view. This is ensured by the SafeProposal check the process does on a PROPOSE message from a leader (line 12). This checks that the value is valid and that, if the process has previously locked on a value, then either the leader proposes the same value, or its proposal is justified by a prepared certificate from a higher view than the lock. In the latter case the process can be sure that no decision was reached in the view it is locked on.

Having locked a value, the process participates in the final message exchange: it disseminates a COMMITTED message with the hash of the value and waits until it gathers a quorum of matching COMMITTED messages for the locked value (line 26). Once this happens, the process decides on this value. Gathering a quorum of COMMITTED messages on a value x ensures that at least $f+1$ correct processes are locked on the same value. This guarantees that a leader in a future view cannot get processes to decide on a different value: this would require $2f+1$ processes to accept the leader's proposal; but at least one correct process out of these would be locked on x and would refuse to accept a different value due to the SafeProposal check. Thus, while the exchange of PRECOMMITTED messages ensures that a future correct leader will be aware of the value being decided and will be able to make a proposal passing SafeProposal checks (liveness), the exchange of COMMITTED ensures that a faulty leader cannot revert the decision (safety).

Since processes transition through increasing views (Property 1 in Figure 1), we get

▶ **Proposition 4.** *The variables* locked_view, prepared_view *and* curr_view *at a correct process never decrease and we always have* locked_view $\leq$ prepared_view $\leq$ curr_view.

Note that, when a process enters view 1, it trivially knows that no decision could have been reached in prior views. Hence, the leader of view 1 can send its proposal immediately, without waiting to receive a quorum of NEWLEADER messages (line 10), and processes can avoid sending these messages to this leader. For brevity, we omit this optimization from the pseudocode, even though we take it into account in our latency analysis.

Since the synchronizer is not guaranteed to switch processes between views all at the same time, a process in a view v may receive a message from a higher view $v' > v$, which needs to be stored in case the process finally switches to v'. If implemented naively, this would require a process to store unboundedly many messages. Instead, we allow a process to store, for each message type and sender, only the message of this type received from this sender that has the highest view. As we show below (Theorem 5), this does not violate liveness. Thus, assuming consensus proposals of bounded size, the protocol Figure 3 runs in bounded space, and so does the overall consensus protocol with the FastSync synchronizer.

We defer the proof that the protocol satisfies Validity and Agreement to [14, §B.1] and focus on our core contribution: proving its liveness and analyzing its latency.

Protocol liveness. Assume that the protocol is used with a synchronizer satisfying Properties 1-5 on the left of Figure 1; to simplify the following latency analysis, we assume $d = 2\delta$, as for FastSync. The next theorem states requirements on a view sufficient for the protocol to reach a decision and quantifies the resulting latency.

▶ **Theorem 5.** *Let $v \geq \mathcal{V}$ be a view such that $F(v) > 7\delta$ and* leader(v) *is correct. Then in single-shot HotStuff all correct processes decide in view v by $E_{\mathrm{last}}(v) + 5\delta$.*

Proof. By Property 2 we have $E_{\text{first}}(v) \geq \mathsf{GST}$, so that all messages sent by correct processes after $E_{\text{first}}(v)$ get delivered to all correct processes within δ. Once a correct process enters v, it sends its `NEWLEADER` message, so that $\mathsf{leader}(v)$ will receive a quorum of such messages by $E_{\text{last}}(v)+\delta$. When this happens, the leader will send its proposal in a `PROPOSE` message, which correct processes will receive by $E_{\text{last}}(v) + 2\delta$. If they deem the proposal safe, it takes them at most 3δ to exchange the sequence of `PREPARED`, `PRECOMMITTED` and `COMMITTED` messages. By (2), all correct processes will stay in v until at least $E_{\text{last}}(v) + (F(v) - d) > E_{\text{last}}(v) + 5\delta$, and thus will not send a message with a view $> v$ until this time. Thus, none of none of the above messages will be discarded at correct processes before this time, and assuming the safety checks pass, the sequence of message exchanges will lead to decisions by $E_{\text{last}}(v) + 5\delta$.

It remains to show that the proposal $\mathsf{leader}(v)$ makes in view v (line 5) will satisfy SafeProposal at all correct processes (line 11). It is easy to show that the proposal satisfies valid, so we now need to prove the last conjunct of SafeProposal. This trivially holds if no correct process is locked on a value when receiving the `PROPOSE` message from the leader.

We now consider the case when some correct process is locked on a value when receiving the `PROPOSE` message, and let p_i be a process that is locked on the highest view among correct processes. Let $x = p_i.\mathsf{prepared_val}$ be the value locked and $v_0 = p_i.\mathsf{locked_view} < v$ be the corresponding view. Since p_i locked x at v_0, it must have previously received messages $\texttt{PRECOMMITTED}(v_0, \mathsf{hash}(x))$ from a quorum of processes (line 22), at least $f + 1$ of which have to be correct. The latter processes must have prepared the value x at view v_0 (line 16). By Proposition 4, when each of these $f + 1$ correct processes enters view v, it has $\mathsf{prepared_view} \geq v_0$ and thus sends the corresponding value and its prepared certificate in the $\texttt{NEWLEADER}(v, \dots)$ message to $\mathsf{leader}(v)$. The leader is guaranteed to receive at least one of these messages before making a proposal, since it only does this after receiving at least $2f + 1$ `NEWLEADER` messages (line 5). Hence, the leader proposes a value x' with a prepared certificate formed at some view $v' \geq v_0$ no lower than any view that a correct process is locked on when receiving the leader's proposal. Furthermore, if $v' = v_0$, then by Proposition 3 we have that $x' = x$ and x is the only value that can be locked by a correct process at v_0. Hence, the leader's proposal will satisfy SafeProposal at each correct process. ◀

Since by Property 3 correct processes enter every view starting from $\mathcal{V}$ and, by the definition of $\mathsf{leader}()$, leaders rotate round-robin, we are always guaranteed to encounter a correct leader after at most f view changes. Then Theorem 5 implies that the protocol is live when using a timeout function F that grows without bound.

▶ **Corollary 6.** *Let F be such that (1) holds. Then in single-shot HotStuff all correct processes eventually decide.*

Protocol latency. When single-shot HotStuff is used with the FastSync synchronizer, rather than an arbitrary one, we can use Properties A-C on the right of Figure 1 to bound how quickly the protocol reaches a decision after GST. To this end, we combine Theorem 5 with Property C, which bounds the latency of view synchronization, and Proposition 2, which bounds the latency of going through up to f views with faulty leaders.

▶ **Corollary 7.** *Let $v = \mathsf{GV}(\mathsf{GST} + \rho) + 1$ and assume that $F(v) > 7\delta$ and $S_{f+1} \leq \mathsf{GST} + \rho$. Then in single-shot HotStuff all correct processes decide by $\mathsf{GST} + \rho + \sum_{k=v-1}^{v+f-1}(F(k) + \delta) + 7\delta$.*

We can also quantify the latency of the protocol under favorable conditions, when it is started after GST. In this we rely on Property B, which gives conditions under which processes synchronize in view 1. The following corollary of Theorem 5 exploits this property to bound

the latency of HotStuff when it is started after GST and the initial timeout is set appropriately, but the protocol may still go through a sequence of up to f faulty leaders. The summation in the bound (coming from Proposition 2) quantifies the overhead in the latter case.

▶ **Corollary 8.** *Assume that $S_{\text{first}} \geq \mathsf{GST}$ and $F(1) > 7\delta$. Then in single-shot HotStuff all correct processes decide no later than $S_{\text{last}} + \sum_{k=1}^{f}(F(k) + \delta) + 6\delta$.*

Finally, the next corollary bounds the latency when additionally the leader of view 1 is correct, in which case the protocol can benefit from the optimized execution of this view noted earlier. The corollary follows from Property B and an easy strengthening of Theorem 5 for the special case of $v = \mathcal{V} = 1$.

▶ **Corollary 9.** *Assume that $S_{\text{first}} \geq \mathsf{GST}$, $F(1) > 6\delta$, and $\mathsf{leader}(1)$ is correct. Then in single-shot HotStuff all correct processes decide no later than $S_{\text{last}} + 5\delta$.*

4.2 Two-Phase HotStuff

We next consider a *two-phase* variant of HotStuff [41], which processes the leader's proposals in two phases instead of three. In exchange, it uses timeouts not just for view synchronization, but also in the core consensus protocol to delimit different stages of a single view. This demonstrates that our synchronizer specification is strong enough to deal with interactions between the timeouts in different parts of the overall protocol. When used with our FastSync synchronizer, the protocol furthermore requires only bounded space. Two-phase HotStuff is similar to Tendermint [15] and Casper [16], which use timeouts for the same purposes. We chose this protocol for conciseness of presentation, but in [14, §B.5] we also present a variant of the original Tendermint consensus based on our synchronizer (see §4.3).

Due to space constraints, we describe the changes to the protocol in Figure 3 required to get its two-phase version informally and defer the pseudocode to [14, §B.2]. In two-phase HotStuff, a process handles a proposal from the leader in the same way as in the three-phase one, by sending a `PREPARED` message (line 11 in Figure 3). Upon assembling a quorum of matching `PREPARED` messages (line 16), the process updates its variables as per lines 18-20, but in addition immediately becomes locked on the prepared value prepared_val, without exchanging `PRECOMMITTED` messages: the process assigns locked_view to the current view and sends a `COMMITTED` message with the hash of the value. As before, assembling a quorum of such messages causes the process to decide on the value (line 26). Upon entering a new view (line 1), a process sends to the leader a `NEWLEADER` message with the information about the last value it prepared (and therefore locked, line 4). The leader chooses its proposal in the same way as in three-phase HotStuff (line 5).

The two-phase version of HotStuff is safe for the same reasons as the three-phase one: the exchange of `PRECOMMITTED` messages, omitted from the current protocol, is only needed for liveness, not safety. However, ensuring liveness in two-phase HotStuff requires a different mechanism: since a correct process p_i gets locked on a value immediately after preparing it, gathering prepared values from an arbitrary quorum of processes is not enough for the leader to ensure it will make a proposal that will pass the SafeProposal check at p_i: the quorum may well exclude this process. To solve this problem, the leader waits before making a proposal so that eventually in some view it will receive `NEWLEADER` messages from *all correct processes*. This ensures the leader will eventually make a proposal that will pass the SafeProposal checks at all of them. In more detail, when a process enters a view where it is the leader, it sets a special timer timer_newleader for the duration determined by a function F_p. The leader makes a proposal by executing the handler in line 5 only after the timer expires.

For the leader to make an acceptable proposal, the duration of timer_newleader needs to be long enough for all NEWLEADER messages for this view from correct processes to reach the leader. For the protocol to decide, after timer_newleader expires, processes also need to stay in the view long enough to complete the necessary message exchanges. The following theorem characterizes these requirements formally, again assuming $d = 2\delta$ in Property 4. Note that in the proof of the theorem we rely on the guarantees about the timing of correct processes entering a view (Property 4) to show that timer_newleader fulfills its intended function.

▶ **Theorem 10.** *Let $v \geq \mathcal{V}$ be a view such that $F_p(v) > 3\delta$, $F(v) - F_p(v) > 5\delta$ and* leader(v) *is correct. Then in two-phase HotStuff all correct processes decide at v by $E_{\text{last}}(v) + F_p(v) + 3\delta$.*

Proof. Once a correct process enters v, it sends its NEWLEADER message, so that leader(v) is guaranteed to receive such messages from all correct processes by $E_{\text{last}}(v) + \delta$. By Property 4, the leader enters v by $E_{\text{last}}(v) - 2\delta$ at the earliest. Since the leader starts its timer_newleader when it enters v and $F_p(v) > 3\delta$, timer_newleader can only expire after $E_{\text{last}}(v) + \delta$. Thus, the leader is guaranteed to receive NEWLEADER messages from all correct processes before timer_newleader expires. When timer_newleader expires, which happens no later than $E_{\text{last}}(v) + F_p(v)$, the leader will send its proposal in a PROPOSE message, which correct processes will receive by $E_{\text{last}}(v) + F_p(v) + \delta$. If they deem the proposal safe, it takes them at most 2δ to exchange the sequence of PREPARED and COMMITTED messages leading to decisions. By (2), all correct processes will stay in v until at least $E_{\text{last}}(v) + (F(v) - d) > E_{\text{first}}(v) + F_p(v) + 3\delta$. By then the above sequence of message exchanges will complete, and all correct processes will decide.

It remains to show that the proposal leader(v) makes in view v will satisfy SafeProposal at all correct processes. It is easy to show that this proposal is valid, so we now need to prove the last conjunct of SafeProposal. This trivially holds if no correct process is locked on a value when receiving the PROPOSE message from the leader. We now consider the case when some correct process is locked on a value when receiving the PROPOSE message, and let p_i be a process that is locked on the highest view among correct processes. Let $x = p_i$.prepared_val be the value locked and $v_0 = p_i$.locked_view $< v$ be the corresponding view. Since leader(v) receives all of the NEWLEADER messages sent by correct processes before making its proposal, it proposes a value x' with a prepared certificate formed at some view $v' \geq v_0$. Also, if $v' = v_0$, then by Proposition 3, $x' = x$ and x is the only value that can be locked by a correct process at v_0. Hence, the leader's proposal will satisfy SafeProposal at each correct process. ◀

Since leaders rotate round-robin, Theorem 10 implies that the protocol is live, provided the functions F and F_p, *as well as the difference between them*, grow without bound. This can be satisfied, e.g., by letting $F(v) = 2v$ and $F_p(v) = v$.

▶ **Corollary 11.** *Let F and F_p be such that (1) holds and $\forall \theta. \exists v. \forall v'. v' \geq v \implies F(v') - F_p(v') > \theta$. Then in two-phase HotStuff all correct processes eventually decide.*

Protocol latency. Similarly to §4.1, when the protocol is used with the FastSync synchronizer, we can quantify its latency in both unfavorable scenarios (when starting before GST) and favorable scenarios (when starting after GST). The first corollary of Theorem 10 below uses Property C and Proposition 2, and the following two corollaries, Property B.

▶ **Corollary 12.** *Let $v = \mathsf{GV}(\mathsf{GST} + \rho) + 1$ and assume that $S_{f+1} \leq \mathsf{GST} + \rho$, $F_p(v) > 3\delta$ and $F(v) - F_p(v) > 5\delta$. Then in two-phase HotStuff all correct processes decide no later than $\mathsf{GST} + \rho + \sum_{k=v-1}^{v+f-1}(F(k) + \delta) + F_p(v + f) + 5\delta$.*

▶ **Corollary 13.** *Assume that* $S_{\mathrm{first}} \geq \mathsf{GST}$*,* $F_p(1) > 3\delta$ *and* $F(1) - F_p(1) > 5\delta$*. Then in two-phase HotStuff all correct processes decide no later than* $S_{\mathrm{last}} + \sum_{k=1}^{f}(F(k)+\delta) + F_p(f+1) + 4\delta$*.*

▶ **Corollary 14.** *Assume that* $S_{\mathrm{first}} \geq \mathsf{GST}$*,* $F(1) > 5\delta$ *and* $\mathsf{leader}(1)$ *is correct. Then in two-phase HotStuff all correct processes decide no later than* $S_{\mathrm{last}} + 4\delta$*.*

Like in §4.1, the last corollary takes into account the optimized execution of view 1. The above latency bounds allow us to compare the two-phase and three-phase versions of HotStuff (§4.1). In the ideal case when the timeouts are set optimally and the leader of view 1 is correct, two-phase HotStuff has a lower latency than three-phase one: 4δ in Corollary 14 vs 5δ in Corollary 9. When the initial leader is faulty, both protocols incur the overhead of switching through several views until they encounter a correct leader (Corollaries 13 and 8). In this case, the latency of deciding in the first view with a correct leader is at most 6δ for three-phase HotStuff and $F_p(f+1)+4\delta$ for two-phase one. Even when $F_p(f+1)$ is the optimal 3δ, the two-phase HotStuff bound yields 7δ – a higher latency than for three-phase HotStuff. The latency bounds for the case of starting before GST relate similarly (Corollaries 12 and 7). The higher latency of two-phase HotStuff in these cases are caused by the inclusion of the timeout $F_p(f+1)$, which reflects the lack of "optimistic responsiveness" of this protocol [41].

4.3 Single-Shot PBFT, SBFT and Tendermint

Using our synchronizer specification, we have also proved the correctness and analyzed the latency of single-shot versions of PBFT [20], SBFT [31] and Tendermint [15], thus demonstrating the wide applicability of the specification. Due to space constraints we defer the details to [14, §B]. Our analysis of PBFT is similar to that of HotStuff. SBFT is a recent improvement of PBFT that adds a fast path for cases when all processes are correct, and our analysis quantifies the latency of both paths.

Tendermint is similar to two-phase HotStuff; in particular, it also uses timeouts both for view synchronization and to delimit different stages of a single view. However, the protocol never sends messages with certificates, and thus, like FastSync, does not need digital signatures. Tendermint integrates the functionality required for view synchronization with the core consensus protocol, breaking its control flow in multiple places. We consider its variant that delegates this functionality to the synchronizer, thus simplifying the protocol. Our analysis of the resulting protocol is similar to the one of two-phase HotStuff in §4.2. Apart from deriving latency bounds for the protocol, our analysis exploits the synchronizer specification to give a proof of its liveness that is more rigorous than the existing ones [8, 15], which lacked a detailed correctness argument for the view synchronization mechanism used in the protocol.

5 Related Work

Most Byzantine consensus protocols are based on the concept of views (aka rounds), and thus include a mechanism for view synchronization. This mechanism is typically integrated with the core consensus protocol, which complicates the design [15, 20, 31]. Subtle view synchronization mechanisms have often come without a proof of liveness (e.g., PBFT [19]) or had liveness bugs (e.g., Tendermint [7] and Casper [1]). Furthermore, liveness proofs have not usually given concrete bounds on the latency of reaching a decision (exceptions are [6, 36]).

Several papers suggested separating the functionality of view synchronization into a distinct component, starting with the seminal DLS paper on consensus under partial synchrony [27]. DLS specified the guarantees provided by view synchronization indirectly, by

proving that its implementation simulated an abstract computational model with a built-in notion of rounds. Unlike us, DLS did not give a specification determining how long processes stay in a round and how quickly they switch between rounds; as we have demonstrated, such properties are needed to reason about modern Byzantine consensus protocols. DLS implemented rounds using a distributed protocol that synchronizes process-local clocks obtained by counting state transitions of each process. This protocol has to synchronize local clocks on every step of the consensus algorithm, which results in prohibitive communication overheads and makes this solution impractical.

Abraham et al. [3] build upon ideas from fault-tolerant clock synchronization [25, 40] to implement view synchronization assuming that processes have access to hardware clocks with bounded drift. But this work only gives a solution for a synchronous system. Our FastSync synchronizer also assumes hardware clocks but removes the assumption of bounded drift before GST, thus making them compatible with partial synchrony. We note that, although the problems of clock and view synchronization are different, they are closely related at the algorithmic level. We therefore believe that our view synchronization techniques can in the future be adapted to obtain an efficient partially synchronous clock synchronization protocol.

The HotStuff protocol [41] delegated the functionality of view synchronization to a separate component, called a pacemaker. But it did not provide a formal specification of this component or a practical implementation. To address this, Naor et al. have recently formalized view synchronization as a separate problem [38, 39]. Unlike us, they did not provide a comprehensive study of the applicability of their specifications to a wide range of modern Byzantine consensus protocols. In particular, their specifications do not expose bounds on how quickly processes switch views (Property 4 in Figure 1), which are necessary for protocols such as two-phase HotStuff (§4.2) and Tendermint (§4.3).

Naor et al. also proposed synchronizer implementations in a simplified variant of partial synchrony where δ is known a priori, and messages sent before GST are guaranteed to arrive by $\mathsf{GST} + \delta$ [38, 39]. These implementations focus on optimizing communication complexity, making it linear in best-case scenarios [38] or in expectation [39]. They achieve linearity by relying on digital signatures (more precisely, threshold signatures), which FastSync eschews. Unlike FastSync, they also require unbounded space (for the reasons explained in §3.1). Finally, we give exact latency bounds for FastSync under both favorable and unfavorable conditions whereas [38,39] only provide expected latency analysis. It is interesting to investigate whether the benefits of the two approaches can be combined to tolerate message loss before GST with both bounded space and a low communication complexity.

LibraBFT [2] extends HotStuff with a view synchronization mechanism, integrated with the core protocol; the protocol assumes reliable channels. LibraBFT is optimized to solve repeated consensus, whereas in this paper we focus on single-shot one. We leave investigating synchronizer abstractions optimized for the multi-shot case to future work.

The original idea of using synchronizers to simulate a round-based synchronous system on top of an asynchronous one is due to Awerbuch [10]. This work however, did not consider failures. Augmented round models to systematically study properties of distributed consensus under various failure and environment assumptions were proposed in [12, 23, 29, 33]. These papers however, do not deal with implementing the proposed models under partial synchrony. Upper bounds for deciding after GST in round-based crash fault-tolerant consensus algorithms were studied in [5, 26]. While we derive similar bounds for Byzantine failures, it remains open if these are optimal or can be further improved. Failure detectors [21, 22], which abstract away the timeliness guarantees of the environment, have been extensively used for developing and analyzing consensus algorithms [22, 37] in the presence of benign failures. However, since

capturing all possible faulty behaviors is algorithm-specific, the classical notion of a failure detector does not naturally generalize to Byzantine settings. As a result, the existing work on Byzantine failure detectors either limits the types of failures being addressed (e.g., [35]), or focuses on other means (such as accountability [32]) to mitigate faulty behavior.

References

1 Incorrect by construction-CBC Casper isn't live. https://pyrofex.io/wp-content/uploads/2018/12/Incorrect-By-Construction.pdf.

2 State machine replication in the Libra blockchain. https://developers.libra.org/docs/assets/papers/libra-consensus-state-machine-replication-in-the-libra-blockchain.pdf.

3 Ittai Abraham, Srinivas Devadas, Danny Dolev, Kartik Nayak, and Ling Ren. Synchronous Byzantine agreement with expected $O(1)$ rounds, expected $O(n^2)$ communication, and optimal resilience. In *Conference on Financial Cryptography and Data Security (FC)*, 2019.

4 Ittai Abraham, Guy Gueta, Dahlia Malkhi, Lorenzo Alvisi, Ramakrishna Kotla, and Jean-Philippe Martin. Revisiting fast practical Byzantine fault tolerance. *arXiv*, abs/1712.01367, 2017. `arXiv:1712.01367`.

5 Dan Alistarh, Seth Gilbert, Rachid Guerraoui, and Corentin Travers. How to solve consensus in the smallest window of synchrony. In *Symposium on Distributed Computing (DISC)*, 2008.

6 Yair Amir, Brian A. Coan, Jonathan Kirsch, and John Lane. Prime: Byzantine replication under attack. *IEEE Trans. Dependable Sec. Comput.*, 8(4):564–577, 2011.

7 Yackolley Amoussou-Guenou, Antonella Del Pozzo, Maria Potop-Butucaru, and Sara Tucci-Piergiovanni. Correctness of Tendermint-core blockchains. In *Conference on Principles of Distributed Systems (OPODIS)*, 2018.

8 Yackolley Amoussou-Guenou, Antonella Del Pozzo, Maria Potop-Butucaru, and Sara Tucci-Piergiovanni. Dissecting Tendermint. In *Conference on Networked Systems (NETYS)*, 2019.

9 Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolic, Sharon Weed Cocco, and Jason Yellick. Hyperledger Fabric: a distributed operating system for permissioned blockchains. In *European Conference on Computer Systems (EuroSys)*, 2018.

10 Baruch Awerbuch. Complexity of network synchronization. *J. ACM*, 32(4):804–823, 1985.

11 Rida A. Bazzi and Yin Ding. Non-skipping timestamps for Byzantine data storage systems. In *Symposium on Distributed Computing (DISC)*, 2004.

12 Martin Biely, Josef Widder, Bernadette Charron-Bost, Antoine Gaillard, Martin Hutle, and André Schiper. Tolerating corrupted communication. In *Symposium on Principles of Distributed Computing (PODC)*, 2007.

13 Gabriel Bracha. Asynchronous Byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.

14 Manuel Bravo, Gregory Chockler, and Alexey Gotsman. Making Byzantine consensus live (extended version). *arXiv*, abs/2008.04167, 2020. `arXiv:2008.04167`.

15 Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on BFT consensus. *arXiv*, abs/1807.04938, 2018. `arXiv:1807.04938`.

16 Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *arXiv*, abs/1710.09437, 2017. `arXiv:1710.09437`.

17 Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *International Cryptology Conference (CRYPTO)*, 2001.

18 Christian Cachin and Marko Vukolic. Blockchain consensus protocols in the wild (keynote talk). In *Symposium on Distributed Computing (DISC)*, 2017.

19 Miguel Castro. *Practical Byzantine Fault Tolerance*. PhD thesis, Massachusetts Institute of Technology, 2001.
20 Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 1999.
21 Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722, 1996.
22 Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.
23 Bernadette Charron-Bost and André Schiper. The Heard-Of model: computing in distributed systems with benign faults. *Distributed Comput.*, 22(1):49–71, 2009.
24 Tyler Crain, Vincent Gramoli, Mikel Larrea, and Michel Raynal. DBFT: efficient leaderless Byzantine consensus and its application to blockchains. In *Symposium on Network Computing and Applications (NCA)*, 2018.
25 Danny Dolev, Joseph Y. Halpern, Barbara Simons, and Ray Strong. Dynamic fault-tolerant clock synchronization. *J. ACM*, 42(1):143–185, 1995.
26 Partha Dutta, Rachid Guerraoui, and Leslie Lamport. How fast can eventual synchrony lead to consensus? In *Conference on Dependable Systems and Networks (DSN)*, 2005.
27 Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.
28 Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.
29 Eli Gafni. Round-by-round fault detectors: Unifying synchrony and asynchrony. In *Symposium on Principles of Distributed Computing (PODC)*, 1998.
30 Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling Byzantine agreements for cryptocurrencies. In *Symposium on Operating Systems Principles (SOSP)*, 2017.
31 Guy Golan-Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael K. Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. SBFT: A scalable and decentralized trust infrastructure. In *Conference on Dependable Systems and Networks (DSN)*, 2019.
32 Andreas Haeberlen and Petr Kuznetsov. The fault detection problem. In *Conference on Principles of Distributed Systems (OPODIS)*, 2009.
33 Idit Keidar and Alexander Shraer. Timeliness, failure-detectors, and consensus performance. In *Symposium on Principles of Distributed Computing (PODC)*, 2006.
34 Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.
35 Dahlia Malkhi and Michael Reiter. Unreliable intrusion detection in distributed computations. In *Workshop on Computer Security Foundations (CSFW)*, 1997.
36 Zarko Milosevic, Martin Biely, and André Schiper. Bounded delay in Byzantine-tolerant state machine replication. In *Symposium on Reliable Distributed Systems (SRDS)*, 2013.
37 Achour Mostéfaoui and Michel Raynal. Solving consensus using Chandra-Toueg's unreliable failure detectors: A general quorum-based approach. In *Symposium on Distributed Computing (DISC)*, 1999.
38 Oded Naor, Mathieu Baudet, Dahlia Malkhi, and Alexander Spiegelman. Cogsworth: Byzantine view synchronization. In *Cryptoeconomics Systems Conference (CES)*, 2020.
39 Oded Naor and Idit Keidar. Expected linear round synchronization: The missing link for linear Byzantine SMR. In *Symposium on Distributed Computing (DISC)*, 2020.
40 Barbara Simons, Jennifer Welch, and Nancy Lynch. An overview of clock synchronization. In *Fault-Tolerant Distributed Computing*, 1986.
41 Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan-Gueta, and Ittai Abraham. HotStuff: BFT consensus with linearity and responsiveness. In *Symposium on Principles of Distributed Computing (PODC)*, 2019.

Leaderless State-Machine Replication: Specification, Properties, Limits

Tuanir França Rezende
Telecom SudParis, Évry, France
tuanir.franca-rezende@telecom-sudparis.eu

Pierre Sutra
Telecom SudParis, Évry, France
pierre.sutra@telecom-sudparis.eu

Abstract

Modern Internet services commonly replicate critical data across several geographical locations using state-machine replication (SMR). Due to their reliance on a leader replica, classical SMR protocols offer limited scalability and availability in this setting. To solve this problem, recent protocols follow instead a leaderless approach, in which each replica is able to make progress using a quorum of its peers. In this paper, we study this new emerging class of SMR protocols and states some of their limits. We first propose a framework that captures the essence of leaderless state-machine replication (Leaderless SMR). Then, we introduce a set of desirable properties for these protocols: (R)eliability, (O)ptimal (L)atency and (L)oad Balancing. We show that protocols matching all of the ROLL properties are subject to a trade-off between performance and reliability. We also establish a lower bound on the message delay to execute a command in protocols optimal for the ROLL properties. This lower bound explains the persistent chaining effect observed in experimental results.

2012 ACM Subject Classification General and reference → Performance; Software and its engineering → Distributed systems organizing principles; Theory of computation → Distributed computing models

Keywords and phrases Fault Tolerance, State Machine Replication, Consensus

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.24

Related Version A full version of the paper is available at [22], `https://arxiv.org/abs/2008.02512`.

Funding This research is partly funded by the ANR RainbowFS project and the H2020 CloudButton project.

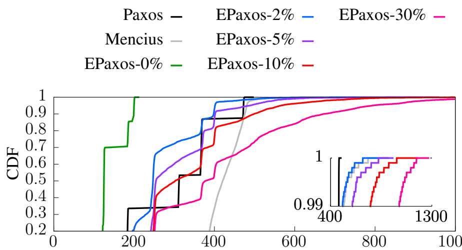
Acknowledgements The authors thank Vitor Enes and Alexey Gotsman for fruitful discussions on Leaderless SMR.

1 Introduction

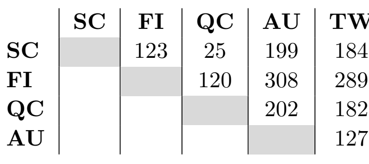
The standard way of implementing fault-tolerant distributed services is state-machine replication (SMR) [25]. In SMR, a service is defined by a deterministic state machine, and each process maintains its own local copy of the machine. Classical SMR protocols such as Paxos [13] and Raft [20] rely on a leader replica to order state-machine commands. The leader orchestrates a growing sequence of agreements, or consensus, each defining the next command to apply on the state machine. Such a scheme has however clear limitations, especially in a geo-distributed setting. First, it increases latency for clients that are far away from the leader. Second, as the leader becomes a bottleneck or its network gets slower, system performance decreases. Last, this approach harms availability because when the leader fails the whole system cannot serve new requests until an election takes place.

To sidestep the above limitations, a new class of leaderless protocols has recently emerged [18, 19, 3, 8, 26, 7]. These protocols allow any replica to make progress as long as it is able to contact enough of its peers. Mencius [18] pioneered this idea by rotating the ownership of consensus instances. Many other works have followed, and in particular the Egalitarian Paxos

© Tuanir França Rezende and Pierre Sutra;
licensed under Creative Commons License CC-BY
34th International Symposium on Distributed Computing (DISC 2020).
Editor: Hagit Attiya; Article No. 24; pp. 24:1–24:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



(a) Latency distribution when varying the conflict rate.



	SC	FI	QC	AU	TW
SC		123	25	199	184
FI			120	308	289
QC				202	182
AU					127

(b) Ping distance between sites (in ms).

Figure 1 Performance comparison of EPaxos, Paxos and Mencius – *5 sites: South Carolina (SC), Finland (FI), Canada (QC), Australia (AU), Taiwan (TW, leader); 128 clients per site;* no-op *service.*

(EPaxos) protocol [19]. As Generalized Paxos [14], EPaxos orders only non-commuting, aka. conflicting, state-machine commands. To this end, the protocol maintains at each replica a directed graph that stores the execution constraints between commands. Execution of a command proceeds by linearizing the graph of constraints. In the common case, EPaxos executes a command after two message delays if the fast path was taken, that is, if the replicas spontaneously agree on the constraints, and four message delays otherwise.

Problem statement. Unfortunately the latency of EPaxos may raise in practice well above four message delays. To illustrate this point, we ran an experimental evaluation of EPaxos, Paxos and Mencius in Google Cloud Platform. The results are reported in Figure 1a, where we plot the cumulative distribution function (CDF) of the command latency for each protocol. In this experiment, the system spans five geographical locations distributed around the globe, and each site hosts 128 clients that execute *no-op* commands in closed-loop. Figure 1b indicates the distance between any two sites. The conflict rate among commands varies from 0% to 30%.[1] We measure the latency from the submission of a command to its execution (at steady state).

Two observations can be formulated at the light of the results in Figure 1. First, the tail of the latency distribution in EPaxos is larger than for the two other protocols and it increases with the conflict rate. Second, despite Mencius clearly offering a lower median latency, it does not exhibit such a problem.

Contributions. In this paper, we provide a theoretical framework to understand and explain the above phenomena. We study in-depth this new class of leaderless state-machine replication (Leaderless SMR) protocols and state some of their limits.

Paper Outline. We recall the principles of state-machine replication (§2). Then, we define Leaderless SMR and deconstruct it into basic building blocks (§3). Further, we introduce a set of desirable properties for Leaderless SMR: (R)eliability, (O)ptimal (L)atency and (L)oad Balancing. Protocols that match all of the ROLL properties are subject to a trade-off between performance and reliability. More precisely, in a system of n processes, the ROLL theorem

[1] Each command has a key and any two commands conflict, that is they must be totally ordered by the protocol, when they have the same key. When a conflict rate ρ is applied, each client picks key 42 with probability ρ, and a unique key otherwise.

(§4) states that Leaderless SMR protocols are subject to the inequality $2F + f - 1 \leq n$, where $n - F$ is the size of the fast path quorum and f is the maximal number of tolerated failures. A protocol is ROLL-optimal when F and f cannot be improved according to this inequality. We establish that ROLL-optimal protocols are subject to a chaining effect that affect their performance (§5). As EPaxos is ROLL-optimal and Mencius not, the chaining effect explains the performance results observed in Figure 1. We discuss the implications of this result (§6) then put our work in perspective (§7) before closing (§8).

2 State machine replication

State-machine replication (SMR) allows a set of distributed processes to construct a linearizable shared object. The object is defined by a deterministic state machine together with a set of commands. Each process maintains its own local replica of the machine. An SMR protocol coordinates the execution of commands applied to the state machine, ensuring that the replicas stay in sync. This section recalls the fundamentals of SMR, as well as its generalization that leverages the commutativity of state-machine commands.

2.1 System model

We consider the standard model of wait-free computation in a distributed message-passing system where processes may fail-stop [9]. In [6], the authors extend this framework to include failure detectors. This paper follows such a model of distributed computation. Further details appear in [22].

2.2 Classic SMR

State machine replication is defined over a set of $n \geq 2$ processes Π using a set $\mathcal{C}$ of state-machine commands. Each process p holds a log, that is a totally ordered set of entries that we assume unbounded. Initially, each entry in the log is empty (i.e., $log_p[i] = \bot$ for $i \in \mathbb{N}$), and over time it may include one state-machine command. The operator $(log_p \bullet \mathsf{c})$ *appends* command c to the log, assigning it to the next free entry.

Commands are submitted by the processes that act as proxies on behalf of a set of remote clients (not modeled). A process takes the step *submit*(c) to submit command c for inclusion in the log. Command c is *decided* once it enters the log at some position i. It is executed against the state machine when all the commands at lower positions ($j < i$) are already executed. When the command is executed, its response value is sent back to the client. For simplicity, we shall consider that two processes may submit the same command.

When the properties below hold during every execution, the above construct ensures that the replicated state machine implements a linearizable shared object.

Validity: A command is decided once and only if it was submitted before.

Stability: If $log_p[i] = \mathsf{c}$ holds at some point in time, it is also true at any later time.

Consistency: For any two processes p and q, if $log_p[i]$ and $log_q[i]$ are both non-empty, then they are equal.

2.3 Generic SMR

In their seminal works, Pedone and Schiper [21] and concurrently Lamport [14] introduce an alternative approach to Classic SMR. They make the key observation that if commands submitted to the state machine commute, then there is no need to order them. Leveraging this, they replace the totally-ordered log used in Classic SMR by a partially-ordered one. We call this approach Generic SMR.

Two commands c and d do not commute when for some state s, applying cd to s differs from applying dc. This means that either both sequences do not lead to the same state, or one of the two commands does not return the same response value in the two sequences. Generic SMR relies on the notion of *conflicts* which captures a safe over-approximation of the non-commutativity of two state-machine commands. In what follows, conflicts are expressed as a binary, non-reflexive and symmetric relation $\asymp$ over $\mathcal{C}$.

In Generic SMR, each variable log_p is a partially ordered log, i.e., a directed acyclic graph [14]. In this graph, vertices are commands and any two conflicting commands have a directed edge between them. We use $G.V$ and $G.E$ to denote respectively the vertices of some partially ordered log G and its edges. The append operator is defined as follows: $G \bullet \mathsf{c} := (G.V \cup \{\mathsf{c}\}, G.E \cup \{(\mathsf{d}, \mathsf{c}) : \mathsf{d} \in G.V \wedge \mathsf{d} \asymp \mathsf{c}\}$. A command is decided once it is in the partially ordered log. As previously, it gets executed once all its predecessors are executed.

For correctness, Generic SMR defines a set of properties over partially ordered logs similar to Classic SMR. Stability is expressed in close terms, using a prefix relation between the logs along time. Consistency requires the existence of a common least upper bound over the partially ordered logs.

To state this precisely, consider two partially ordered logs G and H. G is prefix of H, written $G \sqsubseteq H$, when G is a subgraph of H and for every edge $(\mathsf{a}, \mathsf{b}) \in H.E$, if $\mathsf{b} \in G.V$ then $(\mathsf{a}, \mathsf{b}) \in G.E$. Given a set $\mathcal{G}$ of partially ordered logs, H is an *upper bound* of $\mathcal{G}$ iff $G \sqsubseteq H$ for every G in $\mathcal{G}$. Two logs G and H are *compatible* iff they have a common upper bound.[2] By extension, a set $\mathcal{G}$ of partially ordered logs is compatible iff its elements are pairwise compatible.

Based on the above definitions, we may express Generic SMR using the set of properties below. Validity is identical to Classic SMR and thus omitted.

Stability: For any process p, at any given time log_p is prefix of itself at any later time.

Consistency: The set of all the partially ordered logs is always compatible.

3 Leaderless SMR

Some recent protocols [18, 19] further push the idea of partially ordered log, as proposed in Generic SMR. In a leaderless state-machine replication (Leaderless SMR) protocol, there is no primary process to arbitrate upon the ordering of commands. Instead, any process may decide a command submitted to the replicated service. A command is stable, and thus executable, once the transitive closure of its predecessors is known locally. As this transitive closure can be cyclic, the log is replaced with a directed graph.

This section introduces a high-level framework to better understand Leaderless SMR. In particular, we present the notion of dependency graph and explain how commands are decided. With this framework, we then deconstruct several Leaderless SMR protocols into basic building blocks. Further, three key properties are introduced: Reliability, Optimal Latency and Load Balancing. These properties serve in the follow-up to establish lower bound complexity results for this class of protocols.

[2] In [14], compatibility is defined in terms of least upper bound between two c-structs. For partially ordered logs, the definition provided here is equivalent.

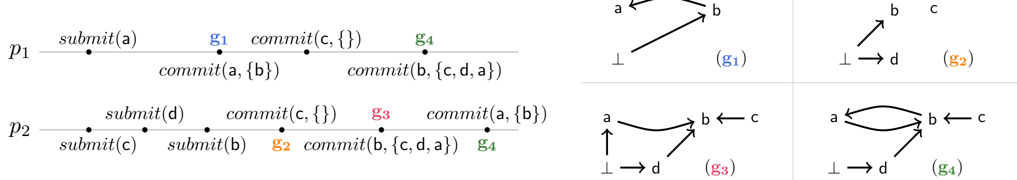


Figure 2 An example run of Leaderless SMR – *(left) processes p_1 and p_2 submit respectively the commands* $\{\mathsf{a}\}$ *and* $\{\mathsf{b}, \mathsf{c}, \mathsf{d}\}$*; (right) the dependencies graphs formed at the two processes.*

3.1 Definition

Leaderless SMR relies on the notion of dependency graph instead of partially ordered log as found in Generic SMR. A *dependency graph* is a directed graph that records the constraints defining how commands are executed. For some command c, the incoming neighbors of c in the dependency graph are its *dependencies*. As detailed shortly, the dependencies are executed either before or together with c.

In Leaderless SMR, a process holds two mapping: *deps* and *phase*. The mapping *deps* is a dependency graph storing a relation from $\mathcal{C}$ to $2^{\mathcal{C}} \cup \{\bot, \top\}$. For a command c, $\mathit{phase}(\mathsf{c})$ can take five possible values: pending, abort, commit, stable and execute. All the phases, except execute, correspond to a predicate over *deps*.

Initially, for every command c, $\mathit{deps}(\mathsf{c})$ is set to $\bot$. This corresponds to the pending phase. When a process decides a command c, it changes the mapping $\mathit{deps}(\mathsf{c})$ to a non-$\bot$ value. Operation $\mathit{commit}(\mathsf{c}, D)$ assigns D taken in $2^{\mathcal{C}}$ to $\mathit{deps}(\mathsf{c})$. Command c gets aborted when $\mathit{deps}(\mathsf{c})$ is set to $\top$. In that case, the command is removed from any $\mathit{deps}(\mathsf{d})$ and it will not appear later on. Let $\mathit{deps}^*(\mathsf{c})$ be the transitive closure of the *deps* relation starting from $\{\mathsf{c}\}$. Command c is stable once it is committed and no command in $\mathit{deps}^*(\mathsf{c})$ is pending.

Figure 2 depicts an example run of Leaderless SMR that illustrates the above definitions. In this run, process p_1 submits command a, while p_2 submits in order c, d then b. The timeline in Figure 2 indicates the timing of these submissions. It also includes events during which process p_1 and p_2 commits commands. For some of these events, we depict the state of the dependency graph at the process (on the right of Figure 2). As an example, the two processes obtain the graph $\mathfrak{g}_4$ at the end of the run. In this graph, a, b and c are all committed, while d is still pending. We have $\mathit{deps}(\mathsf{a}) = \{\mathsf{b}\}$ and $\mathit{deps}(\mathsf{b}) = \{\mathsf{a}, \mathsf{d}, \mathsf{c}\}$, with both $\mathit{deps}^*(\mathsf{a})$ and $\mathit{deps}^*(\mathsf{b})$ equal to $\{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}\}$. Only command c is stable in $\mathfrak{g}_4$.

Similarly to Classic and Generic SMR, Leaderless SMR protocols requires that validity holds. In addition, processes must agree on the value of *deps* for stable commands and conflicting commands must see each other. More precisely,

Stability: For each command c, there exists D such that if c is stable then $\mathit{deps}(\mathsf{c}) = D$.

Consistency: If a and b are both committed and conflicting, then $\mathsf{a} \in \mathit{deps}(\mathsf{b})$ or $\mathsf{b} \in \mathit{deps}(\mathsf{a})$.

A command c gets executed once it is stable. Algorithm 1 describes how this happens in Leaderless SMR. To execute command c, a process first creates a set of commands, or *batch*, β that execute together with c. This grouping of commands serves to maintain the following invariant:

▶ Invariant 1. *Consider two conflicting commands* c *and* d*. If p executes a batch of commands containing* c *before executing* d*, then $d \notin \mathit{deps}^*(c)$.*

Algorithm 1 Executing command c – code at process p.

```
1: execute(c) :=
2:     pre: phase(c) = stable
3:     eff: let β be the largest subset of deps*(c) satisfying ∀d ∈ β. phase(d) = stable
4:          forall d ∈ β ordered by →
5:              phase(d) ← execute
```

Satisfying Invariant 1 implies that if some command d is in batch β, then β also contains its transitive dependencies (line 3 in Algorithm 1). Inside a batch, commands are ordered according to the partial order $\rightarrow$ (line 4). Let $<$ be a canonical total order over $\mathcal{C}$. Then, $\mathsf{c} \rightarrow \mathsf{d}$ holds iff

(i) $\mathsf{c} \in deps^*(\mathsf{d})$ and $\mathsf{d} \notin deps^*(\mathsf{c})$; or

(ii) $\mathsf{c} \in deps^*(\mathsf{d})$, $\mathsf{d} \in deps^*(\mathsf{c})$ and $\mathsf{c} < \mathsf{d}$.

Relation $\rightarrow$ defines the *execution order* at a process. If there is a one-way dependency between two commands, Leaderless SMR plays them in the order of their transitive dependencies; otherwise the algorithm breaks the tie using the arbitrary order $<$. This guarantees the following invariant.

▶ INVARIANT 2. *Consider two conflicting commands* c *and* d*. If* p *executes* c *before* d *in the same batch, then* $\mathsf{c} \in deps^*(\mathsf{d})$*.*

Generic and Leaderless SMR are strongly similar. In fact, one may show that Generic SMR reduces to Leaderless SMR without requiring any message exchange. This result is stated in Theorem 1 below, and a proof appears in [22]. Let us observe that such a reduction does not hold between Classic and Generic SMR. Indeed, computing a total order on commuting commands would require processes to communicate.

▶ **Theorem 1.** *Generic SMR reduces to Leaderless SMR.*

However, Theorem 1 offers an incomplete picture of how the two abstractions compare in practice. Indeed, because the dependency graph might be cyclic, Leaderless SMR does not compute an ordering over conflicting commands. Instead, such commands must simply observe one another (Consistency property). This fundamental difference explains the absence of a leader in this class of SMR protocols, a feature that we capture in the next section.

3.2 Deciding commands

In Leaderless SMR, processes have to agree on the dependencies of stable commands. Thus, a subsequent refinement leads to consider a family of consensus objects $(\mathsf{CONS}_\mathsf{c})_{\mathsf{c}\in\mathcal{C}}$ for that purpose. For some command c, processes use CONS_c to decide either the dependencies of c, or the special value $(\top)$ signaling that the command is aborted. This agreement is driven by the command *coordinator* $(coord(\mathsf{c}))$, a process initially in charge of submitting the command to the replicated state machine. In a run during which there is no failure and the failure detector behaves perfectly, that is a *nice run*, only $coord(\mathsf{c})$ calls CONS_c.

To create a valid proposal for CONS_c, $coord(\mathsf{c})$ relies on the dependency discovery service (DDS). This shared object offers a single operation $announce(\mathsf{c})$ that returns a pair (D, b), where $D \in 2^{\mathcal{C}} \cup \{\top\}$ and $b \in \{0, 1\}$ is a flag. When the return value is in $2^{\mathcal{C}}$, the service suggests to commit the command. Otherwise, the command should be aborted. When the flag is set, the service indicates that a spontaneous agreement occurs. In such a case, the coordinator can directly commit c with the return value of the DDS service and bypass CONS_c; this is called *a fast path*. A *recovery* occurs when command c is announced at a process which is not $coord(\mathsf{c})$.

Algorithm 2 Deciding a command c – code at process p.

```
1: submit(c) :=
2:     pre:  p = coord(c) ∨ coord(c) ∈ D
3:     eff:  (D, b) ← DDS.announce(c)
4:           if b = false then D ← CONS_c.propose(D)
5:           deps(c) ← D
6:           send(c, deps(c)) to Π \ {p}
7:
8: when recv(c, D)
9:     eff:  deps(c) ← D
```

The DDS service ensures two safety properties. First, if two conflicting commands are announced, they do not miss each other. Second, when a command takes the fast path, processes agree on its committed dependencies.

More formally, assume that $announce_p(\mathsf{c})$ and $announce_q(\mathsf{c}')$ return respectively (D, b) and (D', b') with $D \in 2^{\mathcal{C}}$. Then, the properties of the DDS service are as follows.

Visibility: If $\mathsf{c} \asymp \mathsf{c}'$ and $D' \in 2^{\mathcal{C}}$, then $\mathsf{c} \in D'$ or $\mathsf{c}' \in D$.
Weak Agreement: If $\mathsf{c} = \mathsf{c}'$ and $b = true$, then $D' \in 2^{\mathcal{C}}$ and for every $\mathsf{d} \in D \oplus D'$, every invocation to $announce_r(\mathsf{d})$ returns $(\top, _)$.

To illustrate these properties, consider that no command was announced so far. In that case $(\varnothing, true)$ is a valid response to $announce(\mathsf{c})$. If $coord(\mathsf{c})$ is slow, then a subsequent invocation of $announce(\mathsf{c})$ may either return $\varnothing$, or a non-empty set of dependencies D. However in that case, because the fast path was taken by the coordinator, all the commands in D must eventually abort.

Based on the above decomposition of Leaderless SMR, Algorithm 2 depicts an abstract protocol to decide a command. This algorithm uses a family of consensus objects $((\mathsf{CONS}_\mathsf{c})_{\mathsf{c} \in \mathcal{C}})$, a dependency discovery service (DDS) and a failure detector ($\mathcal{D}$) that returns a set of suspected processes. To submit a command c, a process announces it then retrieves a set of dependencies. This set is proposed to CONS_c if the fast path was not taken (line 4). The result of the slow or the fast path determines the value of the local mapping $deps(\mathsf{c})$ to commit or abort command c. Notice that such a step may also be taken when a process receives a message from one of its peers (line 8).

During a nice run, the system is failure-free and the failure detector service behaves perfectly. As a consequence, only $coord(\mathsf{c})$ may propose a value to CONS_c and this value gets committed. In our view, this feature is the *key characteristic* of Leaderless SMR.

Below, we establish the correctness of Algorithm 2. A proof appears in [22].

▶ **Theorem 2.** *Algorithm 2 implements Leaderless SMR.*

3.3 Examples

To illustrate the framework introduced in the previous sections, we now instantiate well-known Leaderless SMR protocols using it.

Rotating coordinator. For starters, let us consider a rotating coordinator algorithm (e.g., [27]). In this class of protocols, commands are ordered *a priori* by some relation $\ll$. Such an ordering is usually defined by timestamping commands at each coordinator and breaking

ties with the process identities. When *coord*(c) calls DDS.*announce*(c), the service returns a pair (D, *false*), where D are all the commands prior to c according to $\ll$. Upon recovering a command, the DDS service simply suggests to abort it.

Clock-RSM. This protocol [7] improves on the above schema by introducing a fast path. It also uses physical clocks to speed-up the stabilization of committed commands. Once a command is associated to a timestamp, its coordinator broadcasts this information to the other processes in the system. When it receives such a message, a process waits until its local clock passes the command's timestamp to reply. Once a majority of processes have replied, the DDS service informs the coordinator that the fast path was taken.

Mencius. The above two protocols require a committed command to wait all its predecessors according to $\ll$. Clock-RSM propagates in the background the physical clock of each process. A command gets stable once the clocks of all the processes is higher than its timestamp. Differently, Mencius [18] aborts prior pending commands at the time the command is submitted. In detail, *announce*(c) first approximates D as all the commands prior to c according to $\ll$. Then, command c is broadcast to all the processes in the system. Upon receiving such a message, a process q computes all the commands d smaller than c it is coordinating. If d is not already announced, q stores that d will be aborted. Then, q sends d back to *coord*(c) that removes it from D. The DDS service returns (D, f) with f set to *true* if *coord*(c) received a message from everybody. Upon recovering c, if the command was received the over-approximation based on $\ll$ is returned together with the flag *false*. In case c is unknown, the DDS service suggests to abort it.

EPaxos. In [19], the authors present Egalitarian Paxos (EPaxos), a family of efficient Leaderless SMR protocols. For simplicity, we next consider the variation which does not involve sequence numbers. To announce a command c, the coordinator broadcasts it to a quorum of processes. Each process p computes (and records) the set of commands D_p conflicting with c it has seen so far. A call to *announce*(c) returns $(\cup_p D_p, b)$, with b set to *true* iff processes spontaneously agree on dependencies (i.e., for any p, q, $D_p = D_q$). When a process in the initial quorum is slow or a recovery occurs, c is broadcast to everybody. The caller then awaits for a majority quorum to answer and returns (D, *false*) such that if at least $\frac{f+1}{2}$ processes answer the same set of conflicts for c, then D is set to this value (with $n = 2f + 1$). Alternatively, if at least one process knows c, the union of the response values is taken. Otherwise, the DDS service suggests to abort c.

Caesar. To avoid cycles in the dependency graph, Caesar [3] orders commands using logical timestamps. Upon submitting a command c, the coordinator timestamps it with its logical clock then it executes a broadcast. As with EPaxos, when it receives c a process p computes the conflicting commands D_p received so far. Then, it awaits until there is no conflicting command d with a higher timestamp than c such that c $\notin$ *deps*(d). If such a command exists, p replies to the coordinator that the fast path cannot be taken. The DDS service returns $(\cup_p D_p, b)$, where $b =$ *true* iff no process disables the fast path.

The above examples show that multiple implementations are possible for Leaderless SMR. In the next section, we introduce several properties of interest to characterize them.

3.4 Core properties

State machine replication helps to mask failures and asynchrony in a distributed system. As a consequence, a first property of interest is the largest number of failures (parameter f) tolerated by a protocol. After f failures, the protocol may not guarantee any progress.[3]

(*Reliability*) In every run, if there are at most f failures, every submitted command gets eventually decided at every correct process.

Leaderless SMR protocols exploit the absence of contention on the replicated service to boost performance. In particular, some protocols are able to execute a command after a single round-trip, which is clearly optimal [16]. To ensure this property, the fast path is taken when there is no concurrent conflicting command. Moreover, the command stabilizes right away, requiring that the DDS service returns only submitted commands.

(*Optimal Latency*) During a nice run, every call to $announce(c)$ returns a tuple (D, b) after two message delays such that
- **(i)** if there is no concurrent conflicting command to c, then b is set to *true*,
- **(ii)** $D \in 2^{\mathcal{C}}$, and
- **(iii)** for every $d \in D$, d was announced before.

The replicas that participate to the fast path vary from one protocol to another. Mencius use all the processes. On the contrary, EPaxos solely contact $\lfloor \frac{3n}{4} \rfloor$ of them (or equivalently, $f + \frac{f+1}{2}$ when $n = 2f + 1$). For some command c, a *fast path quorum* for c is any set of $n - F$ replicas that includes the coordinator of c. Such a set is denoted $FQuorums(c)$ and formally defined as $\{Q \mid Q \subseteq \Pi \wedge coord(c) \in Q \ \wedge |Q| \geq n - F\}$. A protocol has the *Load Balancing* property when it may freely choose fast path quorums to make progress.

(*Load Balancing*) During a nice run, any fast path quorum in $FQuorums(c)$ can be used to announce a command c.

The previous properties are formally defined in [22]. Table 1 indicates how they are implemented by well-known leaderless protocols. The columns 'Reliability' and 'Load Balancing' detail respectively the maximum number of failures tolerated by the protocol and the size of the fast path quorum. Notice that by CAP [11], we have $F, f \leq \lfloor \frac{n-1}{2} \rfloor$ when the protocol matches all of the properties. Table 1 also mentions the optimality of each protocol with respect to the ROLL theorem. This theorem is stated in the next section and establishes a trade-off between fault-tolerance and performance in Leaderless SMR.

4 The ROLL theorem

Reliability, Optimal Latency and Load Balancing are called collectively the ROLL properties. These properties introduce the parameters f and F as key characteristics of a Leaderless SMR protocol. Parameter f translates the reliability of the protocol, stating that progress is guaranteed only if less than f processes crash. Parameter F captures its scalability since, any quorum of $n - F$ processes may be used to order a command. An ideal protocol should strive to minimize $n - F$ while maximizing f.

[3] When f failures occur, the system configuration must change to tolerate subsequent ones. If data is persisted (as in Paxos [13]), the protocol simply stops when more than f failures occurs and awaits that faulty processes are back online.

Table 1 The properties of several leaderless SMR protocols – Min stands for a minority of replicas ($\lfloor \frac{n-1}{2} \rfloor$), Maj a majority ($\lceil \frac{n+1}{2} \rceil$), and LMaj a large majority ($\lfloor \frac{3n}{4} \rfloor$).

Protocols	***Properties***			
	Load Balancing ($n - F$)	Reliability (f)	Optimal Latency	ROLL-optimal
Rotating coord.	0	Min	×	×
Clock-RSM [7]	n	Min	×	×
Mencius [18]	n	Min	√	×
Caesar [3]	$\lceil \frac{3n}{4} \rceil$	Min	√	×
EPaxos [19]	LMaj	Min	√	if $n = 2f + 1$
Alvin [26]	LMaj	Min	√	if $n = 2f + 1$
Atlas [8]	$\lfloor \frac{n}{2} \rfloor + f$	any	√	if $n \in 2\mathbb{N} \cup \{3\} \wedge f = 1$

Unfortunately, we show that there is no free-lunch and that an optimization choice must be made. The ROLL theorem below establishes that $2F + f - 1 \leq n$ must hold. This inequality captures that every protocol must trade scalability for fault-tolerance. EPaxos [19] and Atlas [8] illustrate the two ends of the spectrum of solutions (see Table 1). EPaxos supports that any minority of processes may fail, but requires large quorums. Atlas typically uses small fast path quorums ($\lfloor \frac{n}{2} \rfloor + f$), but exactly handles at most f failures.

Below, we state the ROLL theorem and provide a sketch of proof illustrated in Figure 3. A formal treatment appears in [22].

▶ **Theorem 3** (ROLL). *Consider an SMR protocol that satisfies the ROLL properties. Then, it is true that* $2F + f - 1 \leq n$.

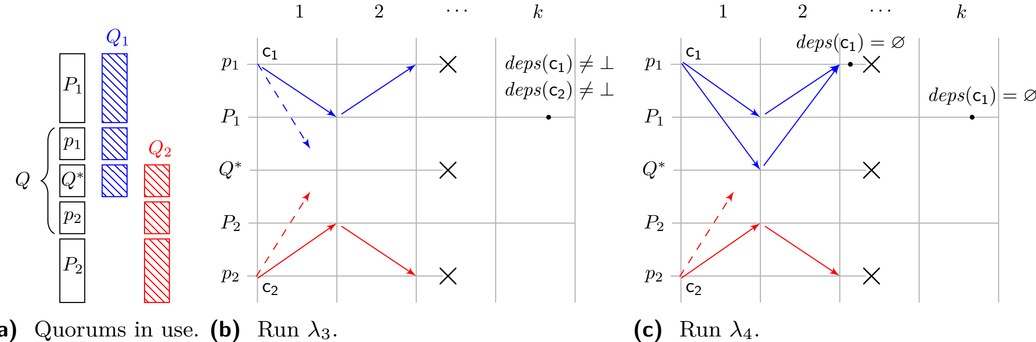
Proof (Sketch). Our proof goes by contradiction, using a round-based reasoning. Let us assume a protocol $\mathcal{P}$ that satisfies all the ROLL properties with $2F + f - 1 > n$. Then, choose two non-commuting commands c_1 and c_2 in $\mathcal{C}$.

As depicted in Figure 3a, the distributed system is partitioned into three sets: P_1 and P_2 are two disjoints sets of $F - 1$ processes, and the remaining $n - 2(F - 1)$ processes form Q. The CAP impossibility result [11] tells us that $2F < n$. As a consequence, there exist at least two distinct processes p_1 and p_2 in Q. We define Q_1 and Q_2 as respectively $P_1 \cup Q \setminus \{p_2\}$ and $P_2 \cup Q \setminus \{p_1\}$. The set Q^* equals $Q \setminus \{p_1, p_2\}$.

Let λ_1 be a nice run that starts from the submission of c_1 by process p_1 during which only Q_1 take steps. Since Q_1 contains $n - F$ processes such a run exists by the Load Balancing property of $\mathcal{P}$. By Optimal Latency, this run lasts two rounds and $deps(\mathsf{c}_1)$ is set to $\varnothing$ at process p_1. Similarly, we may define λ_2 a run in which p_2 announces command c_2 and in which only the processes in Q_2 participate.

Then, consider a run λ_3 in which p_1 and p_2 submit concurrently commands c_1 and c_2. This run is illustrated in Figure 3b. At the end of the first round, the processes in P_1 (respectively, P_2) receive the same messages as in λ_1 (resp., λ_2). At the start of the second round, they reply to respectively p_1 and p_2 as in λ_1 and λ_2. All the other messages sent in the first two rounds are arbitrarily slow. The processes in Q crash at the end of the second round. By Reliability and as $f \geq |Q|$, the commands c_1 and c_2 are stable in λ_3. Let k be the first round at which the two commands are stable at some process $p \in P_1 \cup P_2$.

We now build an admissible run λ_4 of $\mathcal{P}$ as follows. The failure pattern and failure detector history are the same as in λ_3. Commands c_1 and c_2 are submitted concurrently at



(a) Quorums in use. **(b)** Run λ_3. **(c)** Run λ_4.

Figure 3 Illustration of Theorem 3 – *slow messages are omitted.*

the start of λ_4, as in λ_3. In the first two rounds, P_1 receives the same messages as in λ_1 while P_2 receives the same messages as in λ_3. The other messages exchanged during the first two rounds are arbitrarily slow. Figure 3c depicts run λ_4.

Observe that the following claims about λ_4 are true. First, (C1) for p_1, λ_4 is indistinguishable to λ_1 up to round 2. Moreover, (C2) for the processes in $(P_1 \cup P_2)$, λ_4 is indistinguishable to λ_3 up to round k. From (C1), c_1 is stable at p_1 with $deps(\mathsf{c}_1) = \varnothing$. Claim (C2) implies that both c_1 and c_2 are stable at p when round k is reached. By the stability property of Leaderless SMR, process p and p_1 decide the same dependencies for c_1, i.e., $deps(\mathsf{c}_1) = \varnothing$.

A symmetric argument can be made using run λ_2 and a run λ_5, showing that p decides $deps(\mathsf{c}_2) = \varnothing$ in λ_3. It follows that in λ_3, an empty set of dependencies is decided for both commands at process p; a contradiction to the Consistency property. ◀

Theorem 3 captures an inherent trade-off between performance and reliability for ROLL protocols. For instance, tolerating a minority of crashes, requires accessing at least $\lfloor \frac{3n}{4} \rfloor$ processes. This is the setting under which EPaxos operates. On the other hand, if the protocol uses a plain majority quorum in the fast path, it tolerates at most one failure.

4.1 Optimality

A protocol is *ROLL-optimal* when the parameters F and f cannot be improved according to Theorem 3. In other words, they belong to the skyline of solutions [5]. As an example, when the system consists of 5 processes, there is a single such tuple $(F, f) = (2, 2)$. With $n = 7$, there are two tuples in the skyline, $(2, 3)$ and $(3, 2)$. The first one is attained by EPaxos, while Atlas offers the almost optimal solution $(3, 1)$ (see Table 1).

For each protocol, Table 1 lists the conditions under which ROLL-optimality is attained. EPaxos and Alvin are both optimal under the assumption that $n = 2f + 1$. Atlas adjusts the fast path quorums to the value of f, requiring $\lfloor \frac{n}{2} \rfloor + f$ processes to participate. This is optimal when $f = 1$ and either n is even or equals to 3. In the general case, the protocol is within $O(f)$ of the optimal value. As it uses classical Fast Paxos quorums, Caesar is not ROLL-optimal. This is also the case of protocols that contact all of the replicas to make progress, such as Mencius and Clock-RSM. To the best of our knowledge, no protocol is optimal in the general case.

In the next section, we show that ROLL-optimality has a price. More precisely, we establish that by being optimal, a protocol may create an arbitrarily long chain of commands, even during a nice run. This chaining effect may affect adversely the performance of the protocol. We discuss measures of mitigation in §6.

5 Chaining effect

This section shows that a chaining effect may affect ROLL-optimal protocols. It occurs when the chain of transitive dependencies of a command keeps growing after it gets committed. This implies that the committed command takes time to stabilize, thus delaying its execution and increasing the protocol latency.

At first glance, one could think that this situation arises from the asynchrony of the distributed system. As illustrated in Figure 1, this is not the case. We establish that such an effect may occur during "almost" synchronous runs.

The remaining of this section is split as follows. First, we define the notion of chain, that is a dependency-related set of commands. A chain is live when its last command is not stable. To measure how asynchronous a nice run is, we then introduce the principle of k-asynchrony. A run is k-asynchronous when some message is concurrent to both the first and last message of a sequence of k causally-related messages.

At core, our result shows how to inductively add a new link to a live chain during an appropriate 2-asynchronous run of a ROLL-optimal protocol.

5.1 Notion of chain

A chain is a sequence of commands $\mathsf{c}_1 \ldots \mathsf{c}_\mathsf{n}$ such that for any two consecutive commands $(\mathsf{c}_\mathsf{i}, \mathsf{c}_{\mathsf{i}+1})$ in the chain, $\mathsf{c}_\mathsf{i} \in deps(\mathsf{c}_{\mathsf{i}+1})$ at some process. Two consecutive commands (c, d) in a chain form a *link*. For instance, in the dependency graph g_4 (see Figure 2), cba is a chain.

We shall say that a chain is *live* when its first command is not stable yet (at any of the processes). In g_4, this is the case of the chain dba, since command d is still pending ($deps(\mathsf{d}) = \bot$). When a chain is live, the last command in the chain has to wait to ensure a sound execution order across processes. This increases the protocol latency.

5.2 A measure of asynchrony

In a synchronous system [17], processes executes rounds in lock-step. During a round, the messages sent at the beginning are received at the end (provided there is no failure). On the other hand, a partially synchronous system may delay messages for an arbitrary amount of time. In this model, we propose to measure asynchrony by looking at the overlaps between the exchanges of messages. The larger the overlap is, the more asynchronous is the run.

To illustrate this idea, consider the run depicted in Figure 4. During this run, a red message is sent from p_5 to p_4 (bottom left corner of the figure). In the same amount of time p_1 sends a blue message to p_2 which is followed by a green message to p_4. To characterize such an asynchrony, we shall say that the run is 2-asynchronous. This notion is precisely defined below.

▶ **Definition 1** (Path). *A sequence of event $\rho = send_p(m_1)recv_q(m_1)send_q(m_2)\ldots recv_t(m_{k\geq 1})$ in a run is called a* path. *We note $\rho[i]$ the i-th message in the path. The number of messages in the path, or its* size, *is denoted $|\rho|$.*

▶ **Definition 2** (Overlapping). *Two messages m and m' are overlapping when their respective events are concurrent.*[4] *By extension, a message m overlaps with a path ρ when it overlaps with both $\rho[1]$ and $\rho[|\rho|]$.*

▶ **Definition 3** (k-asynchrony). *A run λ is k-asynchronous when for every message m, if m overlaps with a path ρ then $|\rho| \leq k$.*

---

[4] That is, neither $recv(m)$ precedes $send(m')$, nor $recv(m')$ precedes $send(m)$ in real-time.

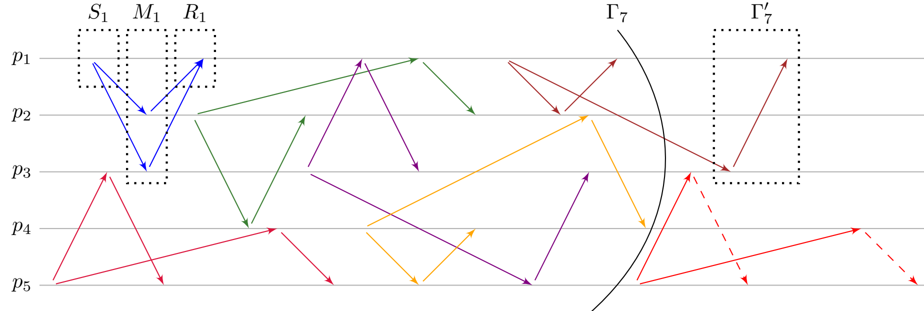


Figure 4 Theorem 4 for $n = 5$ and $k = 7$. The chain $\mathsf{c}_7\mathsf{c}_6\mathsf{c}_5\mathsf{c}_4\mathsf{c}_3\mathsf{c}_2\mathsf{c}_1$ is formed in σ_7. Illustrating the steps S_1, M_1 and R_1 for command c_1, the prefix Γ_7 of σ_7, and the steps Γ'_7.

5.3 Result statement

The theorem below establishes that a ROLL-optimal protocol may create a live chain of arbitrary size during a 2-asynchronous nice run. The full proof appears in [22].

▶ **Theorem 4** (Chaining Effect). *Assume a ROLL-optimal protocol $\mathcal{P}$. For any $k > 0$, there exists a 2-asynchronous nice run of $\mathcal{P}$ containing a live chain of size k.*

Proof (Sketch). The theorem is proved by adding inductively a new link to a live chain of commands created during a nice run. It is illustrated in Figure 4 for a system of five processes when $k = 7$.

The proof is based on the following two key observations about ROLL-optimal protocols. First, during a nice run, the coordinator of a command never rotates. As a consequence, the return value of the DDS service at the coordinator is always the stable value of $deps(c)$. Second, as the protocol satisfies the ROLL properties, a call to $announce(c)$ consists of sending a set of requests to the fast path quorum and receiving a set of replies. As a consequence, its execution can be split into the steps $S_cM_cR_c$, where

(S_c) are the steps taken from announcing c to the sending of the last request at the coordinator;

(R_c) are the steps taken by $coord(c)$ after receiving the first reply until the announcement returns; and

(M_c) are the steps taken during the announcement of c which are neither in S_c, nor in R_c.

By Optimal Latency, this sequence of steps do not create pending messages. As an illustration, the steps S_1, M_1 and R_1 taken to announce command c_1 are depicted in Figure 4.

Leveraging the above two observations, the result is built inductively using a family of k distinct commands $(\mathsf{c}_i)_{i\in[1,k]}$. Each command is associated with a nice run (σ_i), a fast path quorum (Q_i), a subset of $f-1$ processes (P_i), and a process (q_i).

Given a sequence of steps λ and a set of processes Q, let us note $\lambda|Q$ the sub-sequence of steps by Q in λ. We establish that at rank $i > 0$ the following property $\mathfrak{P}(i)$ holds: There exists a 2-asynchronous run σ_i of the form $\Gamma_i S_i(M_i|P_i)\Gamma'_i(M_i|Q_i \setminus P_i)R_i$ such that

(1) $proc(\Gamma'_i) \cap Q_i = P_i$;

(2) every path in Γ'_i is as most of size one;

(3) no message is pending in σ_i; and

(4) σ_i contains a chain $\mathsf{c}_i\mathsf{c}_{i-1}\cdots\mathsf{c}_1$.

Figure 4 depicts the run σ_7, its prefix Γ_7 and the steps Γ'_7.

Starting from $\mathfrak{P}(i)$, we establish $\mathfrak{P}(i+1)$ as follows. First we show that σ_{i+1} as $\Gamma_{i+1}S_{i+1}(M_{i+1}|P_{i+1})\Gamma'_{i+1}(M_{i+1}|Q_{i+1} \setminus P_{i+1})R_{i+1}$, where $\Gamma_{i+1} = \Gamma_i S_i(M_i|P_i)\Gamma'_i$, and $\Gamma'_{i+1} = (M_i|Q_i \setminus P_i)R_i$ is a nice run.

At rank $i+1$, item (1) is proved with appropriate definitions of the quorums (Q_i and Q_{i+1}), and the sub-quorum (P_i). For instance, in Figure 4, the command c_1 and c_2 have respectively $\{p_1, p_2, p_3\}$ and $\{p_3, p_4, p_5\}$ for fast path quorums. The sub-quorum P_2 is set to the intersection of Q_1 and Q_2, that is $\{p_3\}$. Item (2) follows from the definition of Γ'_{i+1}. The Load-Balancing property implies that (3) holds. A case analysis can then show that σ_{i+1} is 2-asynchronous. It relies on the fact that the $(SMR)_{i+1}$ steps create no pending message and the induction property $\mathfrak{P}(i)$.

To prove that a new link was added, we show that σ_{i+1} is indistinguishable to $coord(\mathsf{c_i})$ to a run in which $\mathsf{c_{i+1}}$ gets committed while missing $\mathsf{c_i}$. Going back to Figure 4, observe that the coordinator of $\mathsf{c_6}$ does not know that the replies of p_2 for command $\mathsf{c_5}$ causally precedes the replies of p_4 to $\mathsf{c_7}$. As a consequence, it must add $\mathsf{c_7}$ to the return value of $DDS.announce(\mathsf{c_6})$.

Finally, to obtain a live chain of size k, it suffices to consider the prefix of σ_{i+1} which does not contain the replies of the fast path quorum. In Figure 4, this corresponds to omitting the dashed messages that contain the reply to the announcement of $\mathsf{c_7}$ ◀

6 Discussion

Leaderless SMR offers appealing properties with respect to leader-driven approaches. Protocols are faster in the best case, suffer from no downtime when the leader fails, and distribute the load among participants. For instance, Figure 1 shows that EPaxos is strictly better than Paxos when there is no conflict. However, the latency of a command is strongly related to its dependencies in this family of SMR protocols. Going back to Figure 1, the bivariate correlation between the latency of a command and the size of the batch with which it executes is greater than 0.7.

Several approaches are possible to mitigate the chaining effect established in Theorem 4. Moraru et al. [19] propose that pure writes (i.e., commands having no response value) return once they are committed.[5] In [8], the authors observe that as each read executes at a single process, they can be excluded from the computation of dependencies. A third possibility is to wait until prior commands return before announcing a new one. However, in this case, it is possible to extend Theorem 4 by rotating the command coordinators to establish that a chain of size n can form.

In ROLL, the Load-Balancing and Optimal-Latency properties constrain the form of the DDS service. More precisely, in a contention-free case, executing the service must consist in a back-and-forth between the command coordinator and the fast path quorum. A weaker definition would allow some messages to be pending when *announce* returns. In this case, it is possible to sidestep the ROLL theorem provided that the system is synchronous: When replying to an announcement a process first sends its reply to the other fast path quorum nodes. The fast path is taken by merging all of the replies. Since the system is synchronous, a process recovering a command will retrieve all the replies at any node in the fast path quorum. Note that under this weaker definition, the ROLL theorem (Theorem 3) still applies in a partially synchronous model. Moreover, a chaining effect (Theorem 4) is also possible, but it requires more asynchrony during a nice run.

---

[5] In fact, it is possible to return even earlier, at the durable signal, that is once $f+1$ processes have received the command. To ensure linearizability, a later read must however wait for all the prior (conflicting or not) preceding writes.

7 Related work

Protocols. Early leaderless solutions to SMR include rotating coordinators and deterministic merge, aka. collision-fast, protocols. We cover the first class of protocols in §3.3. In a collision-fast protocol [1, 24], processes replicate an infinite array of vector consensus instances. Each vector consensus corresponds to a round. During a round, each process proposes a command (or a batch) to its consensus instance in the vector. If the process is in late, its peers may take over the instance and propose an empty batch of commands. Commands are executed according to their round numbers, applying an arbitrary ordering per round. The size of the vector can change dynamically, adapting to network conditions and/or the application workload. This technique is also used in Paxos Commit [12].

When the ordering is fixed beforehand, processes must advance at the same pace. To fix this issue, Mencius [18] includes a piggy-back mechanism that allows a process to bail out its instances (i.e., proposing implicitly an empty batch). Clock-RSM [7] follows a similar schema, using physical clocks to bypass explicit synchronization in the good cases.

With the above protocols, commands still get delayed by slow processes. Avoiding this so-called delayed commit problem [18] requires to dynamically discover dependencies at runtime. This is the approach introduced in Zieliński's optimistic generic broadcast [29] and EPaxos [19]. Here, as well as in [8], replicas agree on a fully-fledged dependency graph. Caesar [3] uses timestamps to avoid cycles in the graph. However, even in contention-free cases, committing a command can take two round trips. In our classification (see Table 1), this protocol does not have Optimal Latency.

Deconstruction. In [4], the authors introduce the dependency-set and map-agreement algorithms. The two services allow respectively to gather dependencies and agree upon them. A similar decomposition is proposed in [28]. Compared to these prior works, our framework includes the notion of fast path and distinguishes committed and stable commands. An agreement between the processes is necessary only eventually and on the stable part of the dependency graph. This difference allows to capture a wider spectrum of protocols. Our dependency discovery service (DDS) is reminiscent of an adopt-commit object [10] that allows processes to reach a weak agreement. In our case, when the fast path flag is set, processes may disagree on at most the aborted dependencies of a command.

Complexity. Multiple works study the complexity of consensus, the key underlying building block of SMR. Lamport [16] proves several lower bounds on the time and space complexity of this abstraction. The Hyperfast Learning theorem establishes that consensus requires one round-trip in the general case. This explains why we call optimal protocols that return after two message delays. The Fast Learning theorem requires that $n > 2F + f$. This result explains the trade-off between fault-tolerance and performance in Fast Paxos [15]. However, it does not readily apply to Leaderless SMR because only coordinator-centric quorums are fast in that case. For instance, EPaxos is able to run with $F = 1$ and $f = 1$ in a 3-process system. The ROLL theorem (§4) accurately captures this difference.

Traditional complexity measures for SMR and consensus (e.g., the latency degree [23]) consider contention-free and/or perfectly synchronous scenarios. In [2], the authors study the complexity of SMR over long runs. The paper shows that completing an SMR command can be more expensive than solving a consensus instance. Their complexity measure is different from ours and given in terms of synchronous rounds. In §5, we show that in an almost synchronous scenario, contention may create arbitrarily long chains in Leaderless SMR. We discuss mitigation measures in §6.

8 Conclusion

This paper introduces a framework to decompose leaderless state-machine replication (Leaderless SMR) protocols. The framework allows to break down representative protocols into two simple building blocks: a dependency discovery service and a consensus service. We then define a set of desirable properties for Leaderless SMR: (R)eliability, (O)ptimal (L)atency and (L)oad balancing. Protocols matching all of these properties satisfy the inequality $2F + f - 1 \leq n$, where n is number of processes, f the maximum number of failures tolerated by the protocol, and $n - F$ the size of the fast path quorum. Further, we establish that protocols that optimally solve this inequality suffer from a chaining effect. This effect explains the tail latency of some Leaderless SMR protocols in real-world deployments.

References

1. Marcos Kawazoe Aguilera and Robert E. Strom. Efficient atomic broadcast using deterministic merge. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, page 209–218, New York, NY, USA, 2000. Association for Computing Machinery. doi:10.1145/343477.343620.
2. Karolos Antoniadis, Rachid Guerraoui, Dahlia Malkhi, and Dragos-Adrian Seredinschi. State machine replication is more expensive than consensus. In *32nd International Symposium on Distributed Computing, DISC 2018, New Orleans, LA, USA, October 15-19, 2018*, pages 7:1–7:18, 2018. doi:10.4230/LIPIcs.DISC.2018.7.
3. Balaji Arun, Sebastiano Peluso, Roberto Palmieri, Giuliano Losa, and Binoy Ravindran. Speeding up consensus by chasing fast decisions. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 49–60, 2017.
4. Marijke H. L. Bodlaender, Magnús M. Halldórsson, Christian Konrad, and Fabian Kuhn. Brief announcement: Local independent set approximation. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, July 25-28, 2016*, pages 93–95, 2016. doi:10.1145/2933057.2933068.
5. Stephan Börzsönyi, Donald Kossmann, and Konrad Stocker. The skyline operator. In *Proceedings of the 17th International Conference on Data Engineering*, page 421–430, USA, 2001. IEEE Computer Society.
6. T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Communications of the ACM*, 43(2):225–267, 1996. URL: http://www.acm.org/pubs/toc/Abstracts/jacm/226647.html.
7. Jiaqing Du, Daniele Sciascia, Sameh Elnikety, Willy Zwaenepoel, and Fernando Pedone. Clock-RSM: Low-Latency Inter-datacenter State Machine Replication Using Loosely Synchronized Physical Clocks. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*, pages 343–354, 2014. doi:10.1109/DSN.2014.42.
8. Vitor Enes, Carlos Baquero, Tuanir França Rezende, Alexey Gotsman, Matthieu Perrin, and Pierre Sutra. State-machine replication for planet-scale systems. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3342195.3387543.
9. Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985. doi:10.1145/3149.214121.
10. Eli Gafni. Round-by-round fault detectors (extended abstract): unifying synchrony and asynchrony. In *Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing*, PODC '98, pages 143–152, New York, NY, USA, 1998. ACM.

11 Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002. doi:10.1145/564585.564601.

12 Jim Gray and Leslie Lamport. Consensus on transaction commit. *ACM Trans. Database Syst.*, 31(1):133–160, March 2006. doi:10.1145/1132863.1132867.

13 Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

14 Leslie Lamport. Generalized consensus and Paxos. Technical Report MSR-TR-2005-33, Microsoft Research, 2005.

15 Leslie Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, October 2006.

16 Leslie Lamport. Lower bounds for asynchronous consensus. *Distributed Computing*, 19(2):104–125, 2006.

17 Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

18 Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. Mencius: Building efficient replicated state machines for wans. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 369–384, 2008.

19 Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 358–372, 2013.

20 Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference (USENIX ATC)*, pages 305–320, 2014.

21 Fernando Pedone and André Schiper. Generic broadcast. In *International Symposium on Distributed Computing (DISC)*, pages 94–108, 1999.

22 Tuanir França Rezende and Pierre Sutra. Leaderless state-machine replication: Specification, properties, limits (extended version), 2020. arXiv:2008.02512.

23 André Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distrib. Comput.*, 10(3):149–157, April 1997. doi:10.1007/s004460050032.

24 R. Schmidt, L. Camargos, and F. Pedone. Collision-fast atomic broadcast. In *2014 IEEE 28th International Conference on Advanced Information Networking and Applications*, pages 1065–1072, 2014.

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

26 Alexandru Turcu, Sebastiano Peluso, Roberto Palmieri, and Binoy Ravindran. Be general and don't give up consistency in geo-replicated transactional systems. In *International Conference on Principles of Distributed Systems (OPODIS)*, pages 33–48, 2014.

27 Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. Spin one's wheels? byzantine fault tolerance with a spinning primary. In *Proceedings of the 2009 28th IEEE International Symposium on Reliable Distributed Systems*, SRDS '09, page 135–144, USA, 2009. IEEE Computer Society. doi:10.1109/SRDS.2009.36.

28 Michael Whittaker, Neil Giridharan, Adriana Szekeres, Joseph M. Hellerstein, and Ion Stoica. "bipartisan paxos: A family of fast, leaderless, modular state machine replication protocols". preprint on webpage at https://mwhittaker.github.io/publications/bipartisan_paxos.pdf.

29 Piotr Zieliński. Optimistic generic broadcast. In Pierre Fraigniaud, editor, *Distributed Computing*, pages 369–383, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

Not a COINcidence: Sub-Quadratic Asynchronous Byzantine Agreement WHP

Shir Cohen
Technion – Israel Institute of Technology, Haifa, Israel
shirco@campus.technion.ac.il

Idit Keidar
Technion – Israel Institute of Technology, Haifa, Israel
idish@ee.technion.ac.il

Alexander Spiegelman
VMware Research, Herzliya, Israel
sasha.spiegelman@gmail.com

Abstract

King and Saia were the first to break the quadratic word complexity bound for Byzantine Agreement in synchronous systems against an adaptive adversary, and Algorand broke this bound with near-optimal resilience (first in the synchronous model and then with eventual-synchrony). Yet the question of asynchronous sub-quadratic Byzantine Agreement remained open. To the best of our knowledge, we are the first to answer this question in the affirmative. A key component of our solution is a shared coin algorithm based on a VRF. A second essential ingredient is VRF-based committee sampling, which we formalize and utilize in the asynchronous model for the first time. Our algorithms work against a delayed-adaptive adversary, which cannot perform after-the-fact removals but has full control of Byzantine processes and full information about communication in earlier rounds. Using committee sampling and our shared coin, we solve Byzantine Agreement with high probability, with a word complexity of $\widetilde{O}(n)$ and $O(1)$ expected time, breaking the $O(n^2)$ bit barrier for asynchronous Byzantine Agreement.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Theory of computation → Cryptographic primitives; Mathematics of computing → Probabilistic algorithms

Keywords and phrases shared coin, Byzantine Agreement, VRF, sub-quadratic consensus protocol

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.25

Related Version A full version of the paper is available at [16], `https://arxiv.org/abs/2002.06545`.

Acknowledgements We thank Ittai Abraham, Dahlia Malkhi, Kartik Nayak and Ling Ren for insightful initial discussions.

1 Introduction

Byzantine Agreement (*BA*) [28] has been studied for four decades by now, but until recently, has been considered at a fairly small scale. In recent years, however, we begin to see practical use-cases of BA in large-scale systems, which motivates a push for reduced communication complexity. In deterministic algorithms, Dolev and Reischuk's renown lower bound stipulates that $\Omega(n^2)$ communication is needed [18], and until fairly recently, almost all randomized solutions have also had (expected) quadratic word complexity. Recent work has broken this barrier [23, 21, 32], but not in asynchronous settings. We present here the first sub-quadratic asynchronous Byzantine Agreement algorithm. Our algorithm is randomized and solves binary BA *with high probability (whp)*, i.e., with probability that tends to 1 as n goes to infinity.

© Shir Cohen, Idit Keidar, and Alexander Spiegelman;
licensed under Creative Commons License CC-BY

We consider a system with a static set of n processes, in the so-called "permissioned" setting, where the ids of all processes are well-known. Our algorithm tolerates f failures for $n \approx 4.5f$ (asymptotically). In addition, we assume a trusted *public key infrastructure* (PKI) that allows us to use *verifiable random functions* (VRFs) [30].

We assume a strong adversary that can adaptively take over processes, whereupon it has full access to their private data. It further sees all messages in the system. But we do limit the adversary in two ways. First, we assume that it is computationally bounded so that we may use the PKI. Second, as proven in [1] for the synchronous model, achieving sub-quadratic complexity is impossible when the adversary can perform after-the-fact removal, meaning that it can delete messages that were sent by correct processes before corrupting these processes. Here, we adapt the no after-the-fact removal assumption to the asynchronous model, and define a *delayed-adaptive adversary* based on causality [27].

We formalize the concept of VRF-based committee sampling as used in Algorand [21, 15], and adapt it to the asynchronous model. In a nutshell, the idea is to use a VRF seeded with each process's private key in order to sample uniformly at random $O(\log n)$ processes for a *committee*, and to have different committees execute different parts of the BA protocol. Each committee is used for sending exactly one protocol message and messages are sent only by committee members, thus reducing the communication cost. Whereas in Algorand's synchronous model a process can be sure it receives messages from all correct committee members by a timeout, in the asynchronous model this is not the case. Rather, processes make progress by waiting for some threshold number of messages. Without committees, this threshold is normally $n - f$ (waiting for more than $n - f$ processes might violate termination). But since committees are randomly sampled, we do not know the committee's exact size or the number of Byzantine processes in it. Thus, adapting committees to this model is somewhat subtle and requires ensuring certain conditions regarding the intersection of subsets of committees. In this paper we identify sufficient conditions on sampling, which ensure safety and liveness with high probability.

Randomized BA algorithms can be seen as if processes toss a random coin at some point during the protocol. While some protocols toss a local coin [9, 12] and require exponential expected time to reach agreement, others use the abstraction of a *shared coin*, which involves communication among processes and results in the same coin toss with some well defined *success rate* [34, 14, 13, 21, 24]. In this work we present an asynchronous shared coin algorithm that uses a VRF and provides a constant success rate with an equal probability for tossing 0 and 1. Unlike previous shared coin implementations, our solution does not require a priori knowledge of the set of participants, which makes it useful in committee-based constructions. We then adapt our coin to work with committees and use it to devise a sub-quadratic BA algorithm.

In summary, this paper presents the first formalization of randomly sampled committees using cryptography in asynchronous settings. Based on this technique, it presents the first sub-quadratic asynchronous shared coin and BA whp algorithms. Our algorithms have expected $\widetilde{O}(n)$ word complexity and $O(1)$ expected time.

Roadmap. The rest of this paper is organized as follows. Section 2 describes the model; Section 3 reviews related work. In Section 4, we present our shared coin algorithm and in Section 5, we formalize committee sampling. Then, in Section 6, we use the coin and the committee sampling to construct a BA whp algorithm. We end with some concluding remarks in Section 7.

2 Model and Preliminaries

We consider a distributed system consisting of a well-known static set Π of n processes and a *delayed-adaptive adversary* (see definition below). The adversary may adaptively corrupt up to $f = (\frac{1}{3} - \epsilon)n$ processes in the course of a run, where $\max\{\frac{3}{8 \ln n}, 0.109\} + \frac{1}{8 \ln n} < \epsilon < \frac{1}{3}$. A corrupted process is *Byzantine*; it may deviate arbitrarily from the protocol. In particular, it may crash, fail to send or receive messages, and send arbitrary messages. As long as a process is not corrupted by the adversary, it is *correct* and follows the protocol.

Delayed-adaptive adversary. In the synchronous model, one defines a *late* adversary [35, 25, 7, 4], which at the beginning of round r, can observe the state of the system at the beginning of round $r-1$. This assumption prevents "after-the-fact" removals of messages sent by processes before being taken over by the adversary [1, 21], as required for achieving a sub-quadratic communication cost. We adapt this assumption to the asynchronous model. Since in asynchronous models the natural order between messages is Lamport's happens-before relation [27], we use the notion of causality instead of "rounds" to define what messages the adversary may observe when scheduling other messages. We denote by $m \to m'$ the fact that m causally precedes m'. The adversary is formally defined as follows:

▶ **Definition 1** (delayed-adaptive adversary). *The delayed-adaptive adversary may adaptively corrupt up to f processes over the course of a run and schedules all messages. The adversary has full access to corrupted processes' private information and can observe all communication, but it can use the contents of a message m sent by a correct process for scheduling a message m' only if $m \to m'$.*

In addition, we assume that once the adversary takes over a process, it cannot "front run" messages that that process had already sent when it was correct, causing the correct messages to be supplanted. Blum et al. [10] achieve this property by using a separate key to encrypt each message, and deleting the secret key immediately thereafter.

Cryptographic tools. We assume a trusted PKI, where private and public keys for the processes are generated before the protocol begins and processes cannot manipulate their public keys. In addition, we assume that the adversary is computationally bounded, meaning that it cannot obtain the private keys of processes unless it corrupts them. Furthermore, we assume that the PKI is in place from the outset. (Recall that we assume a permissioned setting, so the public keys of the n processes are well-known). These assumptions allow us to use verifiable random functions, as we now define.

A *verifiable random function (*VRF*)* is a pseudorandom function that provides a proof of its correct computation [30]. Given a secret key sk, one can evaluate the VRF on any input x and obtain a pseudorandom output y together with a proof π, i.e., $\langle y, \pi \rangle = \text{VRF}_{sk}(x)$. From π and the corresponding public key pk, one can verify that y is correctly computed from x and sk using the function $\text{VRF-Ver}_{pk}(x, \langle y, \pi \rangle)$. Additionally, a VRF needs to satisfy *uniqueness*. More formally, a VRF guarantees the following properties:

- Pseudorandomness: for any x, it is infeasible to distinguish $y = \text{VRF}_{sk}(x)$ from a uniformly random value without access to sk.
- Verifiability: $\text{VRF-Ver}_{pk}(x, \text{VRF}_{sk}(x)) = \mathit{true}$.
- Uniqueness: it is infeasible to find $x, y_1, y_2, \pi_1, \pi_2$ such that $y_1 \neq y_2$ but $\text{VRF-Ver}_{pk}(x, \langle y_1, \pi_1 \rangle) = \text{VRF-Ver}_{pk}(x, \langle y_2, \pi_2 \rangle) = \mathit{true}$.

Efficient constructions for VRFs have been described in the literature [17, 20].

Communication. We assume that every pair of processes is connected via a reliable link. Messages are authenticated in the sense that if a correct process p_i receives a message m indicating that m was sent by a correct process p_j, then m was indeed generated by p_j and sent to p_i. The network is asynchronous, i.e., there is no bound on message delays.

Complexity. We use the following standard complexity notions [3, 31]. While measuring complexity, we allow a *word* to contain a signature, a VRF output, or a value from a finite domain. We define the *duration* of an execution as the longest sequence of messages that are causally related in this execution until all correct processes decide. We measure the expected *word communication complexity* of our protocols as the maximum of the expected total number of words sent by correct processes and the expected *running time* of our protocol as the maximum of the expected duration. In both cases the maximum is computed over all inputs and applicable adversaries and expectation is taken over the random VRF outputs.

3 Related Work

Lower bounds. Our assumptions conform with a number of known bounds. Deterministic consensus is impossible in an asynchronous system if even one process may crash (by FLP [19]) and requires $\Omega(n^2)$ communication even in synchronous systems [18]. As for randomized Byzantine Agreement, Abraham et al. [1] state that disallowing after-the-fact removal is necessary even in synchronous settings for achieving sub-quadratic communication.

Asynchronous BA and shared coin algorithms. The algorithms we present in this paper belong to the family of asynchronous BA algorithms, which sacrifice determinism in order to circumvent FLP. We compare our solutions to existing ones in Table 1.

Ben-Or [9] suggested a protocol with resilience $n > 5f$. This protocol uses a local coin (namely, a local source of randomness) and its expected time complexity is exponential (or constant if $f = O(\sqrt{n})$). Bracha [11] improved the resilience to $n > 3f$ with the same complexity. The complexity can be greatly reduced by replacing the local coin with a shared one with a guaranteed success rate.

Later works presented the shared coin abstraction and used it to solve BA with $O(n^2)$ communication. Rabin [34] was the first to do so, suggesting a protocol with resilience $n > 10f$ and a constant expected number of rounds. Cachin et al. [13] were the first to use a shared coin to solve BA with $O(n^2)$ communication and optimal resilience. Mostefaoui et al. [31] then presented a signature-free BA algorithm with optimal resilience and $O(n^2)$ messages that uses a shared coin abstraction as a black box; the shared coin algorithm we provide in Section 4 can be used to instantiate this protocol. All of the aforementioned algorithms solve binary BA, where the processes' initial values are 0 and 1; a recent work solved multi-valued BA with the same $O(n^2)$ word complexity [3].

BA algorithms also differ in the cryptographic assumptions they make and the cryptographic tools they use. Rabin's coin [34] is based on cryptographic secret sharing [36]. Some later works followed suit, and used cryptographic abstractions such as threshold signatures [3, 13]. Other works forgo cryptography altogether and instead consider a full information model, where there are no restrictions on the adversary's computational power [14, 24]. In this model, the problem is harder, and existing works achieve very low resilience [24] ($n > 400f$) or high communication complexity [14]. In this paper we do use cryptographic primitives. We assume a computationally bounded adversary and rely on the abstraction of a VRF [30]. VRFs were previously used in blockchain protocols [21, 22, 5] and were also used by Micali [29] to construct a shared coin in the synchronous model.

Table 1 Asynchronous Byzantine Agreement algorithms.

Protocol	n >	Adversary	Word complexity	Termination	Safety
Ben-Or [9]	$5f$	adaptive	$O(2^n)$	w.p. 1	✓
Rabin [34]	$10f$	adaptive	$O(n^2)$	w.p. 1	✓
Bracha [11]	$3f$	adaptive	$O(2^n)$	w.p. 1	✓
Cachin et al. [13]	$3f$	adaptive	$O(n^2)$	w.p. 1	✓
King-Saia [24]	$400f$	adaptive	polynomial	whp	✓
MMR [31]	$3f$	adaptive	$O(n^2)$	w.p. 1	✓
Our protocol	$\approx 4.5f$	delayed-adaptive	$\tilde{O}(n)$	whp	whp

Several works [2, 8, 26, 33, 37] solve BA with subquadratic complexity in the so-called optimistic case (or "happy path"), when communication is timely and a correct process is chosen as a "leader". In contrast, we focus on the worst-case asynchronous case.

Committees. We use committees in order to reduce the word complexity and allow each step of the protocol to be executed by only a fraction of the processes. King and Saia used a similar concept and presented the first sub-quadratic BA protocol in the synchronous model [23]. Algorand proposed a synchronous algorithm [21] (and later extended it to eventual synchrony [15]) where committees are sampled randomly using a VRF. Each process executes a local computation to sample itself to a committee, and hence the selection of processes does not require interaction among them. We follow this approach in this paper and adapt the technique to the asynchronous model.

Following initial publication of our work, Blum et al. [10] have also achieved subquadratic BA WHP under an adaptive adversary. Their assumptions are incomparable to ours – while they strengthen the adversary to remove the delayed adaptivity requirement, they also strengthen the trusted setup. Specifically, they use a trusted dealer to a priori determine the committee members, flip the shared coin, and share it among the committee members. In contrast, we use a peer-to-peer protocol to generate randomness, and require delayed adaptivity in order to prevent the adversary from tampering with this randomness. As in our protocol, setup has to occur once and may be used for any number of BA instances.

4 Shared Coin

We describe here an asynchronous protocol for a shared coin with a constant success rate against the delayed-adaptive adversary. We assume that for every $r \in \mathbb{N}$, shared_coin(r) is invoked by all correct processes and that the invocation of shared_coin(r) by some process p is causally independent of its progress at other processes. The definition of a shared coin is given below.

▶ **Definition 2** (Shared Coin). *A shared coin with success rate ρ is a shared object that generates an infinite sequence of binary outputs. For each execution of the procedure shared_coin(r) with $r \in \mathbb{N}$, all correct processes output b with probability at least ρ, for any value $b \in \{0, 1\}$.*

The pseudo-code for our shared coin is presented in Algorithm 1. Our protocol is composed of two phases of messages passing. Each process first samples the VRF with its private key and the protocol's argument in order to generate a random initial value. For brevity, we

denote by VRF_i the VRF with p_i's private key. Using a VRF to generate a random initial value effectively weakens the adversary as Byzantine processes can neither choose their initial values nor equivocate. If a Byzantine process would try to act maliciously, the VRF proof would easily expose it and its message would be ignored.

In each phase of the protocol, each process sends one value to every other process. The receiver validates the received values using the VRF proofs, which are sent along with the values. We omit the proof validation from the code for clarity. After two phases of communication, each process chooses the minimum value it received in the second phase and outputs its least significant bit. We follow the concept of a common core, as presented by Attiya and Welch for the crash failure model [6], and argue that if a core of $f+1$ correct processes hold the global minimum value at the end of phase 1, then by the end of the following phase all processes receive this value. We exploit the ϵ parameter in our resilience definition to bound the number of values held by $f+1$ correct processes. We show that this number is linear in n and hence with a constant positive probability, by the end of the second phase, all correct processes receive the global minimum among the VRF outputs and therefore produce the same output.

Algorithm 1 Protocol shared_coin(r): code for process p_i.

1: Initially *first-set, second-set* $= \emptyset$
2: $v_i \leftarrow VRF_i(r)$
3: send $\langle \text{FIRST}, v_i \rangle$ to all processes

4: **upon receiving** $\langle \text{FIRST}, v_j \rangle$ with valid v_j from p_j **do**
5: **if** $v_j < v_i$ **then** $v_i \leftarrow v_j$
6: *first-set* $\leftarrow$ *first-set* $\cup \{j\}$
7: **when** $|$*first-set*$| = n - f$ for the first time
8: send $\langle \text{SECOND}, v_i \rangle$ to all processes

9: **upon receiving** $\langle \text{SECOND}, v_j \rangle$ with valid v_j from p_j **do**
10: **if** $v_j < v_i$ **then** $v_i \leftarrow v_j$
11: *second-set* $\leftarrow$ *second-set* $\cup \{j\}$
12: **when** $|$*second-set*$| = n - f$ for the first time
13: **return** $LSB(v_i)$

We now prove that the shared coin has a constant success rate. We say that a value v is *common* if at least $f+1$ correct processes receive v by the end of phase 1. Denote by c be the number of different common values. The next two lemmas give a lower bound on c and on the probability that the global minimum among the VRF outputs is common.

▶ **Lemma 3.** *In Algorithm 1,* $c \geq \frac{9\epsilon}{1+6\epsilon} n$.

Proof. In a given run of the algorithm, define a table T with n rows and n columns, where for each correct process p_i and each $0 \leq j \leq n-1$, $T[i,j] = 1$ iff p_i receives $\langle \text{FIRST}, v \rangle$ from p_j before sending the second message in line 8. Each row of a correct process contains exactly $n-f$ ones since it waits for $n-f$ $\langle \text{FIRST}, v \rangle$ messages (line 7). Each row of a faulty process is arbitrarily filled with $n-f$ ones and f zeros. Thus, the total number of ones in the table is $n(n-f)$ and the total number of zeros is nf. Let k be the number of columns with at least $2f+1$ ones. Because each column represents a value and out of the $2f+1$ ones at least

$f+1$ represent correct processes that receive this value, $c \geq k$. Denote by x the number of ones in the remaining columns. Because each column has at most n ones we get:

$$x \geq n(n-f) - kn. \tag{1}$$

And because the remaining columns have at most $2f$ ones:

$$x \leq 2f(n-k). \tag{2}$$

Combining (1), (2) we get:

$$2f(n-k) \geq n(n-f) - kn$$

$$2fn - 2fk \geq n^2 - fn - kn$$

$$(n-2f)k \geq n^2 - 3fn$$

$$k \geq \frac{n(n-3f)}{n-2f}.$$

Because $f = (\frac{1}{3} - \epsilon)n$ we get:

$$c \geq k \geq \frac{n(n-3(\frac{1}{3}-\epsilon)n)}{n-2(\frac{1}{3}-\epsilon)n} = \frac{n(1-1+3\epsilon)}{1-\frac{2}{3}+2\epsilon} = \frac{9\epsilon}{1+6\epsilon}n, \text{as required.}$$ ◀

Let $v_{min} \triangleq \min_{p_i \in \Pi}\{VRF_i(r)\}$. We prove that with a constant probability, it is common.

▶ **Lemma 4.** *$Prob[v_{min}$ is common$] \geq \frac{c}{n} - \frac{1}{3} + \epsilon$.*

Proof. Notice that we assume that the invocation of shared_coin(r) by each process is causally independent of its progress at other processes. Hence, for any two processes p_i, p_j, the messages $\langle \text{FIRST}, v_i \rangle$, $\langle \text{FIRST}, v_j \rangle$ are causally concurrent. Thus, due to our *delayed-adaptive adversary* definition, these messages are scheduled by the adversary regardless of their content, namely their VRF random values. Notice that the adversary can corrupt processes before they initially send their VRF values. Since the adversary cannot predict the VRF outputs of the processes, the probability that the process holding v_{min} is corrupted before sending its FIRST messages is at most $\frac{f}{n}$. The adversary is oblivious to the correct processes' VRF values when it schedules their first phase messages. Therefore, each of them has the same probability to become common. Since at most f common values originate at Byzantine processes, this probability is at least $\frac{c-f}{n-f}$. We conclude that v_{min} is common with probability at least $(1-\frac{f}{n})\frac{c-f}{n-f} = (1-\frac{(\frac{1}{3}-\epsilon)n}{n})\frac{c-(\frac{1}{3}-\epsilon)n}{n-(\frac{1}{3}-\epsilon)n} = (\frac{2}{3}+\epsilon)\frac{c-(\frac{1}{3}-\epsilon)n}{(\frac{2}{3}+\epsilon)n} = \frac{c-(\frac{1}{3}-\epsilon)n}{n} = \frac{c}{n} - \frac{1}{3} + \epsilon$. ◀

We next observe that if v_{min} is common, then it is shared by all processes.

▶ **Lemma 5.** *If v_{min} is common then each correct process holds v_{min} at the end of phase 2.*

Proof. Since v_{min} is common, at least $f+1$ correct processes receive it by the end of phase 1 and update their local values to v_{min}. During the second phase, each correct process hears from $n-f$ processes. This means that it hears from at least one correct process that has updated its value to v_{min} and sent it. ◀

▶ **Lemma 6.** *The coin's success rate is at least $\frac{18\epsilon^2+24\epsilon-1}{6(1+6\epsilon)}$.*

Proof. We bound the probability that all correct processes output $b \in \{0,1\}$ as follows:

$Prob$[all correct processes output b] $\geq Prob$[all correct processes have the same v_i at the end of phase 2 and its LSB is b] $\geq Prob$[all correct processes have $v_i = v_{min}$ at the end of phase 2 and its LSB is b] $= \frac{1}{2} \cdot Prob$[all correct processes have $v_i = v_{min}$ at the end of phase 2] $\overset{\text{Lemma 5}}{\geq} \frac{1}{2} \cdot Prob[v_{min}$ is common] $\overset{\text{Lemma 4}}{\geq} \frac{1}{2}(\frac{c}{n} - \frac{1}{3} + \epsilon) \overset{\text{Lemma 3}}{\geq} \frac{18\epsilon^2+24\epsilon-1}{6(1+6\epsilon)}$. ◀

▶ Remark 7. Notice that for $\epsilon = \frac{1}{3}$ (i.e., $f = 0$) it holds that the coin's success rate is $\frac{1}{2}$ and we get a perfect fair coin.

We have shown a bound on the coin's success rate in terms of ϵ. Since $\epsilon > 0.109$, the coin's success rate is a positive constant. We next prove that the coin ensures liveness.

▶ **Lemma 8.** *If all correct processes invoke Algorithm 1 then all correct processes return.*

Proof. All correct processes send their messages in the first phase. As up to f processes may be faulty, each correct process eventually receives $n - f$ $\langle \text{FIRST}, x\rangle$ messages and sends a message in the second phase. As $n - f$ correct processes send their messages, each correct process eventually receives $n - f$ $\langle \text{SECOND}, x\rangle$ messages and returns. ◀

From Lemma 6 and Lemma 8 we conclude:

▶ **Theorem 9.** *Algorithm 1 implements a shared coin with success rate at least* $\frac{18\epsilon^2+24\epsilon-1}{6(1+6\epsilon)}$.

Complexity. In each shared coin instance all correct processes send two messages to all other processes. Each of these messages contains one VRF output (including a value and a proof), in addition to a constant number of bits that identify the message's type. Therefore, each message's size is a constant number of words and the total word complexity of a shared coin instance is $O(n^2)$.

We have presented a new shared coin in the asynchronous model that uses a VRF. This coin can be incorporated into the Byzantine Agreement algorithm of Mostefaoui et al. [31], to yield an asynchronous binary Byzantine Agreement with resilience $f = (\frac{1}{3} - \epsilon)n$, a word complexity of $O(n^2)$, and $O(1)$ expected time.

5 Committees

5.1 Validated committee sampling

With the aim of reducing the number of messages and achieving sub-quadratic word complexity, it is common to avoid all-to-all communication phases [21, 23]. Instead, a subset of processes is sampled to a committee and only processes elected to the committee send messages. As committees are randomly sampled, preventing the adversary from corrupting their members, each committee member cannot predict the next committee sample and send its message to all other processes. Potentially, if the committee is sufficiently small, this technique allow committee-based protocols to result in sub-quadratic word complexity.

Using VRFs, it is possible to implement *validated committee sampling*, which is a primitive that allows processes to elect committees without communication and later prove their election. It provides every process p_i with a private function $sample_i(s, \lambda)$, which gets a string s and a threshold $1 \leq \lambda \leq n$ and returns a tuple $\langle v_i, \sigma_i\rangle$, where $v_i \in \{true, false\}$ and σ_i is a proof that $v_i = sample_i(s, \lambda)$. If $v_i = true$ we say that p_i is *sampled* to the committee for s and λ. The primitive ensures that p_i is sampled with probability $\frac{\lambda}{n}$. In addition, there is a

public (known to all) function, *committee-val*$(s, \lambda, i, \sigma_i)$, which gets a string s, a threshold λ, a process identification i and a proof σ_i, and returns *true* or *false*.

Consider a string s. For every i, $1 \leq i \leq n$, let $\langle v_i, \sigma_i \rangle$ be the return value of $sample_i(s, \lambda)$. The following is satisfied for every p_i:

- *committee-val*$(s, \lambda, i, \sigma_i) = v_i$.
- If p_i is correct, then it is infeasible for the adversary to compute $sample_i(s, \lambda)$.
- It is infeasible for the adversary to find $\langle v, \sigma \rangle$ s.t. $v \neq v_i$ and *committee-val*$(s, \lambda, i, \sigma) =$ *true*.

We refer to the set of processes sampled to the committee for s and λ as $C(s, \lambda)$. In this paper we set λ to $8 \ln n$. Let d be a parameter of the system such that $\max\{\frac{1}{\lambda}, 0.0362\} < d < \frac{\epsilon}{3} - \frac{1}{3\lambda}$. Our committee-based protocols can no longer wait for $n - f$ processes. Instead, they wait for $W \triangleq \lceil (\frac{2}{3} + 3d)\lambda \rceil$ processes. We show that whp at least W processes will be correct in each committee sample and hence waiting for this number does not compromise liveness. In addition, instead of assuming f Byzantine processes, our committee-based protocols assume that whp the number of Byzantine processes in each committee is at most $B \triangleq \lfloor (\frac{1}{3} - d)\lambda \rfloor$. The following claim is proven in the full version of the paper [16] using Chernoff bounds.

▷ Claim 10. For a string s and $\lambda = const \cdot \ln n$ the following hold with high probability:
(S1) $|C(s, \lambda)| \leq (1 + d)\lambda$.
(S2) $|C(s, \lambda)| \geq (1 - d)\lambda$.
(S3) At least W processes in $C(s, \lambda)$ are correct.
(S4) At most B processes in $C(s, \lambda)$ are Byzantine.

If a protocol uses a constant number of committees, then with high probability, Claim 10 holds for all of them. If, however, a protocol uses a polynomial number of committees then it does not guarantee the properties of this claim. The following corollaries are derived from Claim 10 and are used to ensure the safety and liveness properties of our protocols that use committees (a proof is in the full version). Intuitively, S3 allows the protocol to wait for W messages without forgoing liveness. Property S5 below shows that if two processes wait for sets P_1 and P_2 of this size, then they hear from at least $B + 1$ common processes of which, by S4, at least one is correct.

▶ **Corollary 11** (S5). *Consider $C(s, \lambda)$ for some string s and some $\lambda = const \cdot \ln n$ and two sets $P_1, P_2 \subset C(s, \lambda)$ s.t $|P_1| = |P_2| = W$. Then, $|P_1 \cap P_2| \geq B + 1$.*

The following property is used to show that if $B + 1$ correct processes hold some value, and some correct process waits for messages from W processes, then it hears from at least one correct process that holds this value.

▶ **Corollary 12** (S6). *Consider $C(s, \lambda)$ for some string s and some $\lambda = const \cdot \ln n$ and two sets $P_1, P_2 \subset C(s, \lambda)$ s.t $|P_1| = B + 1$ and $|P_2| = W$. Then, $|P_1 \cap P_2| \geq 1$.*

5.2 WHP Coin

We now employ committee sampling to reduce the word complexity of our shared coin. Our new protocol is called whp_coin. As before, we assume that for every $r \in \mathbb{N}$, the invocation of whp_coin(r) by some process p is causally independent of its progress at other processes. We now define the *WHP coin* abstraction:

▶ **Definition 13** (WHP Coin). *A WHP coin with success rate ρ is a shared object exposing whp_coin(r), $r \in \mathbb{N}$ at each process. If all correct processes invoke whp_coin(r) then, whp (1) all correct processes return, and (2) all of them output the same value b with probability at least ρ, for any value $b \in \{0, 1\}$.*

The whp_coin protocol is presented in Algorithm 2. It samples two committees, one for each communication step. In each step, only the processes that are sampled to the committee send messages. However, since the committee samples are unpredictable, messages are sent to all processes. With committees, processes can no longer wait for $n - f$ messages. Instead they wait for W messages. Since a constant number of committees is sampled in the protocol, Claim 10 holds for all of them and by S3, all processes receive W messages, ensuring liveness.

Algorithm 2 Protocol whp_coin(r): code for process p_i.

1: Initially *first-set*, *second-set* $= \emptyset$, $v_i = \infty$
2: **if** $sample_i(\text{FIRST}, \lambda) = true$ **then**
3: $\quad v_i \leftarrow VRF_i(r)$
4: $\quad$ send $\langle \text{FIRST}, v_i \rangle$ to all processes

5: **upon receiving** $\langle \text{FIRST}, v_j \rangle$ with valid v_j from validly sampled p_j **do**
6: $\quad$ **if** $sample_i(\text{SECOND}, \lambda)$ **then**
7: $\quad\quad$ **if** $v_j < v_i$ **then** $v_i \leftarrow v_j$
8: $\quad\quad$ *first-set* $\leftarrow$ *first-set* $\cup \{j\}$
9: $\quad\quad$ **when** $|$*first-set*$| = W$ for the first time
10: $\quad\quad\quad$ send $\langle \text{SECOND}, v_i \rangle\rangle$ to all processes

11: **upon receiving** $\langle \text{SECOND}, v_j \rangle$ with valid v_j from validly sampled p_j **do**
12: $\quad$ **if** $v_j < v_i$ **then** $v_i \leftarrow v_j$
13: $\quad$ *second-set* $\leftarrow$ *second-set* $\cup \{j\}$
14: $\quad$ **when** $|$*second-set*$| = W$ for the first time
15: $\quad\quad$ **return** $LSB(v_i)$

In the full version of the paper [16] we adapt the coin's correctness proof given in Section 4 to the committee-based protocol, proving the following theorem:

▶ **Theorem 14.** *Algorithm 2 implements a WHP coin with a constant success rate.*

Complexity. In each whp_coin instance using committees all correct processes that are sampled to the two committees (lines 2,6) send messages to all other processes. Each of these messages contains a VRF output (including a value and a proof), a VRF proof of the sender's election to the committe and a constant number of bits that identify the type of message that is sent. Therefore, each message's size is a constant number of words and the total word complexity of a WHP coin instance is $O(nC)$ where C is the number of processes that are sampled to the committees. Since each process is sampled to a committee with probability $\frac{1}{\lambda}$, we get a word complexity of $O(n\lambda) = O(n \log n) = \widetilde{O}(n)$ in expectation.

6 Asynchronous sub-quadratic Byzantine Agreement

We adapt the Byzantine Agreement algorithm of Mostefaoui et al. [31] to work with committees. Our protocol leverages an *approver* abstraction, which we implement in Section 6.1 and then integrate it into a Byzantine Agreement protocol in Section 6.2.

6.1 Approver abstraction

The *approver* abstraction is an adaptation of the Synchronized Binary-Value Broadcast (SBV-broadcast) primitive in [31]. It provides processes with the procedure *approve(v)*, which takes a value v as an input and returns a set of values.

▶ **Assumption 1.** *Correct processes invoke the approver with at most 2 different values.*

Under this assumption, an approver satisfies the following:

▶ **Definition 15** (Approver). *In an approver instance the following properties hold whp:*

- *Validity. If all correct processes invoke approve(v) then the only possible return value of correct processes is $\{v\}$.*
- *Graded Agreement. If a correct process p_i returns $\{v\}$ and another correct process p_j returns $\{w\}$ then $v = w$.*
- *Termination. If all correct processes invoke approve then approve returns with a non-empty set at all of them.*

Our approver uses different committees for different message types, as illustrated in Fig. 1. Importantly, the protocol satisfies the so-called *process replaceability* [21] property, whereby a correct process selected for a committee C broadcasts at most one message in its role as a member of C. Thus, our delayed-adaptive adversary can learn of a process's membership in a committee only after that process ceases to partake in the committee. This allows us to leverage the sampling analysis in the previous section. For clarity of the presentation, we discuss the algorithm here under the assumption that properties S1-S6 hold for all sampled committees. As shown above, these hold whp for each committee, and the algorithm employs a constant number of committees, so they hold for all of them whp.

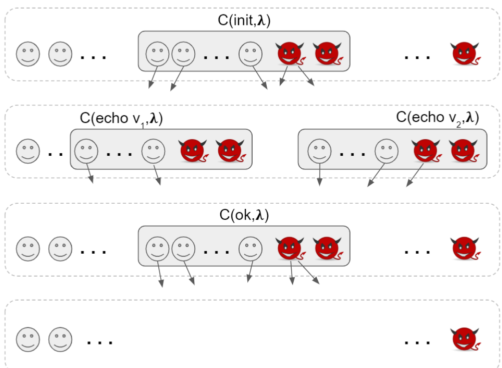


Figure 1 Committees sampled in Algorithm 3.

The approver's pseudo-code appears in Algorithm 3. It consists of three phases – init, echo, and ok. In each phase, committee members broadcast to all processes. Messages are validated to originate from legitimate committee members using the *committee-val* primitive; this validation is omitted from the pseudo-code for clarity. In the first phase, each init committee member broadcasts its input value to all processes.

The role of the echo phase is to "boost" values sent by sufficiently many processes in the init phase, and make sure that *all* correct processes receive them. "Sufficiently many" here means at least $B + 1$, which by S4 includes at least one correct process. Ensuring process replaceability in the echo phase is a bit tricky, since committee members must echo every

value they receive from least $B+1$ processes, and there might be two such values. (Recall that we assume that correct processes invoke the protocol with at most two different values, so there cannot be more than two values that exceed this threshold). To ensure that each committee member broadcasts at most once, we sample a different committee for each value. That is, the value v is part of the string passed to the sample function for this phase.

When a member of the ok committee receives $\langle \text{ECHO}, v\rangle$ messages from W different members of the same echo committee for the first time, it broadcasts an $\langle \text{OK}, v\rangle$ message. Note that the process sends an ok message only for the first value that exceeds this threshold. An $\langle \text{OK}, v\rangle$ message includes, as proof of its validity, W signed $\langle \text{ECHO}, v\rangle$ messages. Again, the proof and its validation are omitted from the pseudo-code for clarity. Once a correct process receives W valid OK messages, it returns the set of values in these messages.

Algorithm 3 Protocol approve(v_i): code for process p_i.

1: **if** $sample_i(\text{INIT}, \lambda) = true$ **then** broadcast $\langle \text{INIT}, v_i\rangle$

2: **upon receiving** $\langle \text{INIT}, v\rangle$ from $B+1$ different processes **do**
3:     **if** $sample_i(\langle \text{ECHO}, v\rangle, \lambda) = true$ **then** broadcast $\langle \text{ECHO}, v\rangle$

4: **upon receiving** $\langle \text{ECHO}, v\rangle$ from W different processes **do**
5:     **if** $sample_i(\text{OK}, \lambda) = true \wedge$ haven't sent any $\langle \text{OK}, *\rangle$ message **then**
6:         broadcast $\langle \text{OK}, v\rangle$

7: **upon receiving** $\langle \text{OK}, *\rangle$ from W different processes **do**
8:     return the set of values received in these messages

We next prove that Algorithm 3 implements an approver. The following three lemmas are stated here and their proofs appear in the full version [16]:

▶ **Lemma 16** (Validity). *If all correct processes invoke* approve(v) *then the only possible return value of correct processes is* $\{v\}$ *whp.*

▶ **Lemma 17** (Graded Agreement). *If a correct process* p_i *returns* $\{v\}$ *and another correct process* p_j *returns* $\{w\}$ *then* $v = w$ *whp.*

▶ **Lemma 18** (Termination). *If all correct processes invoke approve then at every correct process approve returns with a non-empty set whp.*

From Lemmas 16,17,18, we conclude the following theorem:

▶ **Theorem 19.** *Algorithm 3 implements an approver.*

Complexity. In each approver instance correct processes that are sampled to the four committees (lines 1,3,5) send messages to all other processes. The committee size is $O(\lambda) = O(\log n)$ whp. Messages contain values, VRF proofs of the sender's election to the committee, signatures of $O(\lambda)$ committee members, and a constant number of bits that identify the type of message that is sent. Therefore, each message's size is at most $O(\lambda)$ words and the total word complexity of a shared coin instance is $O(n\lambda^2) = O(n \log^2 n) = \widetilde{O}(n)$ in expectation. The λ^2 appears in the expression due to the signatures of $O(\lambda)$ processes sent along the ok messages.

6.2 Byzantine Agreement WHP

Our next step is solving Byzantine Agreement whp, formally defined as follows:

▶ **Definition 20** (Byzantine Agreement WHP). *In Byzantine Agreement WHP, each correct process $p_i \in \Pi$ proposes a binary input value v_i and decide on an output value $decision_i$ s.t. with high probability the following properties hold:*

- *Validity. If all correct processes propose the same value v, then any correct process that decides, decides v.*
- *Agreement. No two correct processes decide differently.*
- *Termination. Every correct process eventually decides.*

We present the pseudo-code for our algorithm in Algorithm 4. Our protocol executes in rounds. Each round consists of two approver invocations and one call to the WHP coin. Again, we discuss the algorithm assuming S1-S6 hold. We will argue that the algorithm decides in a constant number of rounds whp, and so these properties hold for all the committees it uses. The local variable est_i holds p_i's current estimate of the decision value. The variable $decision_i$ holds p_i's irrevocable decision. It is initialized to $\perp$ and set to a value in $\{0, 1\}$ at most once. Every process p_i begins by setting est_i to hold its initial value. At the beginning of each round processes execute the approver with their est values. If they return a singleton $\{v\}$, they choose to invoke the next approver with v as their proposal and otherwise they invoke the next approver with $\perp$. By the approver's graded agreement property, different processes do not return different singletons. Thus, at most two different values ($\perp$ and one in $\{0, 1\}$) are given as an input by correct processes to the next approver, satisfying Assumption 1.

At this point, after all correct processes have chosen their proposals, they all invoke the WHP coin in line 8 in order to select a fall-back value. Notice that executing the WHP coin protocol after proposals have been set prevents the adversary from biasing proposals based on the coin flip. Then, in in line 9, all processes invoke the approver with their proposals. If a process does not receive $\perp$ in its return set, it can safely decide the value it received. It does so by updating its *decision* variable in line 13. If it receives some value other than $\perp$ it adopts it to be its estimated value (line 18), whereas if it receives only $\perp$, it adopts the coin flip (line 16). If all processes receive $\perp$ in line 4 then the probability that they all adopt the same value is at least 2ρ, where ρ is the coin's success rate. If some processes receive v, then the probability that all the processes that adopt the coin flip adopt v is at least ρ. With high probability, after a constant number of rounds, all correct processes have the same estimated value. By validity of the approver, once they all have common estimate, they decide upon it within 1 round.

We now prove our main theorem:

▶ **Theorem 21.** *Algorithm 4 when using an approver (Definition 15) and a WHP coin (Definition 13) solves Byzantine Agreement whp (Definition 20).*

We first show that Algorithm 4 satisfies the approver and WHP coin primitives' assumptions whp. Proving this allows us to use their properties while proving Theorem 21.

▶ **Lemma 22.** *For every round r of Algorithm 4 the following hold:*

1. *All correct processes invoke approve with at most 2 different values.*
2. *The invocation of $whp_coin(r)$ by a correct process p is causally independent of its progress at other processes.*

Algorithm 4 Protocol Byzantine Agreement(v_i): code for process p_i.

```
1:  est_i ← v_i
2:  decision_i ← ⊥

3:  for r = 0, 1, ... do
4:      vals ← approve(est_i)
5:      if vals = {v} for some v then
6:          propose_i ← v
7:      otherwise propose_i ← ⊥

8:      c ← whp_coin(r)
9:      props ← approve(propose_i)
10:     if props = {v} for some v ≠ ⊥ then
11:         est_i ← v
12:         if decision_i = ⊥ then
13:             decision_i ← v
14:     else
15:         if props = {⊥} then
16:             est_i ← c
17:         else                          ▷ props = {v, ⊥}
18:             est_i ← v
```

Proof. 1. It is easy to see, by induction on the number of rounds, that since the processes' inputs are binary and we use a binary coin, the *est* of all processes is in $\{0, 1\}$ at the beginning of each round. Hence, the approver in line 4 is invoked with at most two different values. Due to its graded agreement property, all processes that update their propose to $v \neq \perp$ in line 6 update it to the same value whp. Thus, whp, in line 9 approver is invoked with either v or $\perp$.

2. Correct processes call *whp_coin*(r) without waiting for indication that other processes have done so. ◀

The following lemma shows that for any given round of the algorithm, (1) whp all processes complete this round, and (2) with a constant probability, they all have the same estimate value by its end. Its proof is in the full version of the paper.

▶ **Lemma 23.** *If all correct processes begin round r of Algorithm 4 then whp:*

1. *All correct processes complete round r, i.e. they're not blocked during round r.*
2. *With probability greater than ρ, where ρ is the success rate of the WHP coin, all correct processes have the same est value at the end of round r.*

The following lemmas indicate that the Byzantine Agreement whp properties are satisfied, which completes the proof of Theorem 21.

▶ **Lemma 24.** *(Validity) If at the beginning of round r of Algorithm 4 all correct processes have the same estimate value v, then whp any correct process that has not decided before decides v in round r.*

Proof. If all correct processes start round r then by Lemma 23 they all complete round r. Since they all being with the same estimate value v, they all execute approve(v) in line 4. Hence, by approver's validity and termination, whp they all return the non-empty set $\{v\}$ and update their *propose* values to v. Then, they all execute approve(v) for the second time in line 9, and due to the same reason, they all return $\{v\}$ whp. Any correct process that has not decided before decides v in line 13. ◀

▶ **Lemma 25.** *(Termination) Every correct process decides whp.*

Proof. By Lemma 23, for every round r of Algorithm 4, with probability greater than ρ, where ρ is the success rate of the WHP coin, all correct processes have the same *est* value at the end of r whp. Hence, by Lemma 24, with probability greater than ρ, all correct processes

decide by round $r+1$ whp. It follows that the expected number of rounds until all processes decide is bounded by $\frac{1}{\rho}$, which is constant. Thus, by Chebyshev's inequality, whp all correct processes decide within a constant number of rounds. ◀

▶ **Lemma 26.** *(Agreement) No two correct processes decide different values whp.*

Proof. Let r be the first round in which some process p_i decides on some value $v \in \{0,1\}$. Thus, p_i's invocation to approver in line 9 of round r returns $\{v\}$. If another correct process p_j decides w in round r then its approver call in line 9 of round r returns $\{w\}$. By approver's graded agreement, $v = w$ whp. Consider a correct process p_k that does not decide in round r. By the definition of r, p_k hasn't decided in any round $r' < r$. By approver's graded agreement, whp, p_k returns $\{v, \bot\}$ in line 9 of round r, and p_k updates its est_k value to v in line 18. It follows that whp all correct processes have v as their estimate value at the beginning of round $r+1$. By Lemma 24, every correct process that has not decided in round r decides v in round $r+1$ whp. ◀

Complexity. In each round of the protocol, all correct processes invoke two approver calls and one WHP coin instance. Due to the constant success rate of the WHP coin, the expected number of rounds before all correct processes decide is constant. Thus, due to the word complexity of the WHP coin and approver, the expected word complexity is $O(n \log^2 n) = \widetilde{O}(n)$ and the time complexity is $O(1)$ in expectation.

7 Conclusions and Future Directions

We have presented the first sub-quadratic asynchronous Byzantine Agreement algorithm. To construct the algorithm, we introduced two techniques. First, we presented a shared coin algorithm that requires a trusted PKI and uses VRFs. Second, we formalized VRF-based committee sampling in the asynchronous model for the first time.

Our algorithm solves Byzantine Agreement with high probability. It would be interesting to understand whether some of the problem's properties can be satisfied with probability 1, while keeping the sub-quadratic communication cost. In addition, in order to achieve the constant success rate of the coin and guarantee the committees' properties, we bounded ϵ from below by a constant. This bound prevented us from achieving optimal resilience. The question whether it is possible to relax this bound to allow better resilience remains open.

References

1. Ittai Abraham, TH Hubert Chan, Danny Dolev, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. Communication complexity of byzantine agreement, revisited. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 317–326, 2019.
2. Ittai Abraham, Guy Golan-Gueta, and Dahlia Malkhi. Hot-stuff the linear, optimal-resilience, one-message bft devil. *CoRR*, abs/1803.05069, 2018.
3. Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. Asymptotically optimal validated asynchronous byzantine agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 337–346, 2019.
4. Mohamad Ahmadi, Abdolhamid Ghodselahi, Fabian Kuhn, and Anisur Rahaman Molla. The cost of global broadcast in dynamic radio networks. *Theoretical Computer Science*, 806:363–387, 2020.
5. Avi Asayag, Gad Cohen, Ido Grayevsky, Maya Leshkowitz, Ori Rottenstreich, Ronen Tamari, and David Yakira. Helix: A scalable and fair consensus algorithm resistant to ordering manipulation. *IACR Cryptology ePrint Archive*, 2018:863, 2018.

6 Hagit Attiya and Jennifer Welch. *Distributed computing: fundamentals, simulations, and advanced topics*, volume 19. John Wiley & Sons, 2004.
7 Baruch Awerbuch and Christian Scheideler. A denial-of-service resistant dht. In *International Symposium on Distributed Computing*, pages 33–47. Springer, 2007.
8 Mathieu Baudet, Avery Ching, Andrey Chursin, George Danezis, François Garillot, Zekun Li, Dahlia Malkhi, Oded Naor, Dmitri Perelman, and Alberto Sonnino. State machine replication in the libra blockchain. *The Libra Assn., Tech. Rep*, 2019.
9 Michael Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 27–30. ACM, 1983.
10 Erica Blum, Jonathan Katz, Chen-Da Liu-Zhang, and Julian Loss. Asynchronous byzantine agreement with subquadratic communication. Cryptology ePrint Archive, Report 2020/851, 2020.
11 Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.
12 Gabriel Bracha and Sam Toueg. Resilient consensus protocols. In *Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 12–26. ACM, 1983.
13 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.
14 Ran Canetti and Tal Rabin. Fast asynchronous byzantine agreement with optimal resilience. In *STOC*, volume 93, pages 42–51. Citeseer, 1993.
15 Jing Chen, Sergey Gorbunov, Silvio Micali, and Georgios Vlachos. Algorand agreement: Super fast and partition resilient byzantine agreement. *IACR Cryptology ePrint Archive*, 2018:377, 2018.
16 Shir Cohen, Idit Keidar, and Alexander Spiegelman. Not a coincidence: Sub-quadratic asynchronous byzantine agreement whp. *arXiv preprint arXiv:2002.06545*, 2020.
17 Yevgeniy Dodis and Aleksandr Yampolskiy. A verifiable random function with short proofs and keys. In *International Workshop on Public Key Cryptography*, pages 416–431. Springer, 2005.
18 Danny Dolev and Rüdiger Reischuk. Bounds on information exchange for byzantine agreement. *J. ACM*, 32(1):191–204, January 1985.
19 Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.
20 Matthew Franklin and Haibin Zhang. Unique ring signatures: A practical construction. In *International Conference on Financial Cryptography and Data Security*, pages 162–170. Springer, 2013.
21 Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 51–68, 2017.
22 Timo Hanke, Mahnush Movahedi, and Dominic Williams. Dfinity technology overview series, consensus system. *arXiv preprint arXiv:1805.04548*, 2018.
23 Valerie King and Jared Saia. Breaking the $O(n^2)$ bit barrier: scalable byzantine agreement with an adaptive adversary. *Journal of the ACM (JACM)*, 58(4):1–24, 2011.
24 Valerie King and Jared Saia. Byzantine agreement in polynomial expected time. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, pages 401–410. ACM, 2013.
25 Marek Klonowski, Dariusz R Kowalski, and Jarosław Mirek. Ordered and delayed adversaries and how to work against them on a shared channel. *Distributed Computing*, 32(5):379–403, 2019.
26 Jae Kwon. Tendermint: Consensus without mining. *Draft v. 0.6, fall*, 1(11), 2014.
27 Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

28 Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

29 Silvio Micali. Very simple and efficient byzantine agreement. In Christos H. Papadimitriou, editor, *8th Innovations in Theoretical Computer Science Conference, ITCS 2017, January 9-11, 2017, Berkeley, CA, USA*, volume 67 of *LIPIcs*, pages 6:1–6:1. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.ITCS.2017.6.

30 Silvio Micali, Michael Rabin, and Salil Vadhan. Verifiable random functions. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pages 120–130. IEEE, 1999.

31 Achour Mostéfaoui, Hamouma Moumen, and Michel Raynal. Signature-free asynchronous binary byzantine consensus with $t < n/3$, $O(n^2)$ messages, and $O(1)$ expected time. *Journal of the ACM (JACM)*, 62(4):31, 2015.

32 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2009.

33 Oded Naor, Mathieu Baudet, Dahlia Malkhi, and Alexander Spiegelman. Cogsworth: Byzantine view synchronization. *arXiv preprint arXiv:1909.05204*, 2019.

34 Michael O Rabin. Randomized byzantine generals. In *24th Annual Symposium on Foundations of Computer Science (sfcs 1983)*, pages 403–409. IEEE, 1983.

35 Peter Robinson, Christian Scheideler, and Alexander Setzer. Breaking the $\Omega(\sqrt{n})$ barrier: Fast consensus under a late adversary. In *30th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2018*, pages 173–182. ACM New York, 2018.

36 Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

37 Alexander Spiegelman. In search for a linear byzantine agreement. *arXiv preprint arXiv:2002.06993*, 2020.

Expected Linear Round Synchronization: The Missing Link for Linear Byzantine SMR

Oded Naor
Technion – Israel Institute of Technology, Haifa, Israel

Idit Keidar
Technion – Israel Institute of Technology, Haifa, Israel

Abstract

State Machine Replication (SMR) solutions often divide time into rounds, with a designated leader driving decisions in each round. Progress is guaranteed once all correct processes *synchronize* to the same round, and the leader of that round is correct. Recently suggested Byzantine SMR solutions such as HotStuff, Tendermint, and LibraBFT achieve progress with a linear message complexity and a constant time complexity once such round synchronization occurs. But round synchronization itself incurs an additional cost. By Dolev and Reischuk's lower bound, any deterministic solution must have $\Omega(n^2)$ communication complexity. Yet the question of randomized round synchronization with an expected linear message complexity remained open.

We present an algorithm that, for the first time, achieves round synchronization with expected linear message complexity and expected constant latency. Existing protocols can use our round synchronization algorithm to solve Byzantine SMR with the same asymptotic performance.

2012 ACM Subject Classification Computing methodologies → Distributed algorithms

Keywords and phrases Distributed Systems, State Machine Replication

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.26

Related Version A full version of the paper is available at [33], `https://arxiv.org/abs/2002.07539`.

1 Introduction

Byzantine *State Machine Replication (SMR)* has received a lot of attention in recent years due to the increasing demand for robust and scalable systems. In order to tolerate periods of high load or even denial-of-service attacks, practical solutions commonly assume the *eventual synchrony* model [20], meaning that they guarantee consistency despite asynchrony and make progress during periods when the network is synchronous. Examples of such systems include PBFT [16], SBFT [25], LibraBFT [4], HotStuff [37], Zyzzyva [29], Tendermint [13], and many more. Eventually synchronous SMR solutions typically iterate through a sequence of *rounds*, (also called views), wherein a designated *leader* process tries to drive all correct processes to consensus. The main complexity of such algorithms arises whenever a new round begins and its (new) leader collects information about possible consensus decisions in previous rounds.

When using such protocols, it is common to constantly advance in rounds with a rotating leader [16, 37, 4]; this is because when the leader is faulty, it is possible for some processes to perceive progress while others made no progress.

In the last couple of years, there has been a race to improve the performance of Byzantine SMR. Recent algorithms such as Tendermint [13], Casper [14], HotStuff [37], and LibraBFT [4] allow rounds to advance (and leaders to be replaced) with a constant time complexity and a linear message complexity. Thus, even if every consensus instance is led by a different leader, the message complexity for each decision remains linear. Nevertheless, the linear message complexity is achieved only *after* all correct processes synchronize to execute the same round of the protocol (provided that that round's leader is correct). And such *round synchronization* has a cost of its own. In Tendermint, round advancement is gossiped

© Oded Naor and Idit Keidar;
licensed under Creative Commons License CC-BY
34th International Symposium on Distributed Computing (DISC 2020).
Editor: Hagit Attiya; Article No. 26; pp. 26:1–26:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

throughout the system, entailing an expected $O(n \log n)$ message complexity with expected $O(\log n)$ latency. In HotStuff, it is delegated to a separate round synchronization module called *PaceMaker*, whose implementation is left unspecified. And in LibraBFT, this module is implemented with quadratic message complexity, which in fact matches Dolev and Reischuk's $\Omega(n^2)$ communication complexity lower bound [19] on deterministic Byzantine consensus. Later work on Cogsworth [32] implemented a randomized PaceMaker with expected constant latency and linear message complexity under benign failures, but with expected quadratic message complexity in the Byzantine case.

In this work we present a new round synchronization algorithm that achieves expected constant time complexity and expected linear communication complexity even in the presence of Byzantine processes. Specifically, under an oblivious adversary, we guarantee these bounds on the expected time/message cost until all processes synchronize to the same round from an *arbitrary* state of the protocol. Under a strong adversary, we achieve the same bounds but on the *average* expected time and message cost until round synchronization over all states occurring in an infinite run of the protocol. To this end, we decompose the round synchronization module into a *synchronizer* abstraction and two local functions. The synchronizer abstraction captures the essence of the distributed coordination required in order to synchronize processes to the same round.

Like previous works [32, 37, 4], the main technique used in our algorithm to lower the message complexity is a relay-based message distribution with threshold signatures. Instead of broadcasting messages all-to-all, our algorithm sends each message to a designated *relay*. The relay aggregates messages from multiple processes, and when a certain threshold is met, it combines them into a threshold signature, which it sends to all the processes. Note that a threshold signature's size is the same as the original signature sizes, i.e., remains constant as the number of processes grows. This leads to linear communication complexity per message. The challenge is that the relay can be Byzantine and, for example, send the aggregated message only to a subset of the processes. Another challenge arises when some correct process advances to a new round while others lag behind. We introduce a relay-based linear-complexity helping mechanism to allow lagging processes to catch up with faster ones without all-to-all broadcast.

In summary, the main contribution of this paper is providing an algorithm that for the first time reduces the expected message complexity of Byzantine SMR in the presence of Byzantine faults to linear, while maintaining expected constant latency. The rest of the paper is structured as follows: §2 describes the model; §3 formally defines the round synchronization problem and our performance metrics; §4 explains our decomposition of round synchronization into a synchronizer abstraction and local functions, and proves that this decomposition solves the round synchronization problem; §5 presents our new synchronizer algorithm and proves its expected linear message complexity, expected constant latency, and correctness; §6 gives related work and §7 concludes the paper. Some formal proofs are deferred to the full version [33].

2 Model

Our model consists of a set $\Pi = \{\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_n\}$ of n processes. Every two processes in Π have a bidirectional, reliable, and authenticated link between them, i.e., every process can send a message to another process that will eventually arrive and the recipient can verify the sender's identity. We use the term *broadcast* to indicate sending a message to all processes.

We follow the eventually synchronous model [20] in which there is no global clock, and every execution is divided into two periods: first, an unbounded period of asynchrony; and then, a period of synchrony, where messages arrive within a bounded time, δ. The second period begins at a moment called the *Global Stabilization Time (GST)*. Messages sent before GST arrive by GST $+\,\delta$. We assume that after GST, processes can correctly estimate such an upper bound δ on communication latency and also measure time locally, but this does not imply a global clock. We consider a failure model where $f < n/3$ processes may be *faulty*, or *Byzantine* and act arbitrarily.

We assume a shared source of randomness, $\mathcal{R}$, that is used to derive a function $\textsc{Relay}(r,k)\colon \mathbb{N} \times \{1,\ldots,f+1\} \mapsto \Pi$. This function is used to select for each round r the k-th process that will act as a relay. The relay function satisfies the following properties:

R1 $f+1$ different relays for each round:

$$\forall r \in \mathbb{N}, \forall 1 \le i < j \le f+1\colon \textsc{Relay}(r,i) \ne \textsc{Relay}(r,j).$$

R2 Random relay selection, while ensuring $f+1$ different relays for each round:

$$\forall r \in \mathbb{N}, \forall 1 \le k \le f+1, \forall \mathcal{P}_1, \mathcal{P}_2 \in \Pi \setminus \bigcup_{i=1}^{k-1} \{\textsc{Relay}(r,i)\} :$$
$$\Pr\left[\textsc{Relay}(r,k) = \mathcal{P}_1\right] = \Pr\left[\textsc{Relay}(r,k) = \mathcal{P}_2\right].$$

Note that R2 implies that the first relay is continuously rotated throughout the run, i.e., $\forall r \in \mathbb{N}\colon \bigcup_{i=r}^{\infty} \textsc{Relay}(i,1) = \Pi$. Generating secure randomness as assumed by our protocol has been studied in the literature, e.g., [8, 17, 31, 2], and is beyond the scope of this paper.

For clarity of the algorithm's presentation, we assume that the adversary is a *static oblivious adversary* [10, 5, 21], i.e., has no knowledge of the randomness $\mathcal{R}$. This assumption is required for the worst-case performance bounds as defined in §3.2, and can be relaxed to a *strong static adversary* if we only wish to prove an average-case bound. We discuss this in §5.5 below.

Like previous linear-complexity BFT algorithms [4, 13, 32, 37], we use a cryptographic signing scheme, a public key infrastructure (PKI) to validate signatures, and a threshold signing scheme [9, 15, 36]. The threshold signing scheme is used in order to create a compact-sized signature of K-of-N processes as in other consensus protocols [15]. Usually $K = f+1$ or $K = 2f+1$. The size of a threshold signature is constant and does depend on K or N. We assume that the adversary is polynomial-time bounded, i.e., the probability that it will break the cryptographic assumptions in this paper (e.g., the cryptographic signatures, threshold signatures, etc.) is negligible.

3 Problem Definition - Round Synchronization

We start by specifying the round synchronization problem in §3.1, then discuss performance metrics in §3.2, and conclude by describing how to use a round synchronization module to solve consensus in §3.3.

3.1 Specification

We define a long-lived task of *round synchronization*, parameterized by the desired round duration Δ. It has a single output signal at process $\mathcal{P}_i$, $\mathsf{round_leader}_i(r,\mathcal{P})$, $r \in \mathbb{N}, \mathcal{P} \in \Pi$, indicating to $\mathcal{P}_i$ to enter round r of which $\mathcal{P}$ is the leader. We say that a process $\mathcal{P}_i$ is *in*

round r between the time t when $\mathsf{round_leader}_i(r, \cdot)$ occurs and the next $\mathsf{round_leader}_i(r', \cdot)$ event after t. If no such event occurs, $\mathcal{P}_i$ remains in round r from t onward. The goal of round synchronization is to reach a *synchronization time*, defined as follows:

▶ **Definition 3.1** (Synchronization time). *Time t_s is a* synchronization time *if all correct processes are in the same round r from t_s to at least $t_s + \Delta$, and r has a correct leader.*

A round synchronization module satisfies two properties. The first ensures that in every round all the correct processes have the same leader.

▶ **Property 1** (Leader agreement). *For any two correct processes $\mathcal{P}_i, \mathcal{P}_{i'}$ if $\mathsf{round_leader}_i(r, \mathcal{P}_j)$ and $\mathsf{round_leader}_{i'}(r, \mathcal{P}_{j'})$ occur, then $\mathcal{P}_j = \mathcal{P}_{j'}$.*

The second property ensures that synchronization times eventually occur. Formally:

▶ **Property 2** (Eventual round synchronization). *For every time t in a run, there exists a synchronization time after t.*

3.2 Performance Metrics

For an oblivious adversary, we measure the maximum expected performance after GST under all possible adversary behaviors and protocol states, where the expectation is taken over random outputs of our randomness source $\mathcal{R}$, which drives the relay function. In more detail, let S be the set of all reachable states of a round synchronization algorithm, and let $\mathcal{A}$ be the set of all possible adversary behaviors after GST. This includes selecting up to f processes to corrupt and scheduling all message deliveries within at most δ time. For a state $s \in S$, and adversary behavior $a \in \mathcal{A}$, let $RS(s, a, \pi)$ be the time from when s occurs until the next synchronization time in a run extending s with adversary behavior a and the relay function derived from the random bits $\pi \in \mathcal{R}$.

The *worst-case expected latency* of the round synchronization module is defined as

$$\max_{\substack{s \in S \\ a \in \mathcal{A}}} \left\{ \mathop{\mathbb{E}}_{\pi \in \mathcal{R}} \left[RS(s, a, \pi) \right] \right\}.$$

Similarly, to define message complexity let $M(s, a, r)$ be the total number of messages correct processes send from state s until the next synchronization time in a run extending s with adversary $a \in \mathcal{A}$ and relay output $\pi \in \mathcal{R}$. The *worst-case message complexity* is defined as

$$\max_{\substack{s \in S \\ a \in \mathcal{A}}} \left\{ \mathop{\mathbb{E}}_{\pi \in \mathcal{R}} \left[M\left(s, a, \pi\right) \right] \right\}.$$

For brevity, in the rest of this paper, we omit the parameters s, a, π, and simply bound the expected latency or message cost over all reachable states and adversary behaviors.

3.3 Using Round Synchronization to Solve Consensus

In HotStuff [37], Theorem 4 states the following in regards to reaching a decision in the consensus protocol:

> "After GST, there exists a bounded time period T_f such that if all correct replicas remain in view v during T_f and the leader for view v is correct, then a decision is reached."

The round synchronization module satisfies exactly the conditions of the theorem, i.e., an eventual round that all the correct processes are in at the same time for at least $\Delta = T_f$, and the leader of that round is correct.

Given a round synchronization module with expected linear message complexity and expected constant latency, HotStuff solves consensus in the same expected asymptotic message complexity and latency as the round synchronization module. In addition, HotStuff also uses the same cryptographic primitives (namely threshold signatures) as we use in this paper, incurring similar computational costs.

Note that, in general, processes know neither whether their leader is correct nor whether all correct processes are in the same round as them. Indeed, it is possible for a set of $f+1$ correct processes (and f Byzantine ones) to make progress in a round with a Byzantine leader, while f correct processes are stuck behind. In an SMR algorithm where the processes communicate only with the leader of each round and do not broadcast decisions to all processes, this scenario is indistinguishable from one where the leader is correct and all correct processes make progress. Therefore, to ensure the condition required by HotStuff (and captured by Property 2), we continuously advance in rounds and change leaders, regardless of the observed progress made in the consensus protocol utilizing the round synchronization module.

4 Round Synchronization Decomposition

We build the round synchronization module using a *synchronizer* abstraction and two local modules. The synchronizer captures the necessary distributed coordination among the processes. The abstraction's properties appear in §4.1, and a round synchronization module using this abstraction is given in §4.2. The latter consists of a *timer* function that paces the synchronizer and a *leader* function that outputs the leader and round to the application. This decomposition is illustrated in Figure 1.

4.1 Synchronizer

We define a *synchronizer* abstraction to be a long-lived task with an API that includes an $\mathit{advance}_i()$ input and a $\mathit{new_round}_i(r)$ output signal, where $r \in \mathbb{N}$.

In a similar way to the round synchronization module, we say that process $\mathcal{P}_i$ *enters round* r when $\mathsf{new_round}_i(r)$ occurs. We say *process* $\mathcal{P}$ *is in round* r during the time interval that starts when $\mathcal{P}$ enters round r and ends when it next enters another round. If the process does not enter a new round, then it remains indefinitely in r. We denote by $r_max(t)$ the maximum round a correct process is in at time t.

We define four properties a synchronizer algorithm should guarantee. The first ensures that rounds are monotonically increasing. Formally:

▶ **Property 3** (Monotonically increasing rounds). *For each correct process $\mathcal{P}_i$, if $\mathit{new_round}_i(r')$ occurs after $\mathit{new_round}_i(r)$, then $r' > r$.*

The next property is the validity of new rounds.

▶ **Property 4** (Validity). *If a correct process signals $\mathit{new_round}(r)$ then some correct process called $\mathit{advance}()$ while in round $r-1$.*

Next, we define the two liveness properties. Informally, the first ensures the stabilization of at least $f+1$ correct processes to the same maximum round, and the second ensures progress after the stabilization.

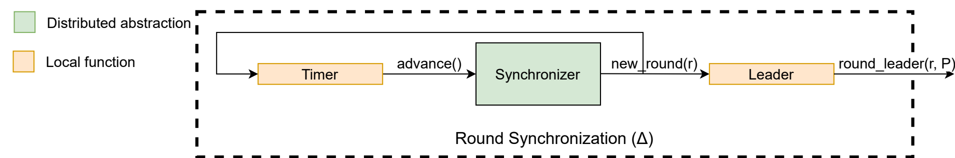


Figure 1 Round synchronization using the synchronizer abstraction.

Algorithm 1 Round synchronization using the synchronizer abstraction.

```
1 Timer:
2   after c1 + Δ from last new_round(r) signal: // c1 is defined in S2
3     advance()
4 Leader:
5   on new_round(r) signal:
6     round_leader(r, RELAY(r, 1))
```

▶ **Property 5** (Stabilization). *For any t during the run, let t_0 be the first time when a correct process enters round $r_max(t)$. If no correct process enters any round $r > r_max(t)$, then:*

S1 *From some time t_1 onward, at least $f + 1$ correct processes are in round $r_max(t)$.*

S2 *If $t_0 \geq GST$ and $\text{Relay}(r_max(t), 1)$ is correct, then from some time t_2 onward all the correct processes enter $r_max(t)$ and $t_2 - t_0 \leq c_1$ for some constant c_1.*

Although Property 5 is primed on no correct processes *ever* entering rounds higher than $r_max(t)$, we observe that S2 holds as long as no process enters rounds higher than $r_max(t)$ by $t_0 + c_1$ because any such run is indistinguishable to all processes until time $t_0 + c_1$ from a run where they never enter a higher round at all. Formally:

▶ **Observation 1.** *Assume Property 5 holds, then for any t during the run, let $t_0 \geq GST$ be the first time when a correct process enters round $r_max(t)$. If no correct process enters any round $r > r_max(t)$ by $t_0 + c_1$ for some constant c_1 and $\text{Relay}(r_max(t), 1)$ is correct, then all correct processes enter round $r_max(t)$ by $t_0 + c_1$.*

The next property ensures progress.

▶ **Property 6** (Progress). *For any t during the run, if $f + 1$ correct processes in round $r_max(t)$ call advance() by t_0, and no correct process calls advance() while in any round $r > r_max(t)$ then:*

P1 *From some time t_1 onward, there is at least one correct process in $r_max(t) + 1$.*

P2 *If $t_0 \geq GST$ and $\text{Relay}(r_max(t), 1)$ is correct, then from some time t_2 onward all the correct processes enter $r_max(t) + 1$ and $t_2 - t_0 \leq c_2$ for some constant c_2.*

Property P2 is not required for round synchronization, but it gives a bound on performance.

4.2 From Synchronizer to Round Synchronization

We now describe how to use the synchronizer abstraction to implement round synchronization. The implementation uses two local functions: a *timer* function that paces a process' advance() calls, and a *leader* function that maps a round to a leader using the Relay function. This construction is illustrated in Figure 1, and specified in Alg. 1. When one module invokes a function in another, we refer to this as a *signal*, e.g., the timer signals advance() to the synchronizer.

We prove that this construction provides round synchronization. Let $t^0 = \text{GST}$ and $\forall \ell \geq 1$ let t^ℓ be the first time after $t^{\ell-1}$ that a correct process enters a new maximum round. We prove the following lemma:

▶ **Lemma 4.1.** *In an infinite run of Alg. 1, t^ℓ eventually occurs for any $\ell \geq 0$.*

Proof. We prove this by induction on ℓ. Based on the model, the base step of the induction, $t^0 = \text{GST}$ eventually occurs.

Next, assume that t^ℓ occurs during the run. If $t^{\ell+1}$ occurs, then we are done.

Assume by contradiction that $t^{\ell+1}$ does not occur, i.e., by the induction hypothesis some correct process entered $r_max(t^\ell)$ but no correct process enters any round $r > r_max(t^\ell)$. By S1, eventually at least $f+1$ correct processes enter $r_max(t^\ell)$. Denote this set of processes by P. The timer function ensures that eventually every process in P calls advance(), so there are at least $f+1$ correct processes in $r_max(t^\ell)$ that call advance(). By P1, eventually at least one correct process enters $r_max(t^{\ell+1}) = r_max(t^\ell) + 1$, a contradiction to the assumption that no correct process enters any round $r > r_max(t^\ell)$. ◀

We prove the main theorem of this section:

▶ **Theorem 4.2.** *Using a synchronizer abstraction, Alg. 1 implements a round synchronization module.*

Proof. Since the relay function's outputs are identical among all correct processes and the leader local function outputs round_leader(r, Relay(r, 1)), it is immediate that the leader agreement property (Property 1) is satisfied.

We now prove eventual round synchronization (Property 2). Define $Leader(\ell) \triangleq \text{Relay}(r_max(t^\ell), 1)$. By Lemma 4.1, t^i occurs for all $i \geq 0$, and since the first relay for each round is randomly chosen, eventually, with probability 1, there exists a $\ell \geq 0$ such that $Leader(\ell)$ is a correct process. Let us look at $r_max(t^\ell)$, and denote $\widetilde{t} \triangleq t^\ell + c_1 + \Delta$.

Recall that t^ℓ is the time when the first correct process enters $r_max(t^\ell)$. By Line 2 in Alg. 1, no correct process calls advance() between t^ℓ and $\widetilde{t}$, and because of validity (Property 4) no correct process enters any round $r > r_max(t^\ell)$ until at least $\widetilde{t}$. By using Observation 1, we can apply S2 for $r_max(t^\ell)$, since by $t^\ell + c_1$ all correct processes enter $r_max(t^\ell)$.

Thus, between t^ℓ and $t^\ell + c_1$, all correct processes enter $r_max(t^\ell)$. Since no correct process calls advance() until at least $\widetilde{t}$, this guarantees that all correct processes remain in $r_max(t^\ell)$ until $\widetilde{t} = t^\ell + c_1 + \Delta$, so $t^\ell + c_1$ is a synchronization time (Def. 3.1), as needed. ◀

5 An Expected Linear Message Complexity and Constant Latency Synchronizer

In this section we present a synchronizer abstraction algorithm with expected linear message complexity and constant latency in the Byzantine case.

We start by describing the main ideas used to lower the message complexity (while still guaranteeing constant latency) in §5.1. We give a more in-depth description of the algorithm in §5.2, reason about the algorithm's correctness in §5.3, and performance in §5.4. Some of the formal proofs are deferred to the full version [33].

5.1 Achieving Linear Message Complexity

The crux of the algorithm is a relay-based distribution of messages among processes. A standard Byzantine broadcast system, which ensures that a message sent by a correct process is eventually delivered by all other correct processes, usually requires quadratic message

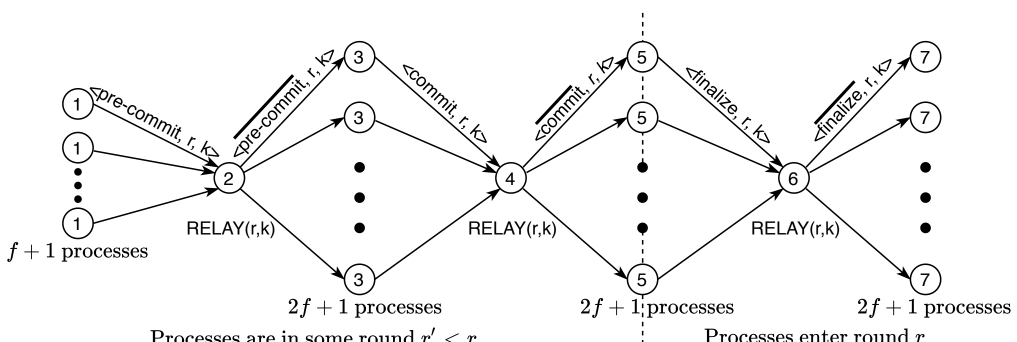


Figure 2 The message flow of the algorithm. A process enters round r when it receives a $\overline{\text{commit}}$ message for round r, the circled numbers represent the different stages of the algorithm (see Alg. 2). Only the $2f+1$ correct processes are illustrated.

complexity for each message disseminated. This is because in Byzantine broadcast protocols such as Bracha's [11], when a correct process delivers a message, it also sends it to all the other processes, resulting in all-to-all communication for each delivered message.

In our algorithm, we instead use a single designated process as a relay. Processes send their messages to the relay, which aggregates messages from a number of processes, combines them into one message using a threshold signature, and broadcasts it to all the processes. This mechanism reduces the total number of protocol messages from $O(n^2)$ to $O(n)$.

A difficulty arises if the relay is Byzantine. We overcome this as follows: when a process $\mathcal{P}$ sends a message to a relay, it expects a response from it within a certain time-bound. If no timely response arrives, $\mathcal{P}$ can deduce that either GST has not occurred yet and the message to/from the relay is delayed, or it is after GST and the relay is Byzantine. In either case, after the allotted time passes, $\mathcal{P}$ proceeds to send a message to a different relay, again waiting for the new relay to respond in a timely manner, and so on. This mechanism uses the relay function described in §2. Once a correct relay is contacted, the algorithm makes progress. In expectation, the number of consecutive Byzantine relays until a correct one is bounded by $3/2$, leading to expected constant latency and linear message complexity. In the worst-case, each round has $f+1$ potential relays, guaranteeing that at least one of them is correct, which ensures liveness.

5.2 Algorithm Description

At a high level, the goal of the algorithm is to eventually enter all rounds during the run, and reach a synchronization time after GST in every round r where $\textsc{Relay}(r,1)$ is a correct process. If the relay is Byzantine, then the goal is to eventually move from r to $r+1$. The randomization of the relay function guarantees that in an infinite run there will be infinitely many rounds with a correct process as the first relay, guaranteeing an infinite number of synchronization times.

Message flow of the algorithm. The algorithm is presented in Alg. 2, and its message flow is depicted in Figure 2. Protocol messages are signed and verified; for brevity, we omit the signatures and their verification from the algorithm description and pseudocode.

A process sends to $\textsc{Relay}(r, k)$ messages of the form $\langle \text{message type}, r, k\rangle$, where r and k are natural numbers, and message type is one of the following: pre-commit, commit, or finalize. The relay's messages to the processes are threshold signatures on an aggregation of the same messages, denoted $\langle\overline{\textsf{pre-commit}}, r, k\rangle$, $\langle\overline{\textsf{commit}}, r, k\rangle$, and $\langle\overline{\textsf{finalize}}, r, k\rangle$, respectively. Each threshold signature is created using some number ($f + 1$ or $2f + 1$) of signatures.

When $\textsf{advance}()$ is signaled via the local timer function (see §4.2) to indicate that it wants to move from round $r - 1$ to round r, the process sends a pre-commit message to the relay (this is stage 1 of the algorithm). Once $f + 1$ processes indicate that they wish to move to round r, the relay broadcasts a $\overline{\text{pre-commit}}$ message (stage 2). The reason $f+1$ processes are needed to initiate the first stage of the algorithm is to ensure that there is at least one correct process among them, preventing Byzantine processes from causing correct ones to advance prematurely. Any process receiving a relay's $\overline{\text{pre-commit}}$ message in a round $r' < r$ joins in by sending a commit message for r (stage 3). Unlike in previous work such as Cogsworth [32], $\overline{\text{pre-commit}}$ messages are linked to a particular relay, and therefore, if the protocol times out and proceeds to the next relay, the new relay needs to collect $f + 1$ pre-commits afresh. This subtle difference prevents Byzantine relays from spuriously engaging in the protocol, which is crucial for avoiding the quadratic message complexity occurring in Cogsworth.

When $2f + 1$ processes indicate that they commit to moving to r, the relay sends a $\overline{\text{commit}}$ message (stage 4) and processes that receive it enter that round (stage 5). Requiring $2f + 1$ processes to commit to a round r before entering it ensures that at least $f + 1$ correct processes are aware of the intent to enter r. This ensures that at least $f + 1$ correct processes will eventually enter r, and those $f + 1$ processes guarantee progress, as it is the minimal quorum required to initiate the stages of the algorithm to the next round, until a round with a first correct relay is reached and in that round a synchronization time will occur.

However, the algorithm for synchronizing for round r does not end when a process receives a $\overline{\text{commit}}$ message for r. Rather, a process that enters round r sends a finalize message to help any lagging processes with the transition to round r. Once $2f + 1$ finalize messages are sent, the relay broadcasts a $\overline{\text{finalize}}$ message (stage 6), and when a process receives it, it completes the algorithm for round r (stage 7). The finalization phase is needed to overcome cases of a Byzantine relay that does not send the $\overline{\text{commit}}$ message to all the processes.

Variables and timeouts. The variable *curr_round* stores the current round a process is currently in which changes in stage 5, and *next_round* indicates to what round the process is attempting to enter. The value of *next_round* becomes *curr_round* + 1 when a process invokes $\textsf{advance}()$, and it can become higher if the process learns (via a $\overline{\text{pre-commit}}$) of at least $f + 1$ other processes that want to advance to a higher round than the one the process is currently in.

The timeouts at the bottom of the pseudocode dictate when a process moves to the next relay of a round. When a process sends a message to a relay, it expects the relay to respond within 2δ, which is the upper bound of the round-trip time after GST. For example, if a process sends a message of round r to $\textsc{Relay}(r, k)$ at time t and does not receive a response by $t + 2\delta$, it sends the message to $\textsc{Relay}(r, k + 1)$. This continues up to $\textsc{Relay}(r, f + 1)$, guaranteeing that at least one of the relays for round r is correct.

Upon a timeout, a process sends a pre-commit message to the next relay in line, and once that relay gets f+1 such messages, it, too, can try to complete the stages of the protocol for the same round. There is a tradeoff involved in choosing the timeout – a shorter timeout may cause a second relay to engage even when the first relay is correct, whereas a longer one delays progress in case of a Byzantine relay. Nevertheless, it is important to note that a

process responds to all relays, so contacting the $(k+1)$-st relay for round r does not in any way prevent the k-th one from making progress. Thus, while setting an aggressive timeout may cause the protocol to send more messages, it does not in any way hamper progress. A process that partakes in the protocol to advance to round r contacts a new relay every 2δ time for as long as it does not make progress in the phases of the algorithm for round r. Since a process takes an expected 6δ to complete the algorithm for round r, the process contacts 3 relays in expectation.

The *round_relay* array holds the highest relay for each round the process sent a pre-commit message to. For example, $round_relay[r] = k$ for $k > 1$ indicates that the process sent $\langle \mathsf{pre\text{-}commit}, r, 1\rangle, \ldots, \langle \mathsf{pre\text{-}commit}, r, k\rangle$ messages to $\textsc{Relay}(r,1), \ldots, \textsc{Relay}(r,k)$, respectively. Note that a process sends a pre-commit message for round r to $\textsc{Relay}(r,1)$ when it first receives a $\overline{\text{pre-commit}}$ message in stage 3, regardless of the relay it received the message from. This is to allow the first relay of round r to complete the stages of the algorithm in case it is correct, and make sure that round synchronization will occur in round r. Note that the fact that some relay sends a message with a threshold signature does not ensure that that relay is correct, even if all the signatures used to create the threshold signature are from correct processes. For example, a Byzantine relay can broadcast a message only to a subset of the correct processes. Thus, to ensure liveness, processes must iterate through all $f+1$ relays of a round, starting from the first one, until progress is made.

We note that the *round_relay* array is introduced in the pseudocode for simplicity, but in a real implementation there is no need for an unbounded array to be stored in memory. A process only sends messages to the relays of rounds stored in the *curr_round* and *next_round* variables, thus limiting the amount of memory needed for an actual implementation to a constant number of integers.

Example. To clarify the need for the last phase of the algorithm (stages 6 and 7), consider the following scenario: Suppose a set P of $f+1$ correct processes are in round $r-1$ and invoke $\mathsf{advance}()$. The remaining f correct processes are in a round $r' < r-1$. The processes in P send a pre-commit message to $\textsc{Relay}(r,1)$, which is Byzantine. The relay generates a threshold signature and sends a $\overline{\text{pre-commit}}$ only to the processes in P, which respond with a commit message. Now, $\textsc{Relay}(r,1)$, with the help of f Byzantine processes, creates a $\overline{\text{commit}}$ message for r, but sends it to only one correct process $\mathcal{P}_i$ in P. This results in a scenario where $\mathcal{P}_i$ is the only correct process in round r, while f correct processes remain in round $r-1$ and continue to timeout and send pre-commit messages to the relays of round r. Since a relay needs at least $f+1$ pre-commit messages to engage the stages of the algorithm, unless $\mathcal{P}_i$ continues to help the rest of the processes in P by sending pre-commit messages, they might get stuck in round $r-1$. Therefore, processes continue to timeout and send pre-commit messages in the previous round until they receive a $\overline{\text{finalize}}$ message. Once a process in r receives a $\overline{\text{finalize}}$ message for r, it knows that there are at least $f+1$ correct processes in round r, and can stop sending pre-commit messages for r. This is crucial for achieving the desired message complexity after GST. These $f+1$ correct processes will eventually call $\mathsf{advance}()$ and proceed to round $r+1$.

Algorithm 2 **Synchronizer Algorithm.** The circles show the protocol's stages.

```
1  initialize:
2    curr_round ← 0 // Processes begin their execution at round 0.
3    next_round ← 0
4    ∀i ∈ ℕ: round_relay[i] ← 1
5    finalized ← True

Every process:
(1)
6  on advance() signal:
7    if curr_round < next_round then // old round
8      return
9    next_round ← curr_round + 1
10   send ⟨pre-commit, next_round, 1⟩ to Relay(next_round, 1)
(3)
13 upon receiving the first valid ⟨pre-commit, r, k⟩ from Relay(r, k):
14   if r < next_round then // old round
15     return
16   if r > next_round then /* start participating in round r */
17     next_round ← r
18     send ⟨pre-commit, r, 1⟩ to Relay(r, 1)
19   send ⟨commit, r, k⟩ to Relay(r, k)
(5)
22 upon receiving the first valid ⟨commit, r, k⟩ from Relay(r, k):
23   if r < curr_round then // old round
24     return
25   if r > curr_round then // enter round r
26     curr_round ← r
27     finalized ← False
28     send ⟨commit, r, 1⟩ to Relay(r, 1)
29     new_round(r) // signal new round
30   send ⟨finalize, r, k⟩ to Relay(r, k)
(7)
33 upon receiving the first valid ⟨finalize, r, k⟩ from Relay(r, k):
34   if r = curr_round then
35     finalized ← True

Relay (Relay(r, k)):
(2)
11 upon receiving the first valid f + 1 ⟨pre-commit, r, k⟩ messages:
12   broadcast ⟨pre-commit, r, k⟩
(4)
20 upon receiving the first valid 2f + 1 ⟨commit, r, k⟩ messages:
21   broadcast ⟨commit, r, k⟩
(6)
31 upon receiving the first valid 2f + 1 ⟨finalize, r, k⟩ messages:
32   broadcast ⟨finalize, r, k⟩

Timeouts (for every process):
36 on pre-commit and commit timeouts: /* Every 2δ from last sending pre-commit or commit messages and not receiving the matching pre-commit or commit */
37   if round_relay[next_round] < f + 1 then
38     round_relay[next_round] ← round_relay[next_round] + 1
39     send ⟨pre-commit, next_round, round_relay[next_round]⟩ to Relay(next_round, round_relay[next_round])
40 on finalize timeout: // Every 2δ from last sending finalize and not receiving the matching finalize
41   if finalized = False and round_relay[curr_round] < f + 1 then
42     round_relay[curr_round] ← round_relay[curr_round] + 1
43     send ⟨pre-commit, curr_round, round_relay[curr_round]⟩ to Relay(curr_round, round_relay[curr_round])
```

5.3 Correctness

Next, we prove that the algorithm satisfies the properties of a synchronizer, as defined in §4.1. The proofs of Lemma 5.1 and Lemma 5.5 are in the full version of the paper.

▶ **Lemma 5.1.** *Alg. 2 satisfies monotonically increasing rounds (Property 3).*

▶ **Lemma 5.2.** *Alg. 2 satisfies round validity (Property 4).*

Proof. A correct process enters round r when it is in a round $r' < r$ and receives a $\overline{\text{commit}}$ message for r. A $\overline{\text{commit}}$ message is a threshold signature of $(2f+1)$-of-n commit messages, meaning at least $f+1$ are from correct processes. A correct process sends a commit message for round r when it receives a $\overline{\text{pre-commit}}$ message for r. A $\overline{\text{pre-commit}}$ message is a threshold signature of $(f+1)$-of-n pre-commit messages, meaning at least one correct process sent a pre-commit message for round r.

Denote $\mathcal{P}_i$ as the first correct process that sends a pre-commit message for r during the run. A correct process only sends a pre-commit for r (in Lines 10, 18, 39, and 43) when its *next_round* or *curr_round* variables hold r. *next_round* changes in one of two places – Line 9 when a process calls advance(), and Line 17 on receiving a valid $\overline{\text{pre-commit}}$ for r. *curr_round* changes on receiving a valid $\overline{\text{commit}}$ for r. Because no $\overline{\text{pre-commit}}$ or $\overline{\text{commit}}$ message can be sent for round r before at least one correct process sends a pre-commit for r, then $\mathcal{P}_i$ must have sent its pre-commit message for round r when it changed its *next_round* in Line 9, i.e., on executing advance(). ◀

▶ **Proposition 5.3.** *If a correct process receives a $\overline{\text{finalize}}$ for round r at time t, then at least $f+1$ correct processes entered round r by t.*

Proof. Let t be a time in which a correct process received a $\overline{\text{finalize}}$ message for round r. This message is a threshold signature of $(2f+1)$-of-n finalize messages, of which at least $f+1$ originated from correct processes. A correct process only sends a finalize message for r if it receives a $\overline{\text{commit}}$ message for r, which means that it is already in round r by time t. ◀

▶ **Lemma 5.4.** *Alg. 2 satisfies stabilization (Property 5) with $c_1 = 4\delta$.*

Proof. Let t be a point in time during the execution and $r = r_max(t)$. Let $\mathcal{P}_i$ be the first correct process that enters round r at time t_0. Such a process exists by the definition of $r_max(t)$. $\mathcal{P}_i$ is at round r, so it received a $\overline{\text{commit}}$ message for round r. A $\overline{\text{commit}}$ message is a threshold signature of $(2f+1)$-of-n commit messages, at least $f+1$ of which were sent by correct processes. Denote by S the set of correct processes whose signatures on commit messages are included in the $\overline{\text{commit}}$ message $\mathcal{P}_i$ received. The processes in S are either in round r at time t or in smaller rounds $r' < r$.

We now prove the two sub-properties of Property 5:

S1. If some correct process receives $\overline{\text{finalize}}$ for round r, by Proposition 5.3, there are at least $f+1$ correct processes in r and we are done.

Assume no correct process receives $\overline{\text{finalize}}$. Then, the processes in S continue to timeout and send pre-commit messages for round r to the relays of r. This guarantees that eventually, a correct relay for r receives at least $f+1$ pre-commit messages, as Property R1 of the relay function ensures $f+1$ different relays for each round. This relay eventually completes the stages of the algorithm, allowing all correct processes to advance to round r.

S2. Because $\mathcal{P}_i$ receives $\overline{\text{commit}}$ for round r at time $t_0 \geq \text{GST}$, as argued above, $f+1$ correct processes have sent a commit message for round r by time t_0. Because a process sends pre-commit to $\text{Relay}(r, 1)$ before sending a commit to any relay for round r (Lines

10 or 18), these messages, too, are sent by time t_0. Therefore, by time $t_0 + \delta$, Relay$(r, 1)$ receives $f + 1$ pre-commit messages and sends a $\overline{\text{pre-commit}}$ message to all processes. By $t_0 + 2\delta$ all the correct processes receive the $\overline{\text{pre-commit}}$ message sent from the first relay, by $t_0 + 3\delta$ the relay receives $2f + 1$ commit messages (along with any process that already entered r, Line 28), and by $t_2 \leq t_0 + 4\delta$ all the correct processes receive the $\overline{\text{commit}}$ message and enter round r. ◀

▶ **Lemma 5.5.** *Alg. 2 satisfies progress (Property 6) with $c_2 = 4\delta$.*

The following theorem follows directly from Lemmas 5.1, 5.2, 5.4, and 5.5.

▶ **Theorem 5.6.** *Alg. 2 satisfies the synchronizer abstraction.*

5.4 Performance: Latency and Message Complexity

We prove that our algorithm has expected constant latency and linear message complexity. The proofs of Proposition 5.7 and Lemmas 5.8 and 5.9 appear in the full version of the paper.

▶ **Proposition 5.7.** *For any round r, let X_r be the number of consecutive Byzantine relays until the first correct relay. Then, $\forall r\colon \mathbb{E}[X_r] \leq 3/2$.*

▶ **Lemma 5.8.** *For any $t \geq GST$ let t_0 be the first time during the run where a correct process enters $r_max(t)$. There exists a time $t_1 \geq t_0$ such that up to t_1 either (i) at least $f+1$ correct processes are in $r_max(t)$ **or** (ii) a correct process enters a round $r > r_max(t)$; and $\mathbb{E}[t_1 - \max\{t_0, GST\}] \leq \frac{3}{2} \cdot 6\delta$.*

▶ **Lemma 5.9.** *For any $t \geq GST$ let t_0 be the first time when $f + 1$ correct processes call advance() while in round $r_max(t)$. There exists a time $t_1 \geq t_0$ such that there is at least one correct process in $r_max(t) + 1$ and $\mathbb{E}[t_1 - \max\{t_0, GST\}] \leq \frac{3}{2} \cdot 6\delta$.*

▶ **Theorem 5.10.** *The synchronizer algorithm along with a Timer local function (as defined in §4) achieves expected constant latency and linear message complexity.*

Proof. The latency for our algorithm is based on the definition in §3.2. We go over all possible states after GST the correct processes in our algorithm can be in, and look at the expected latency until the synchronization time. Let $t^0 = \text{GST}$ and for all $\ell \geq 1$ let t^ℓ represent the first time after $t^{\ell-1}$ that a correct process enters a new maximum round. By Lemma 4.1, in an infinite run, t^ℓ eventually occurs for any $\ell \geq 0$. For any time $t \geq \text{GST}$ during the run, let $sync_time(t)$ be the first time after t until a synchronization time (Def. 3.1). To calculate the expected latency of our algorithm, we need to show that for any $t \geq \text{GST}$, $E_1 \triangleq \mathbb{E}[sync_time(t) - t] \leq O(\delta)$.

Denote E_2 as the expected time from any time $t \geq \text{GST}$ until the next t^ℓ, i.e., for any $l \geq 0$ and t, $E_2 \triangleq \mathbb{E}\left[\min_{t^\ell \geq t}\left\{t^\ell\right\} - t\right]$ and $E_3 \triangleq \mathbb{E}\left[t^{\ell+1} - t^\ell\right]$. If Relay$(r_max(t^\ell), 1)$ is correct, then based on P2, by $t^\ell + 4\delta$ all the correct processes enter $r_max(t^\ell)$. Therefore:

$$
\begin{aligned}
E_1 &\leq E_2 + \underbrace{\frac{n-f}{n}}_{\substack{\text{Probability that}\\ \text{Relay}(r_max(t^\ell),1)\\ \text{is correct}}} \cdot \underbrace{4\delta}_{\substack{\text{The maximum time for}\\ \text{all correct processes}\\ \text{to enter a round}\\ \text{(Lemma 5.4)}}} + \underbrace{\frac{f}{n}}_{\substack{\text{Probability that}\\ \text{Relay}(r_max(t^\ell),1)\\ \text{is Byzantine}}} \cdot \underbrace{\mathbb{E}\left[sync_time(t^{\ell+1}) - t^\ell\right]}_{\substack{\text{The expected time until all}\\ \text{correct processes}\\ \text{enter } r_max(sync_time(t^{\ell+1}))}} = \\
&= E_2 + \frac{n-f}{n} \cdot 4\delta + \frac{f}{n} \cdot \Big(E_1 + \underbrace{\mathbb{E}\left[t^{\ell+1} - t^\ell\right]}_{=E_3}\Big) \\
&\Rightarrow E_1 \leq \frac{n}{n-f}\left(E_2 + \frac{n-f}{n} \cdot 4\delta + \frac{f}{n} \cdot E_3\right). \qquad (1)
\end{aligned}
$$

Assuming that once a correct process enters a new round, the timer calls advance() within $4\delta + \Delta$, the expected time between t^ℓ and $t^{\ell+1}$ can be bounded as follows:

$$E_3 = \mathbb{E}\left[t^{\ell+1} - t^\ell\right] \leq \underbrace{\mathbb{E}\begin{bmatrix}\text{Time from } t^\ell \text{ until} \\ \text{at least } f+1 \text{ correct} \\ \text{processes enter } r_max(t^\ell) \\ \text{or } t^{\ell+1} \text{ occurs}\end{bmatrix}}_{\text{Lemma 5.8}} + \underbrace{4\delta + \Delta}_{\substack{\text{Time until at least } f+1 \\ \text{correct processes call} \\ \text{advance() in } r_max(t^\ell)}} + \underbrace{\mathbb{E}\begin{bmatrix}\text{Time from the first} \\ \text{time } f+1 \text{ correct} \\ \text{processes call advance()} \\ \text{in } r_max(t^\ell) \text{ and} \\ \text{until } t^{\ell+1} \text{ occurs}\end{bmatrix}}_{\text{Lemma 5.9}}$$

$$\leq \frac{3}{2} \cdot 6\delta + 4\delta + \Delta + \frac{3}{2} \cdot 6\delta = 22\delta + \Delta.$$

The calculation of E_3 proves that in expectation, the time between any t^ℓ and $t^{\ell+1}$ is expected constant, assuming Δ is constant. Therefore, E_2 is also expected constant.

To conclude, we proved that $E_2 \leq O(\delta)$ and $E_3 \leq O(\delta)$, and by Eq. (1), $E_1 \leq O(\delta)$, as needed to prove expected constant latency.

For the message complexity of the synchronizer, note that since the expected time between two occurrences of round synchronization is expected constant, the message complexity is expected linear. This is because for a given round the number of consecutive Byzantine relays until a correct one is expected constant, and in the algorithm, every process sends one message to the relay in each stage of the algorithm, and the relay responds with one message to all the processes. Even if a process contacts more than one relay per round, it still contacts an expected constant number of relays, and therefore this does not hamper the asymptotic linear message complexity. ◀

5.5 Relaxed Model

As part of the model in §2 we assumed that the adversary is oblivious. If the adversary is *strong*, and knows the randomness $\mathcal{R}$ before choosing which processes to corrupt, a worst-case bound is tantamount to a deterministic one, because it holds for all coin flips. Therefore, we cannot hope to get a linear message complexity for the worst-case. Nevertheless, in a run with infinitely many round synchronization events, we can bound the *average-case* expected latency and message complexity by considering the limit of the average latency and message complexity on prefixes of length t of the run as t tends to infinity.

Thus, a strong adversary who is aware of the relay function, can choose to corrupt, e.g., the first f relays of some round r, causing that round to have linear latency and quadratic message complexity. But since the adversary is static, it has to corrupt the same processes in all rounds, and by property R2, this does not impact the average-case performance.

6 Related Work

Algorithms for the eventual synchrony model almost invariably use the notion of round or views [30, 34, 26, 6]. A number of works have suggested frameworks and mechanisms for round synchronization in the benign case [3, 23, 27, 28, 24]. For example, Awerbuch introduced synchronizers [3] for failure-free networks. TLC [23] places a barrier on round advancement, so that processes enter round $r+1$ only after a threshold of the processes entered round r. Frameworks like RRFD [24] and GIRAF [27, 28] create a round-based structure for eventually synchronous and failure-detector based algorithms.

A related concurrent work due to Bravo et al. [12] also tackles the liveness of consensus protocols, and creates a general framework to abstract the liveness part of consensus protocols. They show that some protocols such as PBFT [16] and HotStuff [37] can use this framework.

They also provide an algorithm for round synchronization that, starting from some round r, synchronizes all rounds $r' \geq r$ (even rounds with a Byzantine leader, in which decisions are not made), but unlike our algorithm, it requires a quadratic communication cost per synchronization event. They assume a relaxed network model compared to us, where before GST messages might be lost.

Several algorithms include two modes of operation: a normal mode where the leader is correct incurring linear message complexity, and a recovery mode when the leader is faulty and needs to be replaced incurring quadratic or higher message complexity. For example, PABC [35] achieves amortized linear message complexity in an asynchronous atomic broadcast protocol, and Zyzzyva [29] implements a linear fast-track in an SMR algorithm.

Randomization is often used to solve consensus in asynchronous networks to circumvent the seminal FLP result [22]. VABA [1] is the first multi-value asynchronous consensus algorithm that achieves an expected quadratic message complexity against a strong adaptive adversary, and other works in the asynchronous model [18, 7] achieve expected sub-quadratic communication complexity under various assumptions, but do not achieve $O(n)$ communication complexity. In the context of Byzantine SMR, HotStuff [37] specified the round synchronization conditions needed for their algorithm, and abstracted it into a module that was left unspecified. Our work provides the round synchronization they require.

Our algorithm builds on ideas presented in Cogsworth [32], but Cogsworth achieved expected linear message complexity only in the benign case, whereas in the Byzantine case its message complexity was still expected quadratic. In Cogsworth, it is enough that the first relay of a round is Byzantine to create a run with a quadratic number of messages to synchronize for that round. Since the probability that the first relay is Byzantine is constant, i.e., f/n, the overall message complexity under Byzantine failures is expected quadratic.

To reduce the expected message complexity to linear, we modified Cogsworth in a number of ways, including adding another phase to the algorithm, signing each message from a process to a relay with the relay it is intended for, and adding a "helping" mechanism to help processes "catch-up" to the latest round. By incorporating these ideas into our algorithm, we managed to bring the expected message complexity down to linear.

7 Conclusion

We presented an algorithm that reduces the expected message complexity of round synchronization to linear with an expected constant latency. Combined with algorithms like HotStuff, this yields, for the first time, Byzantine SMR with the same asymptotic performance, as round synchronization is the "bottleneck" in previous Byzantine SMR algorithms. While we achieve only expected sub-quadratic complexity, we note that achieving the same complexity in the worst-case is known to be impossible [19], and so cannot be improved.

References

1 Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. Asymptotically optimal validated asynchronous byzantine agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 337–346, 2019.

2 Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. Secure multiparty computations on bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 443–458. IEEE, 2014.

3 Baruch Awerbuch. Complexity of network synchronization. *Journal of the ACM (JACM)*, 32(4):804–823, 1985.

4. Mathieu Baudet, Avery Ching, Andrey Chursin, George Danezis, François Garillot, Zekun Li, Dahlia Malkhi, Oded Naor, Dmitri Perelman, and Alberto Sonnino. State machine replication in the libra blockchain, 2019.
5. Shai Ben-David, Allan Borodin, Richard Karp, Gabor Tardos, and Avi Wigderson. On the power of randomization in on-line algorithms. *Algorithmica*, 11(1):2–14, 1994.
6. Ken Birman and Thomas Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the eleventh ACM Symposium on Operating systems principles*, pages 123–138, 1987.
7. Erica Blum, Jonathan Katz, Chen-Da Liu-Zhang, and Julian Loss. Asynchronous byzantine agreement with subquadratic communication. Cryptology ePrint Archive, Report 2020/851, 2020.
8. Manuel Blum. Coin flipping by telephone a protocol for solving impossible problems. *ACM SIGACT News*, 15(1):23–27, 1983.
9. Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 514–532. Springer, 2001.
10. Allan Borodin, Nathan Linial, and Michael E Saks. An optimal on-line algorithm for metrical task system. *Journal of the ACM (JACM)*, 39(4):745–763, 1992.
11. Gabriel Bracha. Asynchronous Byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.
12. Manuel Bravo, Gregory Chockler, and Alexey Gotsman. Making byzantine consensus live. In *34st International Symposium on Distributed Computing (DISC 2020)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2020.
13. Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on BFT consensus. *arXiv preprint arXiv:1807.04938*, 2018.
14. Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *arXiv preprint arXiv:1710.09437*, 2017.
15. 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.
16. Miguel Castro, Barbara Liskov, et al. Practical Byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.
17. Richard Cleve. Limits on the security of coin flips when half the processors are faulty. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 364–369, 1986.
18. Shir Cohen, Idit Keidar, and Alexander Spiegelman. Not a coincidence: Sub-quadratic asynchronous byzantine agreement whp. In *34st International Symposium on Distributed Computing (DISC 2020)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2020.
19. Danny Dolev and Ruediger Reischuk. Bounds on information exchange for byzantine agreement. In *Proceedings of the First ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '82, page 132–140, New York, NY, USA, 1982. Association for Computing Machinery.
20. Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.
21. Amos Fiat, Richard Karp, Mike Luby, Lyle McGeoch, Daniel Sleator, and Neal E Young. Competitive paging algorithms. *arXiv preprint cs/0205038*, 2002.
22. Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.
23. Bryan Ford. Threshold logical clocks for asynchronous distributed coordination and consensus. *arXiv preprint arXiv:1907.07010*, 2019.

24 Eli Gafni. Round-by-round fault detectors (extended abstract) unifying synchrony and asynchrony. In *Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing*, pages 143–152, 1998.

25 Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. SBFT: a scalable and decentralized trust infrastructure. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 568–580. IEEE, 2019.

26 Idit Keidar and Danny Dolev. Efficient message ordering in dynamic networks. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 68–76, 1996.

27 Idit Keidar and Alexander Shraer. Timeliness, failure-detectors, and consensus performance. In *Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, pages 169–178, 2006.

28 Idit Keidar and Alexander Shraer. How to choose a timing model. *IEEE Transactions on Parallel and Distributed Systems*, 19(10):1367–1380, 2008.

29 Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 45–58. ACM, 2007.

30 Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

31 Tal Moran, Moni Naor, and Gil Segev. An optimally fair coin toss. In *Theory of Cryptography Conference*, pages 1–18. Springer, 2009.

32 Oded Naor, Mathieu Baudet, Dahlia Malkhi, and Alexander Spiegelman. Cogsworth: Byzantine view synchronization. In *Proceedings of the Cryptoeconomic Systems Conference (CES'20)*, 2020.

33 Oded Naor and Idit Keidar. Expected linear round synchronization: The missing link for linear byzantine smr. *arXiv preprint arXiv:2002.07539*, 2020.

34 Brian M Oki and Barbara H Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, pages 8–17. ACM, 1988.

35 HariGovind V Ramasamy and Christian Cachin. Parsimonious asynchronous byzantine-fault-tolerant atomic broadcast. In *International Conference On Principles Of Distributed Systems*, pages 88–102. Springer, 2005.

36 Victor Shoup. Practical threshold signatures. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 207–220. Springer, 2000.

37 Maofan Yin, Dahlia Malkhi, MK Reiter and, Guy Golan Gueta, and Ittai Abraham. HotStuff: BFT consensus with linearity and responsiveness. In *38th ACM symposium on Principles of Distributed Computing (PODC'19)*, 2019.

Asynchronous Reconfiguration with Byzantine Failures

Petr Kuznetsov
LTCI, Télécom Paris, Institut Polytechnique Paris, France
petr.kuznetsov@telecom-paris.fr

Andrei Tonkikh
National Research University Higher School of Economics, Saint-Petersburg, Russia
andrei.tonkikh@gmail.com

Abstract

Replicated services are inherently vulnerable to failures and security breaches. In a long-running system, it is, therefore, indispensable to maintain a *reconfiguration* mechanism that would replace faulty replicas with correct ones. An important challenge is to enable reconfiguration without affecting the availability and consistency of the replicated data: the clients should be able to get correct service even when the set of service replicas is being updated.

In this paper, we address the problem of reconfiguration in the presence of Byzantine failures: faulty replicas or clients may arbitrarily deviate from their expected behavior. We describe a generic technique for building *asynchronous* and *Byzantine fault-tolerant* reconfigurable objects: clients can manipulate the object data and issue reconfiguration calls without reaching consensus on the current configuration. With the help of forward-secure digital signatures, our solution makes sure that superseded and possibly compromised configurations are harmless, that slow clients cannot be fooled into reading stale data, and that Byzantine clients cannot cause a denial of service by flooding the system with reconfiguration requests. Our approach is modular and based on *dynamic lattice agreement* abstraction, and we discuss how to extend it to enable Byzantine fault-tolerant implementations of a large class of reconfigurable replicated services.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Reconfiguration, Asynchronous Models, Byzantine Faults

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.27

Related Version A full version of the paper is available at [27], `https://arxiv.org/abs/2005.13499`.

1 Introduction

Replication and quorums. Replication is a natural way to ensure availability of shared data in the presence of failures. A collection of *replicas*, each holding a version of the data, ensure that the *clients* get a desired service, even when some replicas become unavailable or hacked by a malicious adversary. Consistency of the provided service requires the replicas to *synchronize*: intuitively, every client should be able to operate on the most "up-to-date" data, regardless of the set of replicas it can reach.

It always makes sense to assume as little as possible about the environment in which a system we design is expected to run. For example, *asynchronous* distributed systems do not rely on timing assumptions, which makes them extremely robust with respect to communication disruptions and computational delays. It is, however, notoriously difficult and sometimes even impossible to make such systems *fault-tolerant*. The folklore CAP theorem [12, 21] states that no replicated service can combine consistency, availability, and partition-tolerance. In particular, no consistent and available read-write storage can be implemented in the presence of partitions: clients in one partition are unable to keep track of the updates taking place in another one.

© Petr Kuznetsov and Andrei Tonkikh;
licensed under Creative Commons License CC-BY
34th International Symposium on Distributed Computing (DISC 2020).
Editor: Hagit Attiya; Article No. 27; pp. 27:1–27:17
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Therefore, fault-tolerant storage systems tend to assume that partitions are excluded, e.g., by requiring a majority of replicas to be *correct* [6]. More generally, one can assume a *quorum system*, e.g., a set of subsets of replicas satisfying the intersection and availability properties [20]. Every (read or write) request from a client should be *accepted* by a quorum of replicas. As every two quorums have at least one replica in common, intuitively, no client can miss previously written data.

Of course, failures of replicas may jeopardize the underlying quorum system. In particular, we may find ourselves in a system in which no quorum is available and, thus, no operation may be able to terminate. Even worse, if the replicas are subject to Byzantine failures, we may not be able to guarantee the very correctness of read values.

Asynchronous reconfiguration. To anticipate such scenarios in a long run, we must maintain a *reconfiguration* mechanism that enables replacing compromised replicas with correct ones and update the corresponding quorum assumptions. A challenge here is to find an asynchronous implementation of reconfiguration in a system where both clients and replicas are subject to Byzantine failures that can be manifested by arbitrary and even malicious behavior. In the world of selfishly driven blockchain users, a reconfiguration mechanism must be prepared to this.

Recently, a number of reconfigurable systems were proposed for asynchronous *crash*-fault environments [2, 19, 23, 3, 34, 26] that were originally applied to (read-write) storage systems [2, 19, 3], and then extended to max-registers [23, 34] and more general *lattice* data type [26].

These proposals tend to ensure that the clients reach a form of "loose" agreement on the currently active configurations, which can be naturally expressed via the *lattice agreement* abstraction [8, 16]. We allow clients to (temporarily) live in different worlds, as long as these worlds are properly ordered. For example, we may represent a configuration as a set of *updates* (additions and removals of replicas) and require that all installed configurations should be related by containment. A configuration becomes *stale* as soon as it is subsumed by a new one representing a proper superset of updates.

Challenges of Byzantine fault-tolerant reconfiguration. In this paper, we focus on *Byzantine fault-tolerant* reconfiguration mechanism. We had to address here several challenges, specific to dynamic systems with Byzantine faults, which make it hard to benefit from existing crash fault-tolerant reconfigurable solutions.

First, when we build a system out of lower-level components, we need to make sure that the outputs provided by these components are "authentic". Whenever a (potentially Byzantine) process claims to have obtained a *value* v (e.g., a new configuration estimate) from an underlying object (e.g., Lattice Agreement), it should also provide a *proof* σ that can be independently verified by every correct process. The proof typically consists of digital signatures provided by a quorum of replicas of some configuration. We abstract this requirement out by equipping the object with a function VerifyOutputValue that returns a boolean value, provided v and σ. When invoked by a correct process, the function returns *true* if and only if v has indeed been produced by the object. When "chaining" the objects, i.e., adopting the output v provided by an object A as an input for another object B, which is the typical scenario in our system, a correct process invokes $A.\text{VerifyOutputValue}(v, \sigma)$, where σ is the proof associated with v by the implementation of A. This way, only values actually produced by A can be used as inputs to B.

Second, we face the **"I still work here" attack** [1]. It is possible that a client that did not log into the system for a long time tries to access a *stale*, outdated configuration in which some quorum is entirely compromised by the Byzantine adversary. The client can therefore be provided with an inconsistent view on the shared data. Thus, before accepting a new configuration, we need to make sure that the stale ones are no longer capable of processing data requests from the clients. We address this issue via the use of a forward-secure signature scheme [10]. Intuitively, every replica is provided with a distinct private key associated to each configuration. Before a configuration is replaced with a newer one, at least a quorum of its replicas are asked to destroy their private keys. Therefore, even if the replicas are to become Byzantine in the future, they will not be able to provide slow clients with inconsistent values. The stale configuration simply becomes non-responsive, as in crash-fault-tolerant reconfigurable systems.

Unfortunately, in an asynchronous system it is impossible to make sure that replicas of *all* stale configurations remove their private keys as it would require solving consensus [17]. However, as we show in this paper, it is possible to make sure that the configurations in which replicas do not remove their keys are never accessed by correct clients and are incapable of creating "proofs" for output values.

Finally, there is a subtle, and quite interesting **"slow reader" attack**. Suppose that a client accesses *almost all* replicas in a quorum of the current configuration each holding a stale state, as the only correct replica in the quorum that has the up-to-date state has not yet responded. The client then falls asleep, meanwhile, the configuration is superseded by a new one. As we do not make any assumptions about the correctness of replicas in stale configurations, the replica that has not yet responded can be compromised. Moreover, due to asynchrony, this replica can still retain its original private key. The replica can then pretend to be unaware of the current state. Therefore, the slow client might still be able to complete its request in the superseded configuration and return a stale state, which would violate the safety properties of the system. We show that this issue can be addressed by an additional "confirming" round-trip executed by the client.

Our contribution: Byzantine fault-tolerant reconfigurable services. We provide a systematic solution to each of the challenges described above and present a set of techniques for building reconfigurable services in asynchronous model with Byzantine faults of both clients and replicas. We consider a very strong model of the adversary: any number of clients can be Byzantine and, as soon as some configuration is installed, no assumptions are made about the correctness of replicas in any of the prior configurations.

Moreover, in our quest for a simple solution for the Byzantine model, we devised a new approach to building asynchronous reconfigurable services by further exploring the connection between reconfiguration and the lattice agreement abstraction [23, 26]. We believe that this approach can be usefully applied to crash fault-tolerant systems as well.

Instead of trying to build a complex graph of configurations "on the fly" while transferring the state between those configurations, we start by simply assuming that we are already given a *linear history* (i.e., a sequence of configurations). We introduce the notion of a *dynamic object* – an object that can transfer its own state between the configurations of a given finite linear history and serve meaningful user requests. We then provide dynamic implementations of several important object types such as Lattice Agreement and Max-Register and expect that other asynchronous static algorithms can be translated to the dynamic model using a similar set of techniques.

Finally, we present a *general transformation* that allows us to combine *any* dynamic object with two *Dynamic Byzantine Lattice Agreement* objects in such a way that together they constitute a single *reconfigurable* object, which exports a general-purpose reconfiguration interface and supports all the operations of the original dynamic object.

Roadmap. The rest of the paper is organized as follows. We overview the model assumptions in Section 2 and define our principal abstractions in Section 3. In Section 4, we describe our implementation of Dynamic Byzantine Lattice Agreement and in Section 5, we show how to use it to implement a reconfigurable object. We discuss related work in Section 6 and conclude in Section 7. We refer to [27] for the full version of the paper.

2 System Model

Processes and channels. We consider a system of *processes*. A process can be a *replica* or a *client*. Let Φ and Π denote the (possibly infinite) sets of replicas and clients, resp., that potentially can take part in the computation. At any point in a given execution, a process can be in one of the four states: *idle*, *correct*, *halted*, or *Byzantine*. A process is *idle* if it has not taken a single step in the execution yet. A process stops being idle by taking a step, e.g., sending or receiving a message. A process is considered *correct* as long as it respects the algorithm it is assigned. A process is *halted* if it executed the special "halt" command and not taking any further steps. Finally, a process is *Byzantine* if it prematurely stops taking steps of the algorithm or takes steps that are not prescribed by it. A correct process can later halt or become Byzantine. However, the reverse is impossible: a halted or Byzantine process cannot become correct. We assume that a process that remains correct forever (we call it *forever-correct*) does not prematurely stop taking steps of its algorithm.

We assume asynchronous *reliable authenticated* point-to-point links between each pair of processes [13]. If a forever-correct process p sends a message m to a forever-correct process q, then q eventually delivers m. Moreover, if a correct process q receives a message m from a process p at time t, and p is correct at time t, then p has indeed sent m to q before t.

We assume that the adversary is computationally bounded so that it is unable to break the cryptographic techniques, such as digital signatures, forward security schemes [10] and one-way hash functions.

Configuration lattice. A *join semi-lattice* (or simply a *lattice*) is a tuple $(\mathcal{L}, \sqsubseteq)$, where $\mathcal{L}$ is a set partially ordered by the binary relation $\sqsubseteq$ such that for all elements $x, y \in \mathcal{L}$, there exists the *least upper bound* for the set $\{x, y\}$, i.e., the element $z \in \mathcal{L}$ such that $x, y \sqsubseteq z$ and $\forall\, w \in \mathcal{L}$: if $x, y \sqsubseteq w$, then $z \sqsubseteq w$. The least upper bound for the set $\{x, y\}$ is denoted by $x \sqcup y$. $\sqcup$ is called the *join operator*. It is an associative, commutative, and idempotent binary operator on $\mathcal{L}$. We write $x \sqsubset y$ whenever $x \sqsubseteq y$ and $x \neq y$. We say that $x, y \in \mathcal{L}$ are *comparable* iff either $x \sqsubseteq y$ or $y \sqsubset x$.

For any (potentially infinite) set A, $(2^A, \sqsubseteq)$ is a join semi-lattice, called *the powerset lattice of* A. For all $Z_1, Z_2 \in 2^A$, $Z_1 \sqsubseteq Z_2 \triangleq Z_1 \subseteq Z_2$ and $Z_1 \sqcup Z_2 \triangleq Z_1 \cup Z_2$.

A configuration is an element of a join semi-lattice $(\mathcal{C}, \sqsubseteq)$. We assume that every configuration is associated with a finite set of replicas via a map $replicas : \mathcal{C} \to 2^{\Phi}$, and a *quorum system* via a map $quorums : \mathcal{C} \to 2^{2^{\Phi}}$, such that $\forall C \in \mathcal{C} : quorums(C) \subseteq 2^{replicas(C)}$. Additionally we assume that there is a map $height : \mathcal{C} \to \mathbb{Z}$, such that $\forall C \in \mathcal{C} : height(C) \geq 0$ and $\forall C_1, C_2 \in \mathcal{C}$: if $C_1 \sqsubset C_2$, then $height(C_1) < height(C_2)$. We say that a configuration C is *higher* (resp., *lower*) than a configuration D iff $D \sqsubset C$ (resp, $C \sqsubset D$).[1]

[1] Notice that "C is higher than D" implies "$height(C) > height(D)$", but not vice versa.

We say that $\mathit{quorums}(C)$ is a *dissemination quorum system* at time t iff every two sets (also called *quorums*) in $\mathit{quorums}(C)$ have at least one replica in common that is correct at time t, and at least one quorum is *available* (all its replicas are correct) at time t.

A natural (but not the only possible) way to define the lattice $\mathcal{C}$ is as follows: let *Updates* be $\{+,-\} \times \Phi$, where tuple $(+,p)$ means "add replica p" and tuple $(-,p)$ means "remove replica p". Then $\mathcal{C}$ is the powerset lattice $(2^{\mathit{Updates}}, \sqsubseteq)$. The mappings *replicas*, *quorums*, and *height* are defined as follows: $\mathit{replicas}(C) \triangleq \{s \in \Phi \mid (+,s) \in C \wedge (-,s) \notin C\}$, $\mathit{quorums}(C) \triangleq \{Q \subseteq \mathit{replicas}(C) \mid |Q| > \frac{2}{3}\,|\mathit{replicas}(C)|\}$, and $\mathit{height}(C) \triangleq |C|$. It is straightforward to verify that $\mathit{quorums}(C)$ is a dissemination quorum system when strictly less than one third of replicas in $\mathit{replicas}(C)$ are faulty. Notice that, when this lattice is used for configurations, once a replica is removed from the system, it cannot be added again with the same identifier. In order to add such a replica back to the system, a new identifier must be used.

Forward-secure digital signatures. In a *forward-secure digital signature scheme* [10, 30, 11, 15] the public key of a process is fixed while the secret key can evolve. Each signature is associated with a *timestamp*. To generate a signature with timestamp t, the signer uses secret key sk_t. The signer can *update its secret key* and get sk_{t_2} from sk_{t_1} if $t_1 < t_2 \leq T$.[2] However "downgrading" the key to a lower timestamp, from sk_{t_2} to sk_{t_1}, is computationally infeasible. This way, if the signer updates their secret key to some timestamp t and then removes the original secret key, it will not be able to sign new messages with a timestamp lower than t, even if it later turns Byzantine.

For simplicity, we model a forward-secure signature scheme as an oracle which associates every process p with a timestamp st_p (initially, $st_p = 0$). The oracle provides p with three operations: (1) $\mathrm{UpdateFSKey}(t)$ sets st_p to $t \geq st_p$; (2) $\mathrm{FSSign}(m,t)$ returns a signature for message m and timestamp t if $t \geq st_p$, otherwise it returns $\perp$; and (3) $\mathrm{FSVerify}(m,p,s,t)$ returns *true* iff s was generated by invoking $\mathrm{FSSign}(m,t)$ by process p.[3]

In our protocols, we use the height of the configuration as the timestamp. When a replica answers requests in configuration C, it signs messages with timestamp $\mathit{height}(C)$. When a higher configuration D is installed, the replica invokes $\mathrm{UpdateFSKey}(\mathit{height}(D))$. This prevents the "I still work here" attack described in Section 1.

3 Abstractions and Definitions

In this section, we introduce principal abstractions of this paper (access control-interface, Byzantine Lattice Agreement, Reconfigurable and Dynamic objects), state our quorum assumptions, and recall the definitions of broadcast primitives used in our algorithms.

3.1 Access control

In our implementations and definitions, we parametrize some abstractions by boolean functions $\mathrm{VerifyInputValue}(v,\sigma)$ and $\mathrm{VerifyInputConfig}(C,\sigma)$, where σ is called a *certificate*. Moreover, some objects also export a boolean function $\mathrm{VerifyOutputValue}(v,\sigma)$, which lets anyone to verify that the value v was indeed produced by the object. This helps us to deal with Byzantine clients. In particular, it achieves three important goals.

[2] T is a parameter of the scheme and can be set arbitrarily large (with some modest overhead). We believe that $T = 2^{32}$ or $T = 2^{64}$ should be sufficient for most applications.

[3] We assume that anyone who knows the id of a process also knows its public key. For example, the public key can be directly embedded into the identifier.

First, the parameter VerifyInputConfig allows us to prevent Byzantine clients from reconfiguring the system in an undesirable way or simply flooding the system with excessively frequent reconfiguration requests. In the full version of this paper [27], we propose two simple approaches: each reconfiguration request must be signed by a quorum of replicas of some configuration[4] or by a quorum of preconfigured administrators.

Second, the parameter VerifyInputValue(v, σ) allows us to formally capture the application-specific notions of well-formed client requests and access control. For example, in a key-value storage system, each client can be permitted to modify only the key-value pairs that were created by this client. In this case, the certificate σ is just a digital signature of that client.

Finally, the exported function VerifyOutputValue allows us to *chain* several distributed objects in such a way that the output of one object is passed as input for another one.

3.2 Byzantine Lattice Agreement abstraction

In this section we formally define *Byzantine Lattice Agreement* abstraction (BLA for short), which serves as one of the main building blocks for constructing reconfigurable objects. Byzantine Lattice Agreement is an adaptation of Lattice Agreement [16] that can tolerate Byzantine failures of processes (both clients and replicas). It is parameterized by a join semi-lattice $\mathcal{L}$, called the *object lattice*, and a boolean function VerifyInputValue : $\mathcal{L} \times \Sigma \to \{true, false\}$, where Σ is a set of possible certificates. We say that σ is a *valid certificate for input value* v iff VerifyInputValue$(v, \sigma) = true$.

We say that $v \in \mathcal{L}$ is a *verifiable input value* in a given run iff at some point in time in that run, some process *knows* a certificate σ that is valid for v, i.e., it maintains v and a valid certificate σ in its local memory. We require that the adversary is unable to invert VerifyInputValue by computing a valid certificate for a given value. This is the case, for example, when σ must contain a set of unforgeable digital signatures.

The Byzantine Lattice Agreement abstraction exports one operation and one function.[5]

- Operation Propose(v, σ) returns a response of the form $\langle w, \tau \rangle$, where $v, w \in \mathcal{L}$, σ is a valid certificate for input value v, and τ is a certificate for output value w;
- Function VerifyOutputValue(v, σ) returns a boolean value.

Similarly to input values, we say that τ is a *valid certificate for output value* w iff VerifyOutputValue$(w, \tau) = true$. We say that w is a *verifiable output value* in a given run iff at some point in that run, some process knows τ that is valid for w.

Implementations of Byzantine Lattice Agreement must satisfy the following properties:

- *BLA-Validity*: Every verifiable output value w is a join of some set of verifiable input values;
- *BLA-Verifiability*: If Propose(...) returns $\langle w, \tau \rangle$ to a correct process, then VerifyOutputValue$(w, \tau) = true$;
- *BLA-Inclusion*: If Propose(v, σ) returns $\langle w, \tau \rangle$ to a correct process, then $v \sqsubseteq w$;
- *BLA-Comparability*: All verifiable output values are comparable;
- *BLA-Liveness*: If the total number of verifiable input values is finite, every call to Propose(v, σ) by a forever-correct process eventually returns.

---

[4] Additional care is needed to prevent the "slow reader" attack. See [27] for more details.

[5] The main difference between an operation and a function is that a function can be computed without communicating with other processes and it always returns the same result given the same input.

For the sake of simplicity, we only guarantee liveness when there are finitely many verifiable input values. This is sufficient for the purposes of reconfiguration. The abstraction that provides unconditional liveness is called *Generalized Lattice Agreement* [16].

3.3 Reconfigurable objects

It is possible to define a *reconfigurable* version of every static distributed object by enriching its interface and imposing some additional properties. In this section, we define the notion of a reconfigurable object in a very abstract way. By combining this definition with the definition of a Byzantine Lattice Agreement from Section 3.2, we obtain a formal definition of a Reconfigurable Byzantine Lattice Agreement. Similar combination can be performed with the definition of any static distributed object (e.g., with the definition of a Max-Register from [27]).

A reconfigurable object exports an operation $\text{UpdateConfig}(C, \sigma)$, which can be used to reconfigure the system, and must be parameterized by a boolean function $\text{VerifyInputConfig} : \mathcal{C} \times \Sigma \to \{\mathit{true}, \mathit{false}\}$, where Σ is a set of possible certificates. As for verifiable input values, we say that $C \in \mathcal{C}$ is a *verifiable input configuration* in a given run iff at some point in that run, some process knows σ such that $\text{VerifyInputConfig}(C, \sigma) = \mathit{true}$.

We require the total number of verifiable input configurations to be finite in any given infinite execution of the protocol. In practice, this boils down to assuming sufficiently long periods of stability when no new verifiable input configurations appear. This requirement is imposed by all asynchronous reconfigurable storage systems [1, 34, 26, 3] we are aware of, and, in fact, can be shown to be necessary [33].

When a correct replica r is ready to serve user requests in configuration C, it triggers upcall $\text{InstalledConfig}(C)$. We then say that r *installs* configuration C. A configuration is called *installed* if some correct replica installed it. Finally, a configuration is called *superseded* if some higher configuration is installed.

Each reconfigurable object must satisfy the following properties:

- *Reconfiguration Validity*: Every installed configuration C is a join of some set of verifiable input configurations. Moreover, all installed configurations are comparable;
- *Reconfiguration Liveness*: Every call to $\text{UpdateConfig}(C, \sigma)$ by a forever-correct client eventually returns. Moreover, C or a higher configuration will eventually be installed.
- *Installation Liveness*: If some configuration C is installed by some correct replica, then C or a higher configuration will eventually be installed by all correct replicas.

3.4 Dynamic objects

Reconfigurable objects are hard to build because they need to solve two problems at once. First, they need to order and combine concurrent reconfiguration requests. Second, the state of the object needs to be transferred across installed configurations (we call this *state transfer*). We decouple these two problems by introducing the notion of a *dynamic* object. Dynamic objects solve the second problem while “outsourcing” the first one.

Before we formally define dynamic objects, let us first define the notion of a *history*. In Section 2, we introduced the configuration lattice $\mathcal{C}$. A finite set $h \subseteq \mathcal{C}$ is called a *history* iff all elements of h are comparable (in other words, if they form a sequence). Let $\text{HighestConf}(h)$ be $C \in h$ such that $\forall\ C' \in h : C' \sqsubseteq C$. By definition of a history, $\text{HighestConf}(h)$ is unambiguously defined for any history h.

Dynamic objects must export an operation $\text{UpdateHistory}(h, \sigma)$ and must be parameterized by a boolean function $\text{VerifyHistory} : \mathcal{H} \times \Sigma \to \{\mathit{true}, \mathit{false}\}$, where $\mathcal{H}$ is the set of all histories and Σ is the set of all possible certificates. We say that h is a *verifiable*

history in a given run iff at some point in that run, some process knows σ such that VerifyHistory$(h, \sigma) = \textit{true}$. A configuration C is called *candidate* iff it belongs to some verifiable history. Also, a candidate configuration C is called *active* iff it is not superseded by a higher configuration.

As with verifiable input configurations, the total number of verifiable histories is required to be finite. Additionally, we require all verifiable histories to be related by containment (i.e., comparable w.r.t. $\subseteq$). More formally, if VerifyHistory$(h_1, \sigma_1) = \textit{true}$ and VerifyHistory$(h_2, \sigma_2) = \textit{true}$, then $h_1 \subseteq h_2$ or $h_2 \subseteq h_1$. We discuss how to build such histories in Section 5.

Similarly to reconfigurable objects, a dynamic object must have the InstalledConfig(C) upcall. The object must satisfy the following properties:

- *Dynamic Validity*: Only a candidate configuration can be installed by a correct replica;
- *Dynamic Liveness*: Every call to UpdateHistory(h, σ) by a forever-correct client eventually returns. Moreover, HighestConf(h) or a higher configuration will eventually be installed;
- *Installation Liveness* (the same as for reconfigurable objects): If some configuration C is installed by some correct replica, then C or a higher configuration will eventually be installed by all correct replicas.

Note that Dynamic Validity implies that all installed configurations are comparable, since all verifiable histories are related by containment and all configurations within one history are comparable.

While reconfigurable objects provide general-purpose reconfiguration interface, dynamic objects are weaker, as they require an external service to build comparable verifiable histories. As the main contribution of this paper, we show how to build *dynamic* objects in a Byzantine environment and how to create *reconfigurable* objects using dynamic objects as building blocks. We argue that this technique is applicable to a large class of objects.

3.5 Quorum system assumptions

Most fault-tolerant implementations of distributed objects impose some requirements on the subsets of processes that can be faulty. We say that a configuration C is *available at time t* iff *replicas*(C) is a dissemination quorum system at time t (as defined in Section 2). Correctness of our implementations of *dynamic* objects relies on the assumption that active candidate configurations are available. Once a configuration is superseded by a higher configuration, we make no further assumptions about it.

For *reconfigurable* objects we impose a slightly more conservative requirement: every combination of verifiable input configurations that is not yet superseded must be available. More formally, let $C_1, \dots, C_k$ be verifiable input configurations such that $C = C_1 \sqcup \ldots \sqcup C_k$ is not superseded at time t. Then we require *quorums*(C) to be a dissemination quorum system at time t.

Correctness of our *reconfigurable* objects relies solely on correctness of the dynamic building blocks. Formally, when k configurations are concurrently proposed, we require all possible combinations, i.e., $2^k - 1$ configurations, to be available. However, in practice, at most k of them will be chosen to be put in verifiable histories, and only those configurations actually need to be available. We impose a more conservative requirement because we do not know these configurations *a priori*.

3.6 Broadcast primitives

To make sure that no slow process is "left behind", we assume that a variant of *reliable broadcast primitive* [13] is available. The primitive must ensure two properties: (1) If a forever-correct process p broadcasts a message m, then p eventually delivers m; (2) If some message m is delivered by a *forever-correct* process, every forever-correct process eventually delivers m. In practice such primitive can be implemented by some sort of a gossip protocol [24]. This primitive is "global" in a sense that it is not bound to any particular configuration. In pseudocode we use "**RB-Broadcast** $\langle ... \rangle$" to denote a call to the "global" reliable broadcast.

Additionally, we assume a "local" *uniform reliable broadcast* primitive [13]. It has a stronger totality property: if some *correct* process p delivered some message m, then every forever-correct process will eventually deliver m, even if p later turns Byzantine. This primitive can be implemented in a *static* system, provided a quorum system. As we deal with *dynamic* systems, we associate every broadcast message with a fixed configuration and only guarantee these properties if the configuration is *never superseded*. Notice that any static implementation of uniform reliable broadcast trivially guarantees this property. In pseudocode we use "**URB-Broadcast** $\langle ... \rangle$ **in** C" to denote a call to the "local" uniform reliable broadcast in configuration C.

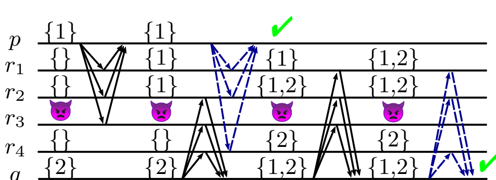
4 Dynamic Byzantine Lattice Agreement

Dynamic Byzantine Lattice Agreement abstraction (DBLA for short) is the main building block in our construction of reconfigurable objects. Its specification is a combination of the specification of Byzantine Lattice Agreement (Section 3.2) and the specification of a dynamic object (Section 3.4). The interface of a DBLA object is depicted in Figure 2a. To respect space limits, the complete pseudocode, the proof of correctness, and further discussion are presented in [27].

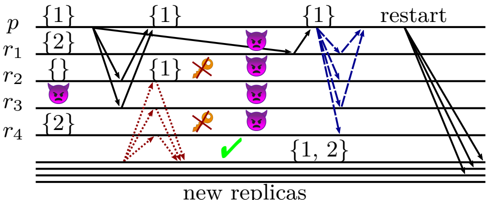
As we mentioned earlier, we use forward-secure digital signatures to guarantee that superseded configurations cannot affect correct clients or forge certificates for output values. Ideally, before a new configuration C is installed (i.e., before a correct replica triggers $\text{InstalledConfig}(C)$ upcall), we would like to make sure that the replicas of all candidate configurations lower than C invoke $\text{UpdateFSKey}(\mathit{height}(C))$. However, this would require the replica to know the set of all candidate configurations lower than C. Unambiguously agreeing on this set would require solving consensus, which is known to be impossible in an asynchronous system [17].

Instead, we classify all candidate configurations in two categories: *pivotal* and *tentative*. A candidate configuration is called *pivotal* if it is the last configuration in some verifiable history. Otherwise it is called *tentative*. A nice property of pivotal configurations is that it is impossible to "skip" one in a verifiable history. Indeed, if $C_1 = \text{HighestConf}(h_1)$ and $C_2 = \text{HighestConf}(h_2)$ and $C_1 \sqsubset C_2$, then, since all verifiable histories are related by containment, $h_1 \subseteq h_2$ and $C_1 \in h_2$. This allows us to make sure that, before a configuration C is installed, the replicas in all pivotal (and, possibly, some tentative) configurations lower than C update their keys.

In order to reconfigure a DBLA object, a correct client must use reliable broadcast to distribute the new verifiable history. Each correct process p maintains, locally, the largest (with respect to $\subseteq$) verifiable history it delivered so far through reliable broadcast. It is called *the local history of process* p and is denoted by $\mathit{history}_p$. We use $\mathit{Chighest}_p$ to denote the most recent configuration in p's local history (i.e., $\mathit{Chighest}_p = \text{HighestConf}(\mathit{history}_p)$). Whenever a replica r updates $\mathit{history}_r$, it invokes $\text{UpdateFSKey}(\mathit{height}(\mathit{Chighest}_r))$. Recall



(a) An example execution of two concurrent Propose operations.



(b) An example execution of a Propose operation concurrent with a reconfiguration.

Figure 1 Example executions of the DBLA protocol. Solid black arrows (resp., dashed blue arrows) correspond to the messages exchanged during the first (resp., the second) stage of the Propose protocol. Dotted red lines correspond to the messages exchanged during reconfiguration. The numbers represent the sets of verifiable input values known to the processes. Replica r_3 is Byzantine and always responds to **Propose** messages with the same set of verifiable input values as in the message itself. In **(b)**, replicas r_1, r_2, and r_4 also become Byzantine after the reconfiguration.

that if at least one forever-correct process delivers something via reliable broadcast, every other correct process will eventually deliver it as well.

Similarly, each process p keeps track of all verifiable input values it has seen $curVals_p \subseteq \mathcal{L} \times \Sigma$, where Σ is the set of all possible certificates. Sometimes, during the execution of the protocol, processes exchange these sets. Whenever a process p receives a message that contains a set of values with certificates $vs \subseteq \mathcal{L} \times \Sigma$, it checks that the certificates are valid (i.e., $\forall\ (v, \sigma) \in vs : \text{VerifyInputValue}(v, \sigma) = true$) and adds these values with certificates to $curVals_p$.

4.1 Client implementation

The client's protocol is simple. As we mentioned earlier, the operation UpdateHistory(h, σ) is implemented as **RB-Broadcast** $\langle$**NewHistory**$, h, \sigma\rangle$. The rest of the reconfiguration process is handled by the replicas. The protocol for the operation Propose(v, σ) consists of two stages: *proposing* a value and *confirming* the result.

The first stage (proposing) mostly follows the implementation of lattice agreement by Faleiro et al. [16]. Client p repeatedly sends message $\langle$**Propose**, $curVals_p$, $seqNum_r$, $C\rangle$ to all replicas in $replicas(C)$, where **Propose** is the message descriptor, $C = Chighest_p$, and $seqNum_r$ is a sequence number used by the client to match sent messages with replies.

After sending these messages to $replicas(C)$, the client waits for responses of the form $\langle$**ProposeResp**, vs, sig, $sn\rangle$, where **ProposeResp** is the message descriptor, vs is the set of all verifiable input values known to the replica with valid certificates (including those sent by the client), sig is a forward-secure signature with timestamp $height(C)$, and sn is the same sequence number as in the message from the client.

During the first stage, three things can happen: (1) the client learns about some new verifiable input values from one of the **ProposeResp** messages; (2) the client updates its local history (by delivering it through reliable broadcast); and (3) the client receives a quorum of valid replies with the same set of verifiable input values. In the latter case, the client combines the responses to form a certificate (called $acks_1$) and proceeds to the second stage. In the first two cases, the client simply restarts the operation. Because the number of verifiable input values, as well as the number of verifiable histories, are assumed to be finite, the number of restarts will also be finite.

The example in Figure 1a illustrates how the first stage of the algorithm ensures the comparability of the results when no reconfiguration is involved. In this example, clients p and q concurrently propose values $\{1\}$ and $\{2\}$, respectively, from the lattice $\mathcal{L} = 2^{\mathbb{N}}$. Client

p successfully returns the proposed value $\{1\}$ while client q is forced to refine its proposal and return the combined value $\{1, 2\}$. The quorum intersection prevents the clients from returning incomparable values (e.g., $\{1\}$ and $\{2\}$).

In the second (confirming) stage of the protocol, the client simply sends the acknowledgments it has collected in the first stage to the replicas of the same configuration. The client than waits for a quorum of replicas to reply with a forward-secure signature with timestamp $\mathit{height}(C)$.

The example in Figure 1b illustrates how reconfiguration can interfere with an ongoing Propose operation in what we call *the "slow reader" attack*, and how the second stage of the protocol prevents safety violations. In this example, client p should not be able to return the proposed value $\{1\}$ because all correct replicas in quorum $\{r_1, r_3, r_4\}$ store value $\{2\}$, which means that previously some other client could have returned value $\{2\}$. The client successfully reaches replicas r_2 and r_3 before the reconfiguration. None of them tell the client about the input value $\{2\}$, because r_2 is outdated and r_3 is Byzantine. The message from p to r_1 is delayed. Meanwhile, a new configuration is installed, and all replicas of the original configuration become Byzantine. If r_1 lies to p, the client may finish the first stage of the protocol with value $\{1\}$. However, because replicas r_2 and r_4 updated their private keys during the reconfiguration, they are unable to send the signed confirmations with timestamp $\mathit{height}(C)$ to the client. The client then waits until it receives the new verifiable history via reliable broadcast and restarts the operation in the new configuration.

The certificate for the output value $v \in \mathcal{L}$ produced by the Propose protocol in a configuration C consists of: (1) the set of verifiable input values (with certificates for them) from the first stage of the algorithm (the join of all these values must be equal to v); (2) a verifiable history (with a certificate for it) that confirms that C is a pivotal configuration; (3) the quorum of signatures from the first stage of the algorithm; and (4) the quorum of signatures from the second stage of the algorithm. Intuitively, the only way for a Byzantine client to obtain such a certificate is to benignly follow the Propose protocol.

4.2 Replica implementation

Each replica r maintains, locally, its *current configuration* (denoted by Ccurr_r) and *the last configuration installed by this replica* (denoted by Cinst_r). $\mathit{Cinst}_r \sqsubseteq \mathit{Ccurr}_r \sqsubseteq \mathit{Chighest}_r$. Intuitively, $\mathit{Ccurr}_r = C$ means that replica r knows that there is no need to transfer state from configurations lower than C, either because r already performed the state transfer from those configurations, or because it knows that sufficiently many other replicas did. $\mathit{Cinst}_r = C$ means that the replica knows that sufficiently many replicas in C have up-to-date states, and that configuration C is ready to serve user requests.

As we saw earlier, each client message is associated with some configuration C. The replica only answers the message when $C = \mathit{Cinst}_r = \mathit{Ccurr}_r = \mathit{Chighest}_r$. If $C \sqsubset \mathit{Chighest}_r$, the replica simply ignores the message. Due to the properties of reliable broadcast, the client will eventually learn about $\mathit{Chighest}_r$ and will repeat its request there (or in an even higher configuration). If $\mathit{Cinst}_r \sqsubset C$ and $\mathit{Chighest}_r \sqsubseteq C$, the replica waits until C is installed before processing the message. Finally, if C is incomparable with Cinst_r or $\mathit{Chighest}_r$, then the message is sent by a Byzantine process and the replica should ignore it.

When a correct replica r receives a **Propose** message, it adds the newly learned verifiable input values to $\mathit{curVals}_r$ and sends $\mathit{curVals}_r$ to the client with a forward-secure signature with timestamp $\mathit{height}(C)$. When a correct replica receives a **Confirm** message, it simply signs the set of acknowledgments in it with a forward-secure signature with timestamp $\mathit{height}(C)$ and sends the signature to the client.

Algorithm 1 DBLA state transfer, code for replica r.

```
1: upon Ccurr ≠ HighestConf({C ∈ history | r ∈ replicas(C)})
2:     let Cnext = HighestConf({C ∈ history | r ∈ replicas(C)})
3:     let S = {C ∈ history | Ccurr ⊑ C ⊏ Cnext}
4:     seqNum ← seqNum + 1
5:     for each C ∈ S do
6:         send ⟨UpdateRead, seqNum, C⟩ to replicas(C)
7:         wait for (C ⊏ Ccurr) ∨ (responses from any Q ∈ quorums(C) with s.n. seqNum)
8:     if Ccurr ⊏ Cnext then
9:         Ccurr ← Cnext
10:        URB-Broadcast ⟨UpdateComplete⟩ in Cnext
11: upon receive ⟨UpdateRead, sn, C⟩ from replica r′
12:     wait for C ⊏ HighestConf(history)
13:     send ⟨UpdateReadResp, curVals, sn⟩ to r′
14: upon receive ⟨UpdateReadResp, vs, sn⟩ from replica r′
15:     if VerifyInputValues(vs \ curVals) then curVals ← curVals ∪ vs
16: upon URB-deliver ⟨UpdateComplete⟩ in C from quorum Q ∈ quorums(C)
17:     wait for C ∈ history
18:     if Cinst ⊏ C then
19:         if Ccurr ⊏ C then Ccurr ← C
20:         Cinst ← C
21:         trigger upcall InstalledConfig(C)
22:         if r ∉ replicas(C) then halt
```

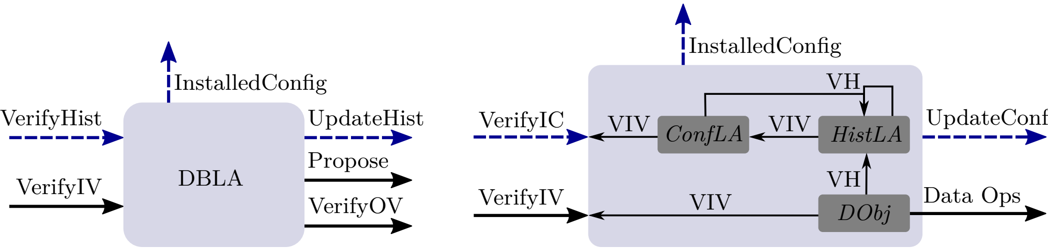
A very important part of the replica's implementation is *the state transfer protocol.* The pseudocode for it is presented in Algorithm 1. Note that we omit the subscript r in the pseudocode because each process can only access its own variables directly. Let $Cnext_r$ be the highest configuration in $history_r$ such that $r \in replicas(Cnext_r)$. Whenever $Ccurr_r \neq Cnext_r$, the replica tries to "move" to $Cnext_r$ by reading the current state from all configurations between $Ccurr_r$ and $Cnext_r$ one by one in ascending order (line 5). In order to read the current state from configuration $C \sqsubset Cnext_r$, replica r sends message $\langle$**UpdateRead**, $seqNum_r$, $C\rangle$ to all replicas in $replicas(C)$. In response, each replica $r_1 \in replicas(C)$ sends $curVals_{r_1}$ to r in an **UpdateReadResp** message (line 13). However, r_1 replies only after its private key is updated to a timestamp larger than $height(C)$ (line 12). We maintain the invariant that for any correct replica r': $st_{r'} = height(\text{HighestConf}(history_{r'}))$, where $st_{r'}$ is the timestamp of the private key of r'.

If r receives a quorum of replies from the replicas of C, there are two distinct cases:

- C is still active. In this case, the quorum intersection property still holds for C, and replica r can be sure that (1) if some Propose operation has either completed in configuration C or reached the second stage, $v \sqsubseteq \text{JoinAll}(curVals_r)$, where v is the value returned by the Propose operation and $\text{JoinAll}(curVals_r)$ is the join of all verifiable input values in the set $curVals_r$; and (2) if some Propose operation has not yet reached the second stage, it will not be able to complete in configuration C (see the example in Figure 1b).
- C is already superseded. In this case, by definition, a higher configuration is installed, and, intuitively, replica r will get the necessary state from that higher configuration.

It may happen that configuration C is already superseded and r will not receive sufficiently many replies from the replicas of C. However, in this case r will eventually discover that some higher configuration is installed, and it will update $Ccurr_r$ (line 19).

When a correct replica completes transferring the state to some configuration C, it notifies other replicas about it by broadcasting message **UpdateComplete** in configuration C (line 10). A correct replica *installs* a configuration C if it receives such messages from a



(a) The interface of DBLA.

(b) The dependencies in a reconfigurable object.

Figure 2 (a): The interface of a DBLA object. Parameters are depicted on the left side, operations and functions are on the right side, and upcalls are at the top. The parts of the interface that are inherited from BLA are depicted as black arrows, while the parts of the interface that are inherited from the specification of a dynamic object are depicted as dashed blue arrows. "IV", "OV", and "Hist" are abbreviations for "InputValue", "OutputValue", and "History", respectively. **(b):** The structure of dependencies in our implementation of a reconfigurable object. An arrow from an object A to another object B marked with VIV (resp., VH) indicates that A.VerifyInputValue (resp., A.VerifyHistory) is implemented using B.VerifyOutputValue.

quorum of replicas in C (line 16). Because we want our protocol to satisfy the Installation Liveness property (if one correct replica installs a configuration, every forever-correct replica must eventually install this or a higher configuration), the **UpdateComplete** messages are distributed through the uniform reliable broadcast primitive that we introduced in Section 3.6.

4.3 Implementing other dynamic objects

While we do not provide any general approach for building dynamic objects, we expect that most asynchronous Byzantine fault-tolerant static algorithms can be adopted to the dynamic case by applying the same set of techniques. These techniques include our state transfer protocol (relying on forward-secure signatures), the use of an additional round-trip to prevent the "slow reader" attack, and the structure of our cryptographic proofs ensuring that tentative configurations cannot create valid certificates for output values. To illustrate this, in the full version of this paper [27], we present the dynamic version of Max-Register [5] and discuss the dynamic version of the Access Control abstraction.

5 Implementing reconfigurable objects

While dynamic objects are important building blocks, they are not particularly useful by themselves because they require an external source of comparable verifiable histories. In this section, we show how to combine several dynamic objects to obtain a single *reconfigurable* object. Similar to dynamic objects, the specification of a reconfigurable object can be obtained as a combination of the specification of a static object with the specification of an abstract reconfigurable object from Section 3.3. In particular, compared to static objects, reconfigurable objects have one more operation – UpdateConfig(C, σ), must be parameterized by a boolean function VerifyInputConfig(C, σ), and must satisfy Reconfiguration Validity, Reconfiguration Liveness, and Installation Liveness.

We build a reconfigurable object by combining three *dynamic* ones. The first one is the dynamic object that we want to make reconfigurable (let us call it *DObj*). For example, it can be an instance of DBLA if we wanted to make a reconfigurable version of Byzantine

Lattice Agreement. The two remaining objects are used to build verifiable histories: *ConfLA* is a DBLA operating on the configuration lattice $\mathcal{C}$, and *HistLA* is a DBLA operating on the powerset lattice $2^{\mathcal{C}}$. The relationships between the three dynamic objects are depicted in Figure 2b.

Algorithm 2 Reconfigurable object (short version).

```
    ▷ Code for client p
23: Data operations are performed directly on DObj.

24: operation UpdateConfig(C, σ)
25:     let ⟨C', σ_C'⟩ = ConfLA.Propose(C, σ)
26:     let ⟨h, σ_h⟩ = HistLA.Propose({C'}, σ_C')
27:     DObj.UpdateHistory(h, σ_h)
28:     ConfLA.UpdateHistory(h, σ_h)
29:     HistLA.UpdateHistory(h, σ_h)

    ▷ Code for replica r
30: upon receive upcall InstalledConfig(C) from all ConfLA, HistLA, and DObj
31:     trigger upcall InstalledConfig(C)
```

The short version of the pseudocode is presented in Algorithm 2. All data operations are performed directly on *DObj*. To update a configuration, the client first submits its proposal to *ConfLA* and then submits the result as a singleton set to *HistLA*. Due to the BLA-Comparability property, all verifiable output values produced by *ConfLA* are comparable, and any combination of them would create a well-formed history as defined in Section 3.4. Moreover, the verifiable output values of *HistLA* are related by containment, and, therefore, can be used as verifiable histories in dynamic objects. We use them to reconfigure all three dynamic objects (lines 27–29).

The full pseudocode of the transformation with formal specification of all parameters, as well as the proof of correctness and the discussion of possible optimizations, are presented in the full version of this paper [27]. Additionally, in [27] we discuss several ways to prevent the Byzantine clients from reconfiguring the system in an undesirable way.

6 Related work

Dynamic replicated systems with *passive reconfiguration* [9, 7, 25] do not explicitly regulate arrivals and departures of replicas. Their consistency properties are ensured under strong assumptions on the churn rate. Except for the recent work [25], churn-tolerant storage systems do not tolerate Byzantine failures. In contrast, *active reconfiguration* allows the clients to explicitly propose configuration updates, e.g., sets of new replica arrivals and departures.

Early proposals of (actively) reconfigurable storage systems tolerating process crashes, such as RAMBO [22] and reconfigurable Paxos [28], used consensus (and, thus, assumed certain level of synchrony) to ensure that the clients agree on the evolution of configurations. DynaStore [2] was the first *asynchronous* reconfigurable storage: clients propose incremental additions or removals to the system configuration. As the proposals commute, the processes can resolve their disagreements without involving consensus.

The *parsimonious speculative snapshot* task [19] allows to resolve conflicts between concurrent configuration updates in a storage system using instances of commit-adopt [18]. The worst-case time complexity, in the number of message delays, of reconfiguration was later reduced from $O(n^2)$ to $O(n)$ [34], where n is the number of concurrently proposed configuration updates.

SmartMerge [23] made an important step forward by treating reconfiguration as an instance of abstract *lattice agreement* [16]. However, the algorithm assumes an external (reliable) lattice agreement service which makes the system not fully reconfigurable. The recently proposed *reconfigurable lattice-agreement* abstraction [26] enables truly reconfigurable versions of a large class of objects and constructions, including state-based CRDTs [32], atomic-snapshot, max-register, conflict detector and commit-adopt. We believe that the reconfiguration service we introduced in this paper can be used to derive Byzantine fault-tolerant reconfigurable implementations of objects in the class.

Byzantine quorum systems [29] introduce abstractions for ensuring availability and consistency of shared data in asynchronous systems with Byzantine faults. In particular, a *dissemination* quorum system ensures that every two quorums have a correct process in common and that at least one quorum only contains correct processes.

Dynamic Byzantine quorum systems [4] appear to be the first attempt to implement a form of active reconfiguration in a Byzantine fault-tolerant data service running on a *static* set of replicas, where clients can raise or lower the resilience threshold. Dynamic Byzantine storage [31] allows a trusted *administrator* to issue ordered reconfiguration calls that might also change the set of replicas. The administrator is also responsible for generating new private keys for the replicas in each new configuration to anticipate the "I still work here" attack [1]. In this paper, we propose an implementation of a Byzantine fault-tolerant reconfiguration service that does not rely on this assumption.

Forward-secure signature schemes [10, 11, 14, 15, 30] enable a decentralized way to construct a sequence of distinct private keys for each process. We use the scheme to provide each process with a unique private key for each configuration. To counter the "I still work here" attack, we ensure that sufficiently many correct processes destroy their configuration keys before a new configuration is installed, without relying on a global agreement of the configuration sequence [31].

7 Discussion

Communication cost. In this paper, we do not intend to provide the optimal implementations of each object or to implement the most general abstractions (such as generalized lattice agreement [16, 26]). Instead, we focused on providing the minimal implementation for the minimal set of abstractions to demonstrate the ideas and the general techniques for defining and building reconfigurable services in the harsh world of asynchrony and Byzantine failures. Therefore, our implementations leave plenty of space for optimizations. We discuss a few possible directions in the full version of this paper [27].

Open questions. We would like to mention two relevant directions for further research.

First, with regard to active reconfiguration, it would be interesting to devise algorithms that efficiently adapt to "small" configuration changes, while still supporting the option of completely changing the set of replicas in a single reconfiguration request. In this paper, we allow the sets of replicas of proposed configurations to be completely disjoint, which incurred an expensive quorum-to-quorum communication pattern. This might seem unnecessary for reconfigurations requests involving only slight changes of the set of replicas.

Second, with regard to Byzantine faults, it would be interesting to consider models with a "weaker" adversary. In this paper, we assumed a very strong model of the adversary: no assumptions are made about correctness of replicas in superseded configurations. This "pessimistic" approach leads to more complicated and expensive protocols.

References

1. Marcos K Aguilera, Idit Keidar, Dahlia Malkhi, Jean-Philippe Martin, Alexander Shraer, et al. Reconfiguring replicated atomic storage: A tutorial. *Bulletin of the EATCS*, 102:84–108, 2010.
2. Marcos Kawazoe Aguilera, Idit Keidar, Dahlia Malkhi, and Alexander Shraer. Dynamic atomic storage without consensus. *J. ACM*, 58(2):7:1–7:32, 2011.
3. Eduardo Alchieri, Alysson Bessani, Fabíola Greve, and Joni da Silva Fraga. Efficient and modular consensus-free reconfiguration for fault-tolerant storage. In *OPODIS*, pages 26:1–26:17, 2017.
4. Lorenzo Alvisi, Dahlia Malkhi, Evelyn Pierce, Michael K Reiter, and Rebecca N Wright. Dynamic byzantine quorum systems. In *Proceeding International Conference on Dependable Systems and Networks. DSN 2000*, pages 283–292. IEEE, 2000.
5. James Aspnes, Hagit Attiya, and Keren Censor. Max registers, counters, and monotone circuits. In *PODC*, pages 36–45, 2009.
6. 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.
7. Hagit Attiya, Hyun Chul Chung, Faith Ellen, Saptaparni Kumar, and Jennifer L. Welch. Emulating a shared register in a system that never stops changing. *IEEE Trans. Parallel Distrib. Syst.*, 30(3):544–559, 2019.
8. Hagit Attiya, Maurice Herlihy, and Ophir Rachman. Atomic snapshots using lattice agreement. *Distributed Comput.*, 8(3):121–132, 1995.
9. Roberto Baldoni, Silvia Bonomi, Anne-Marie Kermarrec, and Michel Raynal. Implementing a register in a dynamic distributed system. In *ICDCS*, pages 639–647, 2009.
10. Mihir Bellare and Sara K Miner. A forward-secure digital signature scheme. In *Annual International Cryptology Conference*, pages 431–448. Springer, 1999.
11. Xavier Boyen, Hovav Shacham, Emily Shen, and Brent Waters. Forward-secure signatures with untrusted update. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 191–200, 2006.
12. Eric A. Brewer. Towards robust distributed systems (abstract). In *PODC*, pages 7–, 2000.
13. Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to reliable and secure distributed programming*. Springer Science & Business Media, 2011.
14. Ran Canetti, Shai Halevi, and Jonathan Katz. A forward-secure public-key encryption scheme. *Journal of Cryptology*, 20(3):265–294, 2007.
15. Manu Drijvers, Sergey Gorbunov, Gregory Neven, and Hoeteck Wee. Pixel: Multi-signatures for consensus. In *29th USENIX Security Symposium (USENIX Security 20)*, Boston, MA, August 2020. USENIX Association. URL: `https://www.usenix.org/conference/usenixsecurity20/presentation/drijvers`.
16. Jose Faleiro, Sriram Rajamani, Kaushik Rajan, Ganesan Ramalingam, and Kapil Vaswani. Generalized lattice agreement. In *PODC*, pages 125–134, 2012.
17. Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.
18. Eli Gafni. Round-by-round fault detectors: Unifying synchrony and asynchrony. In *PODC*, pages 143–152, 1998.
19. Eli Gafni and Dahlia Malkhi. Elastic configuration maintenance via a parsimonious speculating snapshot solution. In *DISC*, pages 140–153, 2015.
20. David K. Gifford. Weighted voting for replicated data. In *SOSP*, pages 150–162, 1979.
21. Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.
22. Seth Gilbert, Nancy A Lynch, and Alexander A Shvartsman. Rambo: a robust, reconfigurable atomic memory service for dynamic networks. *Distributed Computing*, 23(4):225–272, 2010.
23. Leander Jehl, Roman Vitenberg, and Hein Meling. Smartmerge: A new approach to reconfiguration for atomic storage. In *DISC*, pages 154–169, 2015.

24 Anne-Marie Kermarrec and Maarten Van Steen. Gossiping in distributed systems. *ACM SIGOPS operating systems review*, 41(5):2–7, 2007.
25 Saptaparni Kumar and Jennifer L. Welch. Byzantine-tolerant register in a system with continuous churn. *CoRR*, abs/1910.06716, 2019. `arXiv:1910.06716`.
26 Petr Kuznetsov, Thibault Rieutord, and Sara Tucci-Piergiovanni. Reconfigurable lattice agreement and applications. In *OPODIS*, 2019.
27 Petr Kuznetsov and Andrei Tonkikh. Asynchronous reconfiguration with byzantine failures. *CoRR*, abs/2005.13499, 2020. URL: `https://arxiv.org/abs/2005.13499`.
28 Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Reconfiguring a state machine. *SIGACT News*, 41(1):63–73, 2010.
29 Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. *Distributed computing*, 11(4):203–213, 1998.
30 Tal Malkin, Daniele Micciancio, and Sara Miner. Efficient generic forward-secure signatures with an unbounded number of time periods. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 400–417. Springer, 2002.
31 J-P Martin and Lorenzo Alvisi. A framework for dynamic byzantine storage. In *International Conference on Dependable Systems and Networks, 2004*, pages 325–334. IEEE, 2004.
32 Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *SSS*, pages 386–400, 2011.
33 Alexander Spiegelman and Idit Keidar. On liveness of dynamic storage. In *Structural Information and Communication Complexity - 24th International Colloquium, SIROCCO 2017, Porquerolles, France, June 19-22, 2017, Revised Selected Papers*, pages 356–376, 2017.
34 Alexander Spiegelman, Idit Keidar, and Dahlia Malkhi. Dynamic reconfiguration: Abstraction and optimal asynchronous solution. In *DISC*, pages 40:1–40:15, 2017.

Improved Extension Protocols for Byzantine Broadcast and Agreement

Kartik Nayak
Duke University, Durham, NC, USA
kartik@cs.duke.edu

Ling Ren
University of Illinois at Urbana-Champaign, Champaign, IL, USA
renling@illinois.edu

Elaine Shi
Cornell University, Ithaca, NY, USA
runting@gmail.com

Nitin H. Vaidya
Georgetown University, Washington, D.C., USA
nitin.vaidya@georgetown.edu

Zhuolun Xiang
University of Illinois at Urbana-Champaign, Champaign, IL, USA
xiangzl@illinois.edu

Abstract

Byzantine broadcast (BB) and Byzantine agreement (BA) are two most fundamental problems and essential building blocks in distributed computing, and improving their efficiency is of interest to both theoreticians and practitioners. In this paper, we study extension protocols of BB and BA, i.e., protocols that solve BB/BA with long inputs of l bits using lower costs than l single-bit instances. We present new protocols with improved communication complexity in almost all settings: authenticated BA/BB with $t < n/2$, authenticated BB with $t < (1-\epsilon)n$, unauthenticated BA/BB with $t < n/3$, and asynchronous reliable broadcast and BA with $t < n/3$. The new protocols are advantageous and significant in several aspects. First, they achieve the best-possible communication complexity of $\Theta(nl)$ for wider ranges of input sizes compared to prior results. Second, the authenticated extension protocols achieve optimal communication complexity given the current best available BB/BA protocols for short messages. Third, to the best of our knowledge, our asynchronous and authenticated protocols in the setting are the first extension protocols in that setting.

2012 ACM Subject Classification Theory of computation → Communication complexity; Theory of computation → Cryptographic protocols

Keywords and phrases Byzantine agreement, Byzantine broadcast, extension protocol, communication complexity

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.28

Related Version A full version of the paper is available at `https://arxiv.org/abs/2002.11321`.

Funding Kartik Nayak and Ling Ren are supported in part by a VMware early career faculty grant.
Elaine Shi: Elaine Shi is supported in part by NSF award 1561209, and part of the work was done when the author was a long-terms visitor in Simons Institute for the "Proofs, Consensus, and Decentralizing Societ" program.
Nitin H. Vaidya: Nitin H. Vaidya is supported in part by NSF award 1849599.

© Kartik Nayak, Ling Ren, Elaine Shi, Nitin H. Vaidya, and Zhuolun Xiang;
licensed under Creative Commons License CC-BY

Table 1 Cryptographically Secure Extension Protocols for Byzantine Agreement and Broadcast.

Threshold	Model	Problem	Communication Complexity	Input range l to reach optimality	Reference
$t < n/2$	sync.	agreement/broadcast	$O(nl + n\mathcal{B}(k) + kn^3)$ $O(nl + \mathcal{A}(k) + kn^2)$ [2]	$\Omega(n^3 + kn^2)$ $\Omega(n^2 + kn)$	[15, 16] **This paper**
$t < n$	sync.	broadcast	$O(nl + \mathcal{B}(nk) + n^2\mathcal{B}(n \log n))$	$\Omega(n^5 \log n + kn^4 \log n)$	[15, 16]
$t < (1-\varepsilon)n$	sync.	broadcast	$O(nl + \mathcal{B}(k) + kn^2 + n^3)$	$\Omega(n^2 + kn)$	**This paper**
$t < n/3$	async.	agreement reliable broadcast	$O(nl + \mathcal{A}(k) + kn^2)$ $O(nl + \mathcal{B}(k) + kn^2)$	$\Omega(kn)$ $\Omega(kn)$	**This paper** **This paper**

1 Introduction

This paper investigates extension protocols [15] for Byzantine broadcast (BB) and Byzantine agreement (BA). The goal of BB is for some designated party (sender) to send its message to all parties and let them output the same message, despite some malicious parties that may behave in a Byzantine fashion. The goal of BA is to let all parties each with an input message output the same message. We are interested in designing efficient BB/BA protocols with *long messages* since such protocols are widely used as building blocks for other distributed systems such as multi-party computation [32] and permissioned blockchain [24]. For example, practical blockchain systems typically achieve agreement on large blocks (e.g., 1MB).

A straightforward solution for BB/BA with l-bit long messages is to invoke the single-bit BB/BA oracle l times. This approach will incur at least $\Omega(n^2 l)$ communication complexity where n is the number of parties, because any deterministic single-bit BB/BA has cost $\Omega(n^2)$ due to a lower bound in [10]. Another tempting solution is to run BB/BA on the hash digest and let parties disseminate the actual message to each other. However, if a linear fraction of parties can be Byzantine (which is the typical assumption), they can each ask all honest parties for the long message, again forcing the communication complexity to be $\Omega(n^2 l)$.

It turns out that non-trivial techniques are needed to get better than $\Omega(n^2 l)$ or to achieve the optimal communication complexity of $O(nl)$. These are known in the literature as *extension protocols*, which construct BB/BA with long input messages using a small number of BB/BA primitives for short messages. In this paper, we focus on the authenticated setting where cryptographic techniques are used. Table 1 summarizes the most closely related works and our new results on authenticated extension protocols. (In the full version, we also present some improvements to unauthenticated extension protocols.) In Table 1, n is the number of parties, t is the maximum number of Byzantine parties, l is the length of the input, $\mathcal{A}(l)$ is the communication cost of l-bit BA oracle, and $\mathcal{B}(l)$ is the communication cost of l-bit BB or reliable broadcast oracle. Here we describe the related works in the table. Let k_h denote output size of the collision-resistant hash function. For both Byzantine broadcast and agreement in the synchronous setting under $t < n/2$, recent work proposes cryptographically secure extension protocols with communication cost $O(nl + n\mathcal{B}(k_h) + n^3 k_h)$ [15, 16]. For the case of $t < n$, the state-of-the-art cryptographically secure BB extension protocols have communication complexity $O(nl + \mathcal{B}(nk_h) + n^2\mathcal{B}(n \log n))$ [15, 16]. There exist information-theoretic authenticated protocols [14, 8] but they have worse communication complexity than cryptographic ones. To the best of our knowledge, there exist no extension protocols in the authenticated and asynchronous setting when the paper is written [1].

[1] A concurrent work [21] independently developed an extension protocol for validated Byzantine agreement in the authenticated and asynchronous setting.

[2] Our cryptographic BB extension protocol can achieve $O(nl + \mathcal{B}(k) + \mathcal{A}(1) + kn^2)$ in the full version.

Contributions. Table 1 also presents our improved protocols in the respective settings. Several cryptographic primitives have been employed in our work and prior works. To make the communication costs comparable, we assume that the output length of the involved cryptographic building blocks are on the same order, and are all represented by k. We will justify this decision in Section 3.

All our protocols achieve the optimal communication complexity $O(nl)$ for wider ranges of input sizes (see Table 1 above for authenticated protocols and the full version for unauthenticated protocols). In particular, our synchronous and authenticated protocols achieve $O(nl)$ communication complexity when the input size is at least $l = \Omega(n^2 + kn)$. For comparison, state-of-art protocol in the literature require a factor of n larger input size for the $t < n/2$ case, and a factor of $n^3 \log n$ larger input size for $n/2 \leq t < (1-\varepsilon)n$ where ε is a constant. But a limitation of our protocol is that it cannot achieve $O(nl)$ communication if $\varepsilon = o(1)$. As for the round complexity, all our extension protocols only adds $O(1)$ communication rounds, except the one for $t < n/2$ which adds $O(t)$ rounds. All our protocols only invoke the BB/BA oracle $O(1)$ times.

In addition to reaching optimality under smaller input size, our authenticated extension protocols have the following advantages.

- The communication complexity of our BA extension protocols is very close to the lower bound $\Omega(nl + \mathcal{A}(k) + n^2)$. In addition, under the current best BA primitives for short messages, they achieve best-possible communication complexity. In order to improve upon our extension protocols, one must invent BA primitives for short messages with cost $o(kn^2)$, which seems challenging as we discuss in Section 4.3.
- Our protocols can be easily adapted to the asynchronous setting. To the best of our knowledge, these are the first asynchronous authenticated extension protocols. [3]
- Their simplicity makes our protocols less error-prone and more appealing for practical adoption. On this note, in deriving our results, we discover a flaw in the prior best protocol [15, 16] and we provide a simple fix in the full version of this paper.

2 Related Work

Timing and setup assumptions. With different security assumptions on the adversary and timing assumptions, Byzantine broadcast and agreement can be solved for different thresholds of the Byzantine parties. For the timing assumptions, protocols under both synchrony and asynchrony have been studied. If a trusted setup like public-key infrastructure (PKI) exists, it is called the authenticated setting; otherwise, it is the unauthenticated setting. In the synchronous setting, BB/BA can be solved under $t < n/3$ without authentication [19]; with authentication, BA can be solved under $t < n/2$ and BB can be solved under $t < n$ [19, 11, 29]. In the asynchronous setting, BB is impossible; BA (randomized) and reliable broadcast can be solved under $t < n/3$ with or without authentication [5, 6].

Previous extension protocols. Table 1 summarizes the two most closely related works on authenticated extension protocols. Here, we mention several other ones. Cachin and Tessaro [7] adapt Bracha's broadcast [5] to handle l-bit long messages with communication cost $O(nl + k_h n^2 \log n)$. Their method partially inspired our work; but their method does

[3] Asynchronous unauthenticated protocol exist and they can be used in the authenticated setting, but the cost would be much higher than our new protocols (refer to the full version of this paper.)

not seem to apply to general protocols and hence does not yield an extension protocol. Related unauthenticated extension protocols are summarized in the full version. Liang and Vaidya [20] propose the first optimal error-free BB and BA with communication complexity $O(nl + (n^2\sqrt{l} + n^4)\mathcal{B}(1))$ for the synchronous case. Patra [28] improves the communication complexity to $O(nl+n^2\mathcal{B}(1))$ under synchrony and also extended the protocols to asynchrony with increased communication complexity.

State-of-the-art oracle schemes. To better interpret the improvements we obtained for extension protocols, we provide a summary of the state-of-the-art broadcast and agreement protocols that can be used as the oracle in our extension protocol. Since our extension protocols are all deterministic, we focus on *deterministic* solutions for the most part of the paper, except for asynchronous BA where randomization is necessary. The best deterministic solution to authenticated BB for $t < n$ is the classic Dolev-Strong [11] protocol. After applying multi-signatures, the communication complexity to broadcast k bits is $\mathcal{B}(k) = \Theta((k + k_s)n^2 + n^3)$ where k_s is the signature size. The Dolev-Strong protocol can also be modified to solve authenticated BA for the $t < n/2$ case (BA is impossible if $t \geq n/2$). Using an initial all-to-all round with multi-signature to simulate the sender, the communication complexity remains as $\mathcal{A}(k) = \Theta((k+k_s)n^2+n^3)$. In the unauthenticated setting, only $t < n/3$ Byzantine parties can be tolerated and Berman et al. [3] achieves $\mathcal{B}(1) = \mathcal{A}(1) = \Theta(n^2)$ (when $t = \Theta(n)$), matching the lower bound on communication complexity.

In the asynchronous setting, Bracha's reliable broadcast [5] is deterministic and has communication complexity $\mathcal{B}(1) = O(n^2)$. Randomization is necessary for asynchronous BA given the FLP impossibility [13]. State-of-art protocols rely on "common coins" to provide shared randomness but are deterministic otherwise. The most efficient unauthenticated asynchronous BA [25] achieves expected communication complexity $\mathcal{A}(1) = O(n^2)$ assuming a common coin oracle. The most efficient authenticated asynchronous BA [1] achieves expected communication complexity $\mathcal{A}(k) = O((k + k_s)n^2)$ and provides a construction for the common coin oracle.

Coding schemes in consensus systems. Several works have taken advantage of coding schemes in practical fault-tolerant consensus systems. HoneyBadgerBFT [24] and BEAT [12] use the reliable broadcast proposed by Cachin and Tessaro [7] as a component for broadcasting blocks efficiently. Recent works also apply erasure coding to crash-tolerant systems like Paxos [26] and Raft [31].

3 Preliminaries

We consider n parties $P_1, ..., P_n$ connected by a reliable, authenticated all-to-all network, where up to t parties may be corrupted by an adversary A and behave in a Byzantine fashion. We consider both the synchronous model, where there exists a known upper bound on the communication and computation delay, and the asynchronous model, where such an upper bound does not exist. We consider a *static* adversary which decides the set of corrupted parties at the beginning of the execution. We denote parties that are not corrupted by the adversary as honest parties. Two types of the adversary are considered: a computationally bounded adversary is considered in cryptographically secure protocols and a computationally unbounded adversary is considered in the error-free protocols. Our cryptographically secure protocols additionally assume a trusted setup for a public key infrastructure (PKI) and cryptographic accumulators (see Section 3.1). The *communication complexity* [33] of the

protocol is measured by the worst-case or expected number of bits transmitted by the honest parties according to the protocol specification over all possible executions under any adversary strategy. Here, we provide the formal definition of Byzantine broadcast (BB) and Byzantine agreement (BA).

▶ **Definition 1** (Byzantine Broadcast). *A protocol for a set of parties $\mathcal{P} = \{P_1, ..., P_n\}$, where a distinguished party called the sender $P_s \in \mathcal{P}$ holds an initial l-bit input m, is a Byzantine broadcast protocol tolerating an adversary A, if the following properties hold*

- *Termination. Every honest party outputs a message.*
- *Agreement. All the honest parties output the same message.*
- *Validity. If the sender is honest, all honest parties output the message m.*

▶ **Definition 2** (Byzantine Agreement). *A protocol for a set of parties $\mathcal{P} = \{P_1, ..., P_n\}$, where each party $P_i \in \mathcal{P}$ holds an initial l-bit input m_i, is a Byzantine agreement protocol tolerating an adversary A, if the following properties hold*

- *Termination. Every honest party outputs a message.*
- *Agreement. All the honest parties output the same message.*
- *Validity. If every honest party P_i holds the same input message m, then all honest parties output the message m.*

For cryptographically secure protocols and randomized protocols, the above properties hold except for a negligible probability in the security parameter. For brevity, our theorem statements will not mention this explicitly.

3.1 Primitives

In this section, we define several primitives that will be used in our extension protocols. Our extension protocols use standard coding and cryptographic schemes from the literature, such as linear error correcting codes, muti-signature schemes and cryptographic accumulators.

Linear error correcting code [30]. We will use standard Reed-Solomon (RS) codes [30] in our protocols, which is a (n, b) RS code in Galois Field $\mathbb{F} = GF(2^a)$ with $n \leq 2^a - 1$. This code encodes b data symbols from $GF(2^a)$ into codewords of n symbols from $GF(2^a)$, and can decode the codewords to recover the original data.

- $\texttt{ENC}$. Given inputs $m_1, ..., m_b$, an encoding function $\texttt{ENC}$ computes $(s_1, ..., s_n) = \texttt{ENC}(m_1, ..., m_b)$, where $(s_1, ..., s_n)$ are codewords of length n. By the property of the RS code, knowledge of any b elements of the codeword uniquely determines the input message and the remaining of the codeword.
- $\texttt{DEC}$. The function $\texttt{DEC}$ computes $(m_1, ..., m_b) = \texttt{DEC}(s_1, ..., s_n)$, and is capable of tolerating up to c errors and d erasures in codewords $(s_1, ..., s_n)$, if and only if $n - b \geq 2c + d$. In our protocol, We will invoke $\texttt{DEC}$ with specific values of c, d satisfying the above relation, and $\texttt{DEC}$ will return correct output.

In our extension protocols, we will use the above RS codes with n equal the number of all parties, and b equal the number of honest parties, i.e., $b = n - t$.

Multi-signatures [4]. Multi-signature scheme can aggregate n signatures into one signature, therefore reduce the size of signatures. Given n signatures $\sigma_i = \texttt{Sign}(sk_i, m)$ on the same message m with corresponding public keys pk_i for $1 \leq i \leq n$, a multi-signature scheme can combine the n signatures above into one signature Σ where $|\Sigma| = |\sigma_i|$. The combined signature can be verified by anyone using a verification function $\texttt{Ver}(PK, \Sigma, m, \mathcal{L})$, where $\mathcal{L}$ is the list of signers and PK is the union of n public keys pk_i.

Cryptographic accumulators [2, 9]. We present the definition of *cryptographic accumulators* proposed by Barić and Pfitzmann [2]. Intuitively, the cryptographic accumulator constructs an accumulation value for a set of values and can produce a witness for each value in the set. Given the accumulation value and a witness, any party can verify if a value is indeed in the set. Formally, given a parameter k, and a set $\mathcal{D}$ of n values $d_1, ..., d_n$, an accumulator has the following components:

- $\texttt{Gen}(1^k, n)$: This algorithm takes a parameter k represented in unary form 1^k and an accumulation threshold n (an upper bound on the number of values that can be accumulated securely), returns an accumulator key ak. This step is run by a trusted dealer, so the accumulator key ak is known to all parties.
- $\texttt{Eval}(ak, \mathcal{D})$: This algorithm takes an accumulator key ak and a set $\mathcal{D}$ of values to be accumulated, returns an accumulation value z for the value set $\mathcal{D}$.
- $\texttt{CreateWit}(ak, z, d_i)$: This algorithm takes an accumulator key ak, an accumulation value z for $\mathcal{D}$ and a value d_i, returns $\perp$ if $d_i \notin \mathcal{D}$, and a witness w_i if $d_i \in \mathcal{D}$.
- $\texttt{Verify}(ak, z, w_i, d_i)$: This algorithm takes an accumulator key ak, an accumulation value z for $\mathcal{D}$, a witness w_i and a value d_i, returns *true* if w_i is the witness for $d_i \in \mathcal{D}$, and *false* otherwise.

For simplicity, our definition of the cryptographic accumulator above omits the auxiliary information aux that appears in the standard definition [2] because the bilinear accumulator we will use does not use aux. We also assume that the function `Eval` is *deterministic*, which is the case with the bilinear accumulator. We give the detailed description of the *bilinear accumulator* [27, 17] in the full version. The bilinear accumulator satisfies the following property.

▶ **Lemma 3** (Collision-free accumulator [27])**.** *The bilinear accumulator is collision-free. That is, for any set size n and any probabilistic polynomial-time adversary $\mathcal{A}$, there exists a negligible function* $\mathsf{negl}(\cdot)$*, such that*

$$\Pr\left[\begin{array}{l} ak \leftarrow \texttt{Gen}(1^k, n), (\{d_1, ..., d_n\}, d', w') \leftarrow \mathcal{A}(1^k, n, ak), z \leftarrow \texttt{Eval}(ak, \{d_1, ..., d_n\}) : \\ (d' \notin \{d_1, ..., d_n\}) \wedge (\texttt{Verify}(ak, z, w', d') = true) \end{array}\right] \leq \mathsf{negl}(k)$$

To better understand the definition of the cryptographic accumulator, it is helpful to note that the Merkle tree [23] is a cryptographic accumulator, where the accumulator key ak is the hash function, the accumulation value z is the Merkle tree root, and the witness w is the Merkle tree proof. We will use the *bilinear accumulator* [27, 17] instead of Merkle tree in our protocols, since the witness size of the Merkle tree is logarithmic in the number of values whereas the witness size of the bilinear accumulator is a constant. On the other hand, the bilinear accumulator requires a trusted dealer, which is a stronger trust assumption than public key infrastructure (PKI). The trusted dealer needs to know an upper bound on $|\mathcal{D}|$, i.e., the number of items accumulated (see the construction in the full version). In our protocols, $|\mathcal{D}|$ always equals the number of parties n. Hence, the trusted setup (both the PKI and the accumulator) can be reused across invocations if the parties participating in the extension protocol do not change. If a trusted dealer for accumulators cannot be assumed, our protocol can use Merkle tree as the accumulator; in that case, the $O(kn^2)$ term in the communication complexity becomes $O(kn^2 \log n)$ and our protocol still has an advantage (albeit smaller) over prior art.

Normalizing the length of cryptographic building blocks. Let λ denote the security parameter, $k_h = k_h(\lambda)$ denote the hash size, $k_s = k_s(\lambda)$ denote the (multi-)signature size, $k_a = k_a(\lambda)$ denote the size of the accumulation value and witness of the accumulator. Further

let $k = \max(k_h, k_s, k_a)$; we assume $k = \Theta(k_h) = \Theta(k_s) = \Theta(k_a) = \Theta(\lambda)$ This assumption is reasonable since the signature scheme and accumulator scheme with the shortest output length are both based on pairing-friendly curves, which are believed to require $\Theta(\lambda)$ bits for λ-bit security given the state-of-the-art attack [18]. As for hash functions, it is common to model them as random oracles, in which case λ-bit security requires $\Theta(\lambda)$-bit hash size. Therefore, throughout the paper, we can use the same variable k to denote the hash size, signature size and accumulator size for convenience.

4 Cryptographically Secure Extension Protocols under $t < n/2$ Faults

In this section, we design cryptographically secure extension protocols with improved communication complexity for the synchronous and authenticated setting with $t < n/2$ faults. We start by presenting some building blocks that will be frequently used in all our authenticated protocols. Then, we give an extension protocol for synchronous BA with communication complexity $O(nl + \mathcal{A}(k) + kn^2)$. Under synchrony, this also implies a BB protocol with $t < n/2$ and the same communication complexity, by first having the sender send the message to all parties and then performing a Byzantine agreement [22]. In the full version, we show another extension protocol for $t < n/2$ BB with communication complexity $O(nl + \mathcal{B}(k) + \mathcal{A}(1) + kn^2)$. The protocols are adapted to the asynchronous case in Section 6. At the end of this section, we discuss the small gap between our BA protocol and a simple lower bound on BA with long messages.

4.1 Building Blocks: `Encode`, `Distribute` and `Reconstruct`

We first define three subprotocols `Encode`, `Distribute` and `Reconstruct` that will be used as building blocks for our cryptographically secure extension protocols, listed in Figure 1.

- `Encode` first divides a message m into b blocks, then compute n coded values $(s_1, ..., s_n)$ using RS codes (defined in Section 3), and attaches an index j for each value s_j. The purpose of `Encode` is to introduce resilience by encoding the message into fault-tolerant coded values – after applying `Encode` to a message m, even if $n - b$ coded values in $(s_1, ..., s_n)$ are erased, one can recover the message from the remaining coded values.
- `Distribute` computes a witness w_j for each indexed value $\langle j, s_j \rangle$ in the input set, and sends the j-th value with its witness to party j. The purpose of `Distribute` is to distribute the values in a robust yet efficient manner – if at least one honest party that has the correct message m (the accumulation value z of m is correct) invokes `Distribute`, then it is guaranteed that any honest party j receives and accepts the j-th value s_j of m, thanks to the witness w_j sent together with the value.
- `Reconstruct` first removes any invalid value s_j that cannot be verified by witness w_j and the accumulation value z, and then decode the message m using RS code (defined in Section 3) from the remaining values with at most d_0 values being removed. The purpose of `Reconstruct` is to recover the message, despite the presence of at most d_0 corruptions in the value, which will be detected by the accumulator scheme and thus erased.

Our extension protocols in Sections 4 and 5 will use `Encode` at the beginning of the protocol to encode the input message to coded values, use `Distribute` in the middle to let every party distribute their coded values with the witnesses, and use `Reconstruct` to reconstruct the original input message after receiving the coded values.

- **Encode(m, b)**
  Input: a message m, a number b
  Output: n coded values $s_1, ..., s_n$
  Divide m into b blocks evenly, $m_1, ..., m_b$, each has l/b bits where l is the length of m. Compute $(s_1, ..., s_n) = \texttt{ENC}(m_1, ..., m_b)$ using RS codes, where ENC is defined in Section 3.1. Add an index to every value in $(s_1, ..., s_n)$, i.e., $\mathcal{D} = (\langle 1, s_1\rangle, ..., \langle n, s_n\rangle)$, and return $\mathcal{D}$.
- **Distribute($\mathcal{D}, ak, z$)**
  Input: a set of indexed values $\mathcal{D} = (\langle 1, s_1\rangle, ..., \langle n, s_n\rangle)$, an accumulator key ak, an accumulation value z
  Compute $w_j = \texttt{CreateWit}(ak, z, \langle j, s_j\rangle)$ for every $\langle j, s_j\rangle \in \mathcal{D}$. Send (s_j, w_j) to party P_j for every $j \in [n]$.
- **Reconstruct($\mathcal{S}, ak, z, d_0$)**
  Input: $\mathcal{S} = ((\langle 1, s_1\rangle, w_1), ..., (\langle n, s_n\rangle, w_n))$ where each $(\langle i, s_i\rangle, w_i)$ is a pair of indexed value and witness, an accumulator key ak, an accumulation value z, a number d_0
  Output: a message m
  For every $j \in [n]$, if $\texttt{Verify}(ak, z', w_j, \langle j, s_j\rangle) = \text{false}$, let $s_j = \bot$. Apply DEC on the codewords $(s_1, ..., s_n)$ with $c = 0$ and $d = d_0$, where DEC is defined in Section 3.1. Return $m = m_1|...|m_b$ where $m_1, ..., m_b$ are the data returned by DEC.

Figure 1 Building Blocks.

▶ **Lemma 4.** *For any message m, let $z = \textit{Eval}(ak, \textit{Encode}(m, b))$. The adversary cannot find $m' \neq m$ such that $z = \textit{Eval}(ak, \textit{Encode}(m', b))$ except for negligible probability in k.*

Proof. Let $\mathcal{D} = \texttt{Encode}(m, b)$ and $\mathcal{D}' = \texttt{Encode}(m', b)$. By the RS code, the same codewords correspond to the same message. Thus, if $m \neq m'$, we have $\mathcal{D} \neq \mathcal{D}'$, i.e., there exists $d' = \langle i, s_i\rangle$ such that $d \in \mathcal{D}$ and $d' \notin \mathcal{D}'$. However, under the accumulation value $z = \texttt{Eval}(ak, \mathcal{D}'))$, a witness for $d = \langle i, s_i\rangle \notin \mathcal{D}'$ exists. Due to Lemma 3, this happens with negligible in k probability. ◀

4.2 Byzantine Agreement under $< \frac{n}{2}$ faults

The protocol Synchronous Crypto. $\frac{n}{2}$-BA is presented in Figure 2. In the protocol, let t denote the maximum number of Byzantine parties, and let $b = n - t$. We briefly describe each step of the protocol. First, each party encodes its message using RS codes and computes the accumulation value for the set of coded values. With a deterministic Eval, any honest party with the same accumulator key and set will produce the same accumulation value. The RS codes can recover the message with up to t coded values being erased, and the accumulation value uniquely corresponds to the set of coded values and equivalently the original message (Lemma 4). Then every party runs an instance of k-bit Byzantine agreement with the accumulation value as the input. After the above agreement terminates, each party checks whether the agreement output matches its accumulation value, and inputs the result to an 1-bit Byzantine agreement instance. If the above agreement outputs 0, all parties output $\bot$ and abort. If the above agreement outputs 1, then at least one honest party has the accumulation value z_i matching with the agreement output z, and every honest party will agree on the message corresponding to z. Then in Distribute, all parties send the j-th coded value to party P_j. After that, each honest party P_j will send a valid j-th coded

Input of every party P_i: An l-bit message m_i
Primitives: Byzantine agreement oracle, cryptographic accumulator with `Eval`, `CreateWit`, `Verify`
Protocol for party P_i:

1. Compute $\mathcal{D}_i = (\langle 1, s_1\rangle, ..., \langle n, s_n\rangle) = \texttt{Encode}(m_i, b)$. Compute the accumulation value $z_i = \texttt{Eval}(ak, \mathcal{D}_i)$. Input z_i to an instance of k-bit BA oracle.
2. When the above BA outputs z, if $z = z_i$, set $happy_i = 1$, otherwise set $happy_i = 0$. Input $happy_i$ to an instance of 1-bit Byzantine agreement oracle.
3. - If the above BA outputs 0, output $o_i = \perp$ and abort.
   - If the above BA outputs 1 and $happy_i = 1$, invoke $\texttt{Distribute}(\mathcal{D}_i, ak, z)$.
4. For the set of pairs $\{(s_i, w_i)\}$ received from the previous step, if there exists a pair (s_i, w_i) such that $\texttt{Verify}(ak, z, w_i, \langle i, s_i\rangle) = true$, then send (s_i, w_i) to all other parties.
5. If $happy_i = 1$, set $o_i = m_i$. Otherwise, let (s_j, w_j) be the message received from party P_j from the previous step and $\mathcal{S}_i = ((\langle 1, s_1\rangle, w_1), ..., (\langle n, s_n\rangle, w_n))$, and set $o_i = \texttt{Reconstruct}(\mathcal{S}_i, ak, z, t)$.
6. Output o_i.

Figure 2 Protocol Synchronous Crypto. $\frac{n}{2}$-BA.

value to all other parties, from which the correct message can be obtained in `Reconstruct`. One nice property of our protocol is that, if at least one honest party with message m invokes `Distribute`, then all honest parties can obtain m from `Reconstruct` (see the proof of Lemma 6). We prove the validity and agreement properties and analyze the communication complexity below.

▶ **Lemma 5.** *If every honest party has the same input message $m_i = m$, all honest parties output the same message m.*

Proof. If all honest parties have the same input message $m_i = m$, they compute and input the same accumulation value z to the instance of Byzantine agreement in step 1. Then in step 2, the BA outputs z by the validity condition, and any honest party sets $happy_i = 1$. Therefore, every honest party P_i inputs 1 to the 1-bit Byzantine agreement oracle in step 2. By the validity of the Byzantine agreement oracle, the agreement will output 1. Then any honest party P_i sets $o_i = m$ in step 5 since $happy_i = 1$. Hence, all honest parties output m when the protocol terminates. ◀

▶ **Lemma 6.** *All honest parties output the same message.*

Proof. If the Byzantine agreement in step 3 outputs 0, then all honest parties output the same message $\perp$. If the agreement agreement in step 3 outputs 1, then by the validity of the Byzantine agreement, some honest party P_i must input 1 and thus has $z_i = z$. By Lemma 4, any honest party P_i with $happy_i = 1$ has the identical message m corresponding to z, and sets the output to be m at step 5. In step 3, any honest party P_i with $happy_i = 1$ invokes `Distribute` to compute witness w_j for each index value $\langle j, s_j\rangle$, and sends the valid (s_j, w_j) pair computed from message m to party P_j for every P_j. By Lemma 3, the Byzantine parties cannot generate a different pair (s'_j, w'_j) that can be verified. Therefore, in step 4, every honest party P_j receives at least one valid (s_j, w_j) pair, and forwards it to all other parties. Since there are at least $b = n - t$ honest parties, in step 5, each honest party will receive at

least b valid coded values. In `Reconstruct`, using the accumulation value associated with the coded value, any party P_i can detect the corrupted values and remove them. By the property of RS codes, any honest party P_i with $happy_i = 0$ is able to recover the message m, and any honest party P_i with $happy_i = 1$ already has the message m. Therefore all honest parties outputs m. ◀

▶ **Theorem 7.** *Protocol Synchronous Crypto. $\frac{n}{2}$-BA satisfies Termination, Agreement and Validity, and has communication complexity $O(nl + \mathcal{A}(k) + kn^2)$.*

Proof. Termination is clearly satisfied. By Lemma 6, agreement is satisfied. By Lemma 5, validity is satisfied.

Step 1 has cost $\mathcal{A}(k)$, where k is the size of the cryptographic accumulator. Step 2 has cost $\mathcal{A}(1) \leq \mathcal{A}(k)$. Step 3 has cost $O(nl + kn^2)$, since each honest party invokes an instance of `Distribute`, which leads to an all-to-all communication with each message of size $O(l/b + k) = O(l/n + k)$. For step 4, it also has cost $O(nl + kn^2)$ similarly as step 3. Hence the total cost is $O(nl + \mathcal{A}(k) + kn^2)$. ◀

4.3 Lower Bound on BA for Long Messages

Let $\mathcal{A}(l)$ denote the communication complexity in bits of the best possible deterministic protocol for Byzantine agreement with l-bit inputs, n parties, and up to $t = \Theta(n)$ faulty parties. We show a straightforward lower bound that $\mathcal{A}(l) = \Omega(nl + \mathcal{A}(k) + n^2)$ for $l \geq k$ by combining several known lower bounds in the literature.

▶ **Theorem 8.** *$\mathcal{A}(l) = \Omega(nl + \mathcal{A}(k) + n^2)$ for $l \geq k$.*

Proof. The proof combines several simple lower bounds known in the literature.

First of all, $\mathcal{A}(l) = \Omega(nl)$ according to [14]. We briefly mention the proof idea from [14] for completeness. Let a set A of $n - t$ parties have input m and a set B of the rest t parties have input $m' \neq m$. In scenario 1, let parties in B be Byzantine but behave as if they are honest. Then by the validity condition, all parties in A will output m. In scenario 2, let parties in B be honest. To parties in A, the scenario 2 is indistinguishable from scenario 1, and thus they will output m. By the agreement condition, parties in B also need to output m. Therefore each party in B needs to learn the message m, which leads to a lower bound on the communication cost of $\Omega(tl) = \Omega(nl)$.

Secondly, since $\mathcal{A}(l)$ denotes the communication complexity of a BA oracle with l-bit inputs, it is clear that $\mathcal{A}(l) \geq \mathcal{A}(k)$ for $l \geq k$.

Finally, according to [10], $\Omega(n^2)$ is a lower bound on the communication complexity for any deterministic Byzantine agreement protocol tolerating $t = \Theta(n)$ faults (even for single-bit inputs). Thus, $\mathcal{A}(l) \geq \mathcal{A}(1) = \Omega(n^2)$.

The above lower bounds together imply a lower bound $\mathcal{A}(l) = \Omega(nl + \mathcal{A}(k) + n^2)$ for deterministic protocol that solves l-bit BA. ◀

By Theorem 7, our Protocol Synchronous Crypto. $\frac{n}{2}$-BA has cost $O(nl + \mathcal{A}(k) + kn^2)$, which is very close to the lower bound. Although it does not meet the lower bound, we remark that further improvements seem challenging. Notice that if $\mathcal{A}(k) = \Omega(kn^2)$, then a lower bound of $\Omega(nl + \mathcal{A}(k) + kn^2)$ follows, matching our upper bound. Thus, improving upon our upper bound requires a k-bit BA oracle whose communication complexity is $o(kn^2)$.

However, if we were to design an $o(kn^2)$ BA protocol, we have to follow a very particular paradigm. The $\Omega(n^2)$ lower bound from [10] is a lower bound on the number of messages. If every message is signed, then $\Omega(kn^2)$ communication must be incurred. Yet, we know

Input of the sender P_s: An l-bit message m_s
Primitives: Byzantine broadcast oracle, cryptographic accumulator with `Eval`, `CreateWit`, `Verify`
Protocol for party P_i:

1. The sender P_s initializes $o_s = m_s, happy_s = 1$, and other parties P_i initialize $o_i = \perp, happy_i = 0$. The sender computes $\mathcal{D}_s = \texttt{Encode}(m_s, b)$, the accumulator value $z_s = \texttt{Eval}(ak, \mathcal{D}_s)$, and broadcasts z_s by invoking an instance of k-bit Byzantine broadcast oracle. Let z_i denote the output of the Byzantine broadcast at party P_i.
2. For iterations $r = 1, ..., t + 1$:
   Distribution step:
   If $happy_i = 1$, then sign the `HAPPY` message using the multi-signature scheme, send the multi-signature signed by r distinct parties to all other parties, invoke $\texttt{Distribute}(\mathcal{D}_i, ak, z_i)$, and skip the Distribution step in all future iterations.
   Sharing step:
   If a valid (s_i, w_i) pair is received from the Distribution step such that $\texttt{Verify}(ak, z_i, w_i, \langle i, s_i \rangle) = \text{true}$, then send (s_i, w_i) to all other parties and skip the Sharing step in all future iterations.
   Reconstruction step (no communication involved):
   Let (s_j, w_j) be the first message received from party P_j from the Sharing step (possibly from previous iterations). Let $\mathcal{S}_i = ((\langle 1, s_1 \rangle, w_1), ..., (\langle n, s_n \rangle, w_n))$. Compute $M_i = \texttt{Reconstruct}(\mathcal{S}_i, ak, z_i, t)$ and $\mathcal{D}_i = \texttt{Encode}(M_i, b)$. If $\texttt{Eval}(ak, \mathcal{D}_i) = z_i$ and a `HAPPY` message signed by r distinct parties excluding P_i was received in the Distribution step of this iteration, then set $happy_i = 1$, set $o_i = M_i$, and skip the Reconstruction step in all future iterations.
3. Output o_i.

Figure 3 Protocol Synchronous Crypto. $(1 - \varepsilon)$-BB.

authentication is necessary for tolerating minority faults. Thus, such a protocol must use $\Omega(n^2)$ messages in total but only sign a small subset of them. We are not aware of any work exploring this direction, and closing this gap is an interesting open problem.

5 Cryptographically Secure Extension Protocol under $t < (1 - \varepsilon)n$

In this section, we propose an extension protocol for synchronous and authenticated BB with communication complexity $O(nl + \mathcal{B}(k) + kn^2 + n^3)$ under $t < (1 - \varepsilon)n$ where $\varepsilon > 0$ is some constant. The protocol still solves Byzantine broadcast under any $t < n$ faults by setting $b = n - t$, but the communication complexity increases by a factor of $1/\varepsilon$ if ε is not a constant (see Theorem 12). Thus, compared to state-of-art solutions [15, 16], our protocol is more efficient when ε is a constant but less efficient otherwise.

Protocol Synchronous Crypto. $(1 - \varepsilon)n$-BB. The protocol is presented in Figure 3, and we briefly explain each step of the protocol. Again let t denote the maximum number of Byzantine parties and let $b = n - t$. First the sender encodes its message and computes the accumulation value using the coded values. Then the sender broadcasts the accumulation value via an instance of k-bit Byzantine broadcast oracle. By the agreement condition, all honest replicas output the same value for BB. The remaining of the protocol runs in iterations

$r = 1, 2, ..., t + 1$. Each iteration consists 3 steps. The Distribution step, Sharing step and Reconstruction step are analogous to steps $3 - 5$ in Protocol Synchronous Crypto. $\frac{n}{2}$-BA in Figure 2, but here each step is examined in every iteration for execution, and is executed only once. The Distribution step aims to distribute the indexed coded values to other parties. The Sharing step forwards the correct coded value to other parties. The Reconstruction step aims to reconstruct the original message from the coded values received from other parties and set the output. Similar to Protocol Synchronous Crypto. $\frac{n}{2}$-BA, the above steps provide a nice guarantee that if at least one honest party with message m invokes `Distribute` in the Distribution step, then all honest parties can obtain m in the Reconstruction step (see the proof of Lemma 9).

Now we give a more detailed description. A party becomes happy (i.e., sets $happy_i = 1$) if it is ready to output a message that is not $\perp$. In the first iteration, only the sender is happy; it invokes `Distribute` and also signs and sends a message `HAPPY` of a constant size. The role of the message `HAPPY` is to be signed by the rest of the parties using multi-signatures to form a signature chain, similar to the Dolve-Strong Byzantine broadcast algorithm [11]. An honest party becomes happy at the end of iteration r, if it reconstructs the correct message (matching the agreed upon accumulation value) in the Reconstruction step of iteration r and has received a `HAPPY` message signed by r parties in the Distribution step of iteration r. When an honest party becomes happy, it will set its output to be the reconstructed message M_i; then, in the Distribution step of the next iteration (if there is one), it will also send its own signature of `HAPPY` to all other parties, and invoke `Distribute`. This way, if an honest party becomes happy in the last iteration $r = t + 1$, it can be assured that some honest party has invoked `Distribute`, so that all honest parties will be ready to output the correct message. We reiterate that each step is executed at most once in the entire protocol. Finally, after $t + 1$ iterations, every party outputs the message.

▶ **Lemma 9.** *If any honest party P_i invokes `Distribute` with message m, then every honest party P_j outputs $o_j = m$.*

Proof. By the agreement condition of the Byzantine broadcast, the output z_i of the BB at every honest party P_i is identical. If an honest party P_i invokes `Distribute` with message m, m satisfies $z_i = \texttt{Eval}(ak, \texttt{Encode}(m, b))$. If any other honest party P_j sets $o_j = m'$ after initialization, it must satisfy $\texttt{Eval}(ak, \texttt{Encode}(m', b)) = z_j = z_i$. By Lemma 4, $m = m'$. Thus, we only need to show that every other honest party P_j sets o_j.

Suppose that P_i invokes `Distribute` in some iteration r. According to the subprotocol `Distribute`, P_i computes a witness w_j for each indexed value $\langle j, s_j \rangle$ and sends the pair (s_j, w_j) to each party P_j. According to Lemma 3, the adversary cannot generate $d' \notin \mathcal{D}_i$ and a witness w' such that $\texttt{Verify}(ak, z_i, w', d') = true$. Then, in Sharing step of iteration r, every honest party P_j can identify and forward the valid pair (s_j, w_j) to all other parties, unless it has already done that in previous iterations. Since there are at least $n - t = b$ honest parties, in the Reconstruction step of iteration r, every honest party P_j receives at least $n - t = b$ correct coded values. In `Reconstruct`, using the witness associated with the indexed coded value, every party P_j can identify the corrupted values and remove them. The number of erased values is at most t. By the property of RS codes, P_j with $happy_j = 0$ is able to recover the message m.

Furthermore, we will show that each party receives a `HAPPY` message signed by r distinct parties in the Reconstruction step of iteration r. If $r = 1$, then $P_i = P_s$ and every P_j will receive a signature for `HAPPY`. If $r > 1$, then P_i has received a multi-signature of `HAPPY` signed by $r - 1$ distinct parties excluding P_i in the Reconstruction step of iteration $r - 1$; P_i adds its own signature of `HAPPY` in iteration r, so each honest P_j will receive a multi-signature of `HAPPY` signed by r distinct parties in the Reconstruction step of iteration r.

Therefore, if $happy_j = 0$ up till now, then an honest P_j will set $happy_j = 1$ and $o_j = m$ in the Reconstruction step of iteration r. If $happy_j = 1$, then P_j has already set $o_j = m$. Note that an honest sender does not set its output again in the Reconstruction step, since the `HAPPY` message always contains its signature. Once P_j sets o_j, it will skip the Reconstruction step in all future iterations, and o_j will not be changed. Therefore, all honest parties output m when they terminate. ◀

▶ **Lemma 10.** *If the sender is honest and has input m_s, every honest party outputs m_s.*

Proof. In iteration $r = 1$, the sender sends a signed `HAPPY` to all other parties and invokes `Distribute`. By Lemma 9, every honest parties output m_s. ◀

▶ **Lemma 11.** *Every honest party outputs the same message.*

Proof. If all honest parties output $\perp$, then the lemma is true. Otherwise, suppose some honest party P_i outputs $o_i = m$ where $m \neq \perp$. If P_i is the sender, then by Lemma 10, all honest parties output m. Now consider the case where P_i is not the sender. According to the protocol, if $P_i \neq P_s$ sets $o_i = m \neq \perp$ in the Reconstruction step of iteration $1 \leq r \leq t$, P_i will invoke `Distribute` with m in iteration $r + 1$. By Lemma 9, all honest parties output m. If the honest party P_i sets $o_i = m$ in iteration $r = t + 1$, according to the protocol, P_i receives a `HAPPY` signed by $t + 1$ distinct parties. Since there are at most t Byzantine parties, there exists at least one honest party $P_j \neq P_i$ that has signed `HAPPY` and invoked `Distribute` with $o_j = m'$ in a previous iteration $1 \leq r' \leq t$. Then, by Lemma 9, all honest parties including P_i output m'. Therefore, $m' = m$, and all honest parties output m. ◀

▶ **Theorem 12.** *Protocol Synchronous Crypto. $(1-\varepsilon)n$-BB satisfies Termination, Agreement and Validity. The protocol has communication complexity $O(nl/\varepsilon + \mathcal{B}(k) + kn^2 + n^3)$.*

Proof. Termination is clearly satisfied. By Lemma 11, agreement is satisfied. By Lemma 10, validity is satisfied.

Step 1 has cost $\mathcal{B}(k)$ for the k-bit BB oracle. The Distribution step has total communication cost $O(nl/\varepsilon + kn^2 + n^3)$, since each honest party executes the Distribution step at most once, where invoking `Distribute` has cost $O(n\cdot(l/b+k)) = O(\frac{n}{n-t}l+kn) = O(l/\varepsilon+kn)$, and sending the signed `HAPPY` message has cost $O((k + n)n)$ where the $(k + n)$ term is due to the signature size and the list of signers in the multi-signature scheme. The Sharing step is also performed at most once for every honest party, and has total cost $O(nl/\varepsilon + kn^2)$ since each honest party in the Sharing step sends a message of size $O(l/(n\varepsilon)+k)$ to all other parties. The Reconstruction step has no communication cost. Hence, the total communication complexity is $O(nl/\varepsilon + \mathcal{B}(k) + kn^2 + n^3)$. ◀

Optimality with the current best BB oracle. From Section 2, the classic Dolev-Strong [11] protocol remains the best deterministic solution for $t > n/2$ BB, with cost $\mathcal{B}(k) = \Theta((k + k_s)n^2 + n^3)$ for k-bit inputs where k_s is the signature size. Our protocol invokes Dolev-Strong with $k = k_a$ (the size of the accumulation value). Since $\Theta(k_s) = \Theta(k_a)$, our protocol achieves $\mathcal{B}(l) = O(nl + kn^2 + n^3)$.

As before, $\Omega(nl)$ is a trivial lower bound for l-bit BB [14] (intuitively, all parties need to receive the sender's message); in addition $\mathcal{B}(l) \geq \mathcal{B}(k)$ if $l \geq k$. Thus, the $\mathcal{B}(l) = O(nl+kn^2+n^3)$ communication complexity cannot be further improved unless a deterministic BB protocol better than Dolev-Strong is found.

Input of every party P_i: An l-bit message m_i
Primitives: asynchronous Byzantine agreement oracle, cryptographic accumulator with `Eval`, `CreateWit`, `Verify`
Protocol for party P_i:

1. Compute $\mathcal{D}_i = (\langle 1, s_1\rangle, ..., \langle n, s_n\rangle) = \texttt{Encode}(m_i, b)$. Compute the accumulation value $z_i = \texttt{Eval}(ak, \mathcal{D}_i)$. Input z_i to an instance of k-bit asynchronous Byzantine agreement oracle.
2. When the above ABA outputs z, if $z = z_i$, set $happy_i = 1$, otherwise set $happy_i = 0$. Input $happy_i$ to an instance of 1-bit asynchronous Byzantine agreement oracle.
3. - If the above ABA outputs 0, output $o_i = \bot$ and abort.
   - If the above ABA outputs 1 and $happy_i = 1$, invoke $\texttt{Distribute}(\mathcal{D}_i, ak, z)$.
4. Wait for a valid (s_i, w_i) pair such that $\texttt{Verify}(ak, z, w_i, \langle i, s_i\rangle) = \text{true}$, then send (s_i, w_i) to all other parties.
5. If $happy_i = 1$, set $o_i = m_i$. Otherwise, perform the following. Wait for at least $n-t$ valid pairs $\{(s_j, w_j)\}$ from the previous step that satisfies $\texttt{Verify}(ak, z, w_j, \langle j, s_j\rangle) = \text{true}$. Let $\mathcal{S}_i = ((\langle 1, s_1\rangle, w_1), ..., (\langle n, s_n\rangle, w_n))$, where (s_j, w_j) is the pair received from party P_j. Compute $o_i = \texttt{Reconstruct}(\mathcal{S}_i, ak, z, t)$.
6. Output o_i.

Figure 4 Protocol Asynchronous Crypto. $\frac{n}{3}$-BA.

6 Cryptographically Secure Extension Protocols Under Asynchrony

As mentioned, our cryptographically secure extension protocols can be extended to the asynchronous setting to solve BA and reliable broadcast (RB) under $< n/3$ faults. No extension protocol has been proposed for this case to the best of our knowledge. As before, let t denote the maximum number of Byzantine parties, and let $b = n - t$.

6.1 Asynchronous Byzantine Agreement

The protocol is presented in Figure 4, which consists steps analogous to the synchronous protocol. The main difference is that in the asynchronous extension protocol, Steps 4 and 5 are executed once enough messages are received. As a result, the proofs are also similar to the synchronous version and we omit them.

▶ **Theorem 13.** *Protocol Asynchronous Crypto. $\frac{n}{3}$-BA satisfies Termination, Agreement and Validity, and has communication complexity $O(nl + \mathcal{A}(k) + kn^2)$.*

6.2 Asynchronous Reliable Broadcast

Reliable broadcast relaxes the termination property of the broadcast definition (Definition 1): only when the sender is honest, all honest parties are required to output; otherwise, it is allowed that *either* all honest parties output *or* no honest party outputs. The agreement property is slightly modified accordingly.

▶ **Definition 14** (Reliable Broadcast). *A protocol for a set of parties $\mathcal{P} = \{P_1, ..., P_n\}$, where a distinguished party called the sender $P_s \in \mathcal{P}$ holds an initial l-bit input m, is a reliable broadcast protocol tolerating an adversary A, if the following properties hold*

- *Termination. If the sender is honest, then every honest party eventually outputs a message. Otherwise, if some honest party outputs a message, then every honest party eventually outputs a message.*

Input of the sender P_s: An l-bit message m_s
Primitive: asynchronous Byzantine agreement oracle, asynchronous reliable broadcast oracle, cryptographic accumulator with `Eval`, `CreateWit`, `Verify`
Protocol for party P_i:

1. If $i = s$, perform the following. Compute $\mathcal{D}_s = (\langle 1, s_1\rangle, ..., \langle n, s_n\rangle) = \texttt{Encode}(m_s, b)$. Compute the accumulation value $z_s = \texttt{Eval}(ak, \mathcal{D}_s)$. Send m_s to every party, and broadcast z_s by invoking a k-bit asynchronous reliable broadcast oracle.
2. When receiving the message m from the sender, and the reliable broadcast above outputs z, perform the following. Compute $\mathcal{D}_i = (\langle 1, s_1\rangle, ..., \langle n, s_n\rangle) = \texttt{Encode}(m, b)$. Compute the accumulation value $z_i = \texttt{Eval}(ak, \mathcal{D}_i)$. If $z_i = z$, set $happy_i = 1$, otherwise set $happy_i = 0$.
3. If $happy_i = 1$, invoke $\texttt{Distribute}(\mathcal{D}_i, ak, z)$.
4. Step 4 to 6 are identical to that of Protocol Asynchronous Crypto. $\frac{n}{3}$-BA in the Figure 4, except that the replica computes $\mathcal{D}'_i = \texttt{Encode}(o_i, b)$, and invokes $\texttt{Distribute}(\mathcal{D}'_i, ak, z)$ at the end of Step 5.

Figure 5 Protocol Asynchronous Crypto. $\frac{n}{3}$-RB.

- *Agreement. If some honest party outputs a message m', then every honest party eventually outputs m'.*
- *Validity. If the sender is honest, all honest parties eventually output the message m.*

The extension protocol for asynchronous reliable broadcast is presented in Figure 5.

▶ **Lemma 15.** *If an honest party P_i invokes `Distribute` with $\mathcal{D}_i = \texttt{Encode}(m, b)$, then any honest party P_j eventually output $o_j = m$.*

Proof. By the agreement condition of asynchronous reliable broadcast oracle used in step 1, if any honest party obtains z, then any honest party also eventually obtains z. Then at step 2, by Lemma 4, any honest party P_j with $happy_j = 1$ has the identical message m corresponding to z, and sets $o_j = m$ at step 5. For other honest parties, the honest party P_i with $happy_i = 1$ invokes `Distribute` to compute witness w_j for each indexed value $\langle j, s_j\rangle$, and sends the valid (s_j, w_j) pair computed from message m to party P_j for every P_j. By Lemma 3, the Byzantine parties cannot generate a different pair (s'_j, w'_j) that can be verified. Therefore, in step 4, every honest party P_j eventually receives at least one valid (s_j, w_j) pair, and forwards it to all other parties. Since there are at least $n - t$ honest parties, in step 5, each honest party will eventually receive at least $n - t$ valid coded values. In `Reconstruct`, using the accumulation value associated with the coded value, any party P_j can detect the corrupted values and remove them. By the property of RS codes, any honest party P_j with $happy_j = 0$ is able to recover the message m, and any honest party P_j with $happy_j = 1$ already has the message m. Therefore all honest parties output m. ◀

▶ **Lemma 16.** *If the sender is honest and has input m_s, all honest parties eventually output the same message m_s.*

▶ **Lemma 17.** *If some honest party outputs a message m, then every honest party eventually outputs m.*

▶ **Theorem 18.** *Protocol Asynchronous Crypto. $\frac{n}{3}$-RB satisfies Termination, Agreement and Validity. The protocol has communication complexity $O(nl + \mathcal{B}(k) + kn^2)$.*

The proofs of Lemma 16, 17 and Theorem 18 are in the full version of this paper.

7 Conclusion

We investigate and propose several extension protocols with improved communication complexity for solving Byzantine broadcast and agreement under various settings. We propose simple yet efficient authenticated extension protocols with improved communication complexity, for Byzantine agreement under $t < n/2$, and for Byzantine broadcast under $t < (1-\varepsilon)n$ where $\varepsilon > 0$ is a constant. The above results can be extended to the asynchronous case to obtain authenticated extension protocols for Byzantine agreement and reliable broadcast.

References

1. Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. Asymptotically optimal validated asynchronous byzantine agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 337–346, 2019.
2. Niko Barić and Birgit Pfitzmann. Collision-free accumulators and fail-stop signature schemes without trees. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 480–494. Springer, 1997.
3. Piotr Berman, Juan A Garay, and Kenneth J Perry. Bit optimal distributed consensus. In *Computer science*, pages 313–321. Springer, 1992.
4. Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. Aggregate and verifiably encrypted signatures from bilinear maps. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 416–432. Springer, 2003.
5. Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.
6. Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *Annual International Cryptology Conference*, pages 524–541. Springer, 2001.
7. Christian Cachin and Stefano Tessaro. Asynchronous verifiable information dispersal. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*, pages 191–201. IEEE, 2005.
8. Wutichai Chongchitmate and Rafail Ostrovsky. Information-theoretic broadcast with dishonest majority for long messages. In *Theory of Cryptography Conference*, pages 370–388. Springer, 2018.
9. David Derler, Christian Hanser, and Daniel Slamanig. Revisiting cryptographic accumulators, additional properties and relations to other primitives. In *Cryptographers' Track at the RSA Conference*, pages 127–144. Springer, 2015.
10. Danny Dolev and Ruediger Reischuk. Bounds on information exchange for byzantine agreement. In *Proceedings of the first ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 132–140. ACM, 1982.
11. Danny Dolev and H. Raymond Strong. Authenticated algorithms for byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983.
12. Sisi Duan, Michael K Reiter, and Haibin Zhang. Beat: Asynchronous bft made practical. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2028–2041, 2018.
13. Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.
14. Matthias Fitzi and Martin Hirt. Optimally efficient multi-valued byzantine agreement. In *Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, pages 163–168. ACM, 2006.
15. Chaya Ganesh and Arpita Patra. Broadcast extensions with optimal communication and round complexity. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, pages 371–380. ACM, 2016.

16 Chaya Ganesh and Arpita Patra. Optimal extension protocols for byzantine broadcast and agreement. *Distributed Computing*, pages 1–19, 2020.

17 Aniket Kate, Gregory M Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 177–194. Springer, 2010.

18 Taechan Kim and Razvan Barbulescu. Extended tower number field sieve: A new complexity for the medium prime case. In *Annual International Cryptology Conference*, pages 543–571. Springer, 2016.

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

20 Guanfeng Liang and Nitin Vaidya. Error-free multi-valued consensus with byzantine failures. In *Proceedings of the 30th annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 11–20. ACM, 2011.

21 Yuan Lu, Zhenliang Lu, Qiang Tang, and Guiling Wang. Dumbo-mvba: Optimal multi-valued validated asynchronous byzantine agreement, revisited. In *Proceedings of the 39th Symposium on Principles of Distributed Computing*, page 129–138, 2020.

22 Nancy A Lynch. *Distributed algorithms*. Elsevier, 1996.

23 Ralph C Merkle. A digital signature based on a conventional encryption function. In *Conference on the theory and application of cryptographic techniques*, pages 369–378. Springer, 1987.

24 Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 31–42. ACM, 2016.

25 Achour Mostéfaoui, Hamouma Moumen, and Michel Raynal. Signature-free asynchronous binary byzantine consensus with $t < n/3$, $O(n^2)$ messages, and $O(1)$ expected time. *Journal of the ACM (JACM)*, 62(4):1–21, 2015.

26 Shuai Mu, Kang Chen, Yongwei Wu, and Weimin Zheng. When paxos meets erasure code: Reduce network and storage cost in state machine replication. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pages 61–72, 2014.

27 Lan Nguyen. Accumulators from bilinear pairings and applications. In *Cryptographers' Track at the RSA Conference*, pages 275–292. Springer, 2005.

28 Arpita Patra. Error-free multi-valued broadcast and byzantine agreement with optimal communication complexity. In *International Conference On Principles Of Distributed Systems*, pages 34–49. Springer, 2011.

29 Birgit Pfitzmann and Michael Waidner. Information-theoretic pseudosignatures and byzantine agreement for $t \geq n/3$. *Research report, IBM Research*, 1996.

30 Irving S Reed and Gustave Solomon. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics*, 8(2):300–304, 1960.

31 Zizhong Wang, Tongliang Li, Haixia Wang, Airan Shao, Yunren Bai, Shangming Cai, Zihan Xu, and Dongsheng Wang. Craft: An erasure-coding-supported version of raft for reducing storage cost and network cost. In *18th USENIX Conference on File and Storage Technologies*, pages 297–308, 2020.

32 Andrew C Yao. Protocols for secure computations. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, pages 160–164. IEEE Computer Society, 1982.

33 Andrew Chi-Chih Yao. Some complexity questions related to distributive computing. In *Proceedings of the eleventh annual ACM symposium on Theory of computing*, pages 209–213. ACM, 1979.

From Partial to Global Asynchronous Reliable Broadcast

Diana Ghinea

Department of Computer Science, ETH Zurich, Switzerland
ghinead@ethz.ch

Martin Hirt

Department of Computer Science, ETH Zurich, Switzerland
hirt@inf.ethz.ch

Chen-Da Liu-Zhang

Department of Computer Science, ETH Zurich, Switzerland
lichen@inf.ethz.ch

Abstract

Broadcast is a fundamental primitive in distributed computing. It allows a sender to consistently distribute a message among n recipients. The seminal result of Pease et al. [JACM'80] shows that in a complete network of synchronous bilateral channels, broadcast is achievable if and only if the number of corruptions is bounded by $t < n/3$. To overcome this bound, a fascinating line of works, Fitzi and Maurer [STOC'00], Considine et al. [JC'05], and Raykov [ICALP'15], proposed strengthening the communication network by assuming *partial synchronous broadcast* channels, which guarantee consistency among a subset of recipients.

We extend this line of research to the asynchronous setting. We consider *reliable* broadcast protocols assuming a communication network which provides each subset of b parties with reliable broadcast channels. A natural question is to investigate the trade-off between the size b and the corruption threshold t. We answer this question by showing feasibility and impossibility results:

- A reliable broadcast protocol $\Pi_{\texttt{RBC}}$ that:
  - For $3 \leq b \leq 4$, is secure up to $t < n/2$ corruptions.
  - For $b > 4$ even, is secure up to $t < \left(\frac{b-4}{b-2}n + \frac{8}{b-2}\right)$ corruptions.
  - For $b > 4$ odd, is secure up to $t < \left(\frac{b-3}{b-1}n + \frac{6}{b-1}\right)$ corruptions.
- A *nonstop* reliable broadcast $\Pi_{\texttt{nRBC}}$, where parties are guaranteed to obtain output as in reliable broadcast but may need to run forever, secure up to $t < \frac{b-1}{b+1}n$ corruptions.
- There is no protocol for (nonstop) reliable broadcast secure up to $t \geq \frac{b-1}{b+1}n$ corruptions, implying that $\Pi_{\texttt{RBC}}$ is an asymptotically optimal reliable broadcast protocol, and $\Pi_{\texttt{nRBC}}$ is an optimal nonstop reliable broadcast protocol.

2012 ACM Subject Classification Theory of computation → Cryptographic protocols; Theory of computation → Distributed algorithms; Security and privacy → Cryptography

Keywords and phrases asynchronous broadcast, partial broadcast

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.29

Related Version A full version of the paper is available at `https://eprint.iacr.org/2020/963`.

1 Introduction

Broadcast protocols constitute a fundamental building block in distributed computing. They allow a sender to consistently distribute a message among n recipients, even if some of them exhibit arbitrary behaviour. It is used as an important primitive in many applications, such as verifiable secret-sharing or secure-multiparty computation [9, 2, 5, 13].

The seminal result of Pease et al. [12] shows that in the standard communication model of a complete synchronous network of pairwise authenticated channels, perfectly-secure Byzantine broadcast is achievable if and only if less than a third of the parties are corrupted

© Diana Ghinea, Martin Hirt, and Chen-Da Liu-Zhang;
licensed under Creative Commons License CC-BY

(i.e., $t < n/3$). To overcome this bound, a line of works [8, 6, 15] has considered using stronger communication primitives such as *partial broadcast channels*, which guarantee that a message is consistent among all recipients on the channel. Hence, a natural question is to investigate a generalization of the classical broadcast problem, namely the trade-off between the strength of the communication primitives and the corruptive power from the adversary.

To the best of our knowledge, all works investigating such trade-offs for broadcast achievability [8, 6, 15] operate in the so-called *synchronous* model, where parties have access to synchronized clocks, and there is a known upper bound on the network delay.

A more realistic setting is the so-called *asynchronous* model, where no timing assumption is made. In the asynchronous model, the classical notion of synchronous Byzantine broadcast, where termination is guaranteed, is not achievable, since one cannot distinguish between a dishonest sender not sending a message or an honest sender being slow [4, 3, 1]. Hence, one considers the weaker notion of *reliable* broadcast, where parties may not obtain output if the sender is dishonest; however, if an honest party obtains output, every honest party does as well. To the best of our knowledge, constructions of reliable broadcast [3] in the asynchronous setting are only known up to $t < n/3$ corruptions.

A natural question is then to investigate such trade-offs between the communication network and the corruptive power of the adversary in the asynchronous model.[1]

In the asynchronous network $\mathcal{N}_b$ where parties can reliably broadcast to any subset of b parties, for which t is there a reliable broadcast protocol secure up to t corruptions?

We answer this question by showing feasibility and impossibility results:

Feasibility Results. In the network communication $\mathcal{N}_b$, we show:

- A reliable broadcast protocol $\Pi_{\texttt{RBC}}$ that satisfies:
  - For $b \leq 4$, secure up to $t < n/2$ corruptions.
  - For $b > 4$ even, secure up to $t < \left(\frac{b-4}{b-2}n + \frac{8}{b-2}\right)$ corruptions.
  - For $b > 4$ odd, secure up to $t < \left(\frac{b-3}{b-1}n + \frac{6}{b-1}\right)$ corruptions.
- A *nonstop* reliable broadcast $\Pi_{\texttt{nRBC}}$, where parties are guaranteed to obtain output as in reliable broadcast but may need to run forever, secure up to $t < \frac{b-1}{b+1}n$ corruptions.

Impossibility Result. Following the impossibility of [6], we show in the full version of the paper that, in the network $\mathcal{N}_b$, there is no protocol for (nonstop) reliable broadcast secure up to $t \geq \frac{b-1}{b+1}n$ corruptions, implying that $\Pi_{\texttt{RBC}}$ is an asymptotically optimal reliable broadcast protocol when $n \to \infty$ and $b = O(n)$, and that $\Pi_{\texttt{nRBC}}$ is an optimal nonstop reliable broadcast protocol.

1.1 Related Work

Previous results operate in the synchronous model, where parties proceed in *rounds*, and messages sent at round r are guaranteed to be delivered by round $(r + 1)$.

Fitzi and Maurer [8] showed that assuming partial Byzantine broadcast channels among every triplet of parties, global Byzantine broadcast can be realized securely if and only

[1] Investigating such trade-off is additionally motivated as a natural way to overcome the $n/3$-bound of constructing reliable broadcast in the pairwise channels setting. Note that this bound holds even assuming public-key infrastructure (PKI), in contrast to the synchronous counterpart, where Byzantine broadcast (and reliable broadcast) can be achieved under arbitrary many corruptions with PKI [7].

if $t < n/2$. Considine et al. [6] generalized this result to the b-cast model, i.e. a partial Byzantine broadcast channel among any b parties, showing that Byzantine broadcast is achievable if and only if $t < \frac{b-1}{b+1}n$. Raykov [15] generalized this result to the setting of *general adversaries* [10] proving that broadcast is achievable from b-cast channels against adversary structures $\mathcal{A}$ if and only if $\mathcal{A}$ satisfies the so-called $(b+1)$-chain-free condition.

Some additional works focus on the setting of *incomplete* communication networks, where some of the partial b-cast channels might be missing. Ravikant et al. [14] provide necessary and sufficient conditions for 3-cast networks to satisfy so that Byzantine agreement can be achieved while tolerating threshold adversaries in the range $n/3 \leq t < n/2$. In a follow-up work, Jaffe et al. [11] provide asymptotically tight bounds on the number of necessary and sufficient 3-cast channels to construct Byzantine agreement for the same threshold adversary.

1.2 Comparison to Previous Work

We argue that the asynchronous setting has different challenges from the ones in the synchronous setting. Compared to previous works which assume and construct Byzantine broadcast, in this work we assume and construct reliable broadcast. That is, although our constructed primitive is weaker than the traditional Byzantine broadcast primitive (it does not guarantee termination in the dishonest sender case), we also assume a weaker primitive, which poses new challenges. For example, in the synchronous model, the primitive *proxcast* [6] which provides a weak form of consistency where parties output a level of confidence, and is used as a core building block to construct Byzantine broadcast, can be achieved simply by allowing the sender to partially broadcast the input value via all possible b-casts, and letting each recipient R_i take a deterministic decision based on all the outputs: R_i decides on level ℓ_i as the minimum number of parties with whom R_i sees only zeros. However, in the asynchronous model, parties cannot wait for the outcome of all partial reliable broadcasts from the sender, because the sender may be dishonest and so some of the partial channels may not output a value. As a consequence, R_i needs to make a decision *without* knowing the outcome of all partial channels. In general, parties have to make progress in the protocol after seeing the messages from $n-t$ parties, as all other parties could be corrupted. This is especially troublesome in the dishonest majority setting, where parties need to make progress after seeing messages from $n-t \leq t$ parties, i.e., even when potentially no message from any honest party is received. The key idea to overcome this is by observing that honest parties can wait for messages from $n-t$ parties that are *consistent*, i.e., it is allowed to wait for more than $n-t$ parties if inconsistency is received. This prevents the adversary, with the help of partial channels, to send arbitrary inconsistent messages.

2 Model and Definitions

We consider a setting with $n+1$ parties, where a designated party, called the sender S, distributes a value to a set of n recipients $\mathcal{R} = \{R_1, \ldots, R_n\}$. We note that the *insider-sender* setting where the sender is also a recipient is a special case, as it can run in parallel both processors, acting as sender and as recipient simultaneously.

2.1 Communication, Adversary and Setup

In this work, we generalize the communication model where, in addition to a complete network of pair-wise authenticated channels, parties have access to *partial reliable broadcast* channels as well. We refer to such a partial reliable broadcast channel $\mathsf{RBC}(S, \{R_1, \ldots, R_{b-1}\})$ with a

sender S and $b-1$ additional recipients $\{R_1, \ldots, R_{b-1}\}$ as b-cast channel. We denote by $\mathcal{N}_b$ the generalized communication model where each party P, sender or recipient, in addition has access to all channels $\mathsf{RBC}(P, \{R_{i_i}, \ldots, R_{i_{b-1}}\})$, where $R_{i_i}, \ldots, R_{i_{b-1}}$ are $b-1$ additional recipients. The network is fully asynchronous. That is, we assume that the adversary has full control over the network and can schedule the messages in an arbitrary manner. However, each message must be eventually delivered.

We consider the same adversarial model and setup as in [6]. We consider an *adaptive* adversary who can gradually corrupt parties and take full control over them. Moreover, we require our protocols to be *unconditionally secure*, meaning that security holds even against a computationally unbounded adversary. Note, however, that our impossibility proofs hold even with respect to a *static* adversary that is assumed to choose the corrupted parties at the beginning of the protocol execution and in addition is computationally bounded. Finally, we consider the setting where parties have no public-key infrastructure available.

2.2 Reliable Broadcast

Reliable broadcast is a fundamental primitive in distributed computing which allows a designated party S, called the sender, to consistently distribute a message towards a set of recipients $\mathcal{R} = \{R_1, \ldots, R_n\}$.

▶ **Definition 1.** *A protocol π where initially the sender S holds an input m and every recipient R_i terminates upon generating output is a reliable broadcast protocol up to t corruptions, if the following properties are satisfied:*

- ***Validity:*** *If the sender is honest, the sender terminates and every honest recipient terminates with output m.*
- ***Consistency:*** *If an honest recipient terminates with output m, every honest recipient terminates with output m.*

We additionally define a slightly weaker version of broadcast, which requires the recipients to obtain outputs like in reliable broadcast, but may need to run forever.

▶ **Definition 2.** *A protocol π where initially the sender S holds an input m is a nonstop reliable broadcast protocol up to t corruptions, if the following properties are satisfied:*

- ***Validity:*** *If the sender is honest, every honest recipient outputs m.*
- ***Consistency:*** *If an honest recipient outputs m, every honest recipient outputs m.*

3 A Warm-Up Protocol in $\mathcal{N}_3$

In this section, we consider the model $\mathcal{N}_3$, where the parties have access to 3-cast channels. That is, any party can reliably broadcast messages to any subset of 2 recipients.

We present a reliable broadcast protocol $\Pi_{\mathtt{RBC}}^{n,3}$ in the communication network $\mathcal{N}_3$ secure up to $t < n/2$ corruptions inspired by Bracha's reliable broadcast protocol [3]. In the full version of the paper, we show that the construction is optimal with respect to the corruption threshold.

$\Pi_{\mathtt{RBC}}^{n,3}$ first lets the sender S *mega-send* its input message m (distribute m to any two recipients, via all available 3-cast channels). Any recipient R_i that *mega-receives* the same message $(\mathtt{MSG}, m)$ from S (receives consistently $(\mathtt{MSG}, m)$ via all (in total $n-1$) 3-cast channels from S), mega-sends a message $(\mathtt{READY}, m)$ notifying all recipients that it is ready to output m. Any recipient R_i that receives consistent notification messages $(\mathtt{READY}, m)$ from $t+1$ different recipients mega-sends $(\mathtt{READY}, m)$. Finally, any recipient R_i that mega-sent $(\mathtt{READY}, m)$ and

mega-received consistent notification messages $(\texttt{READY}, m)$ from $n-t-1$ other different recipients than himself, outputs m and terminates.

Intuitively, the usage of 3-cast channels guarantees that honest parties send consistent READY messages, because two recipients cannot mega-receive different messages from the sender (they have a common 3-cast channel with the sender). Moreover, note that if an honest recipient R_i mega-receives $(\texttt{READY}, m)$ from R_j, then any honest recipient R_k receives $(\texttt{READY}, m)$ from R_j via the 3-cast containing R_j as the sender and $\{R_i, R_k\}$ as the recipients. This ensures that if an honest recipient outputs a message m, meaning that it sent $(\texttt{READY}, m)$ and mega-received $(\texttt{READY}, m)$ from $n-t-1$ different recipients, then any honest recipient eventually receives $(\texttt{READY}, m)$ from $n-t \geq t+1$ different recipients. It then follows that all honest parties mega-send $(\texttt{READY}, m)$, so all honest parties mega-receive $(\texttt{READY}, m)$ from at least $n-t-1$ parties and terminate. The protocol is described below, and a formal analysis of it can be found in the full version of the paper.

Protocol $\Pi_{\mathsf{RBC}}^{n,3}$

Code for the sender S

1: On input m, send $(\texttt{MSG}, m)$ to every pair of recipients via 3-cast and terminate with output m.

Code for recipient R_i

1: Upon receiving $(\texttt{MSG}, m)$ via all 3-cast $\mathsf{RBC}(S, \{R_i, R_j\})$, $R_j \in \mathcal{R}$, send $(\texttt{READY}, m)$ to every pair of recipients via 3-cast.
2: Upon receiving $(\texttt{READY}, m)$ from $t+1$ different recipients, i.e. from $\mathsf{RBC}(R_j, \{R_i, \cdot\})$, $R_j \in T$, $|T| \geq t+1$, if no READY message was sent, send $(\texttt{READY}, m)$ to every pair of recipients via 3-cast.
3: Upon receiving $(\texttt{READY}, m)$ via all 3-cast $\mathsf{RBC}(R_k, \{R_i, R_j\})$, $R_j \in \mathcal{R}$, $R_k \in T$, $|T| \geq n-t-1$, if $(\texttt{READY}, m)$ was sent, output m and terminate.

4 Notation for Protocols in $\mathcal{N}_b$

We consider the model $\mathcal{N}_b$ where parties, sender or recipients, have access to b-cast channels. That is, any party can reliably broadcast a message to any subset of $b-1$ recipients. We introduce some definitions that will be convenient to describe our protocols. Generalizing the terminology of parties mega-receiving messages in Section 2.2, we add a definition for parties that receive a message via all b-cast channels including a certain subset of recipients.

▶ **Definition 3.** *Let $P \in \{S\} \cup \mathcal{R}$ be a party and $U \subseteq \mathcal{R} \setminus \{P\}$. We denote $\mathcal{B}_P(U) := \{\mathsf{RBC}(P, V) : U \subseteq V \subseteq \mathcal{R} \wedge |V| = b-1\}$ the set of b-cast channels that include P as the sender and any subset of $b-1$ recipients that includes U as receivers.*

In particular, note that if $U' \subseteq U$, then $\mathcal{B}_P(U) \subseteq \mathcal{B}_P(U')$.

▶ **Definition 4.** *We say that a party $P \in \{S\} \cup \mathcal{R}$ U-sends a message m if it sends m through every channel in $\mathcal{B}_P(U)$.*

▶ **Definition 5.** *We say that a recipient $R \in \mathcal{R}$ U-receives a message m from a party P if it receives m through every channel $\mathcal{B}_P(U)$. Moreover, we say that R l-receives m if such $U \subseteq \mathcal{R}$ with $|U| = l$ exists.*

▶ Remark 6. Note that if $|U| \geq b$, $\mathcal{B}_P(U) = \varnothing$, and any recipient b-receives any message.

We add some properties about l-receiving values among parties. Their proofs are included in the full version of the paper.

▶ **Lemma 7.** *Let $l \leq b$. If a recipient R_i l-receives m from P, then all recipients eventually $(l+1)$-receive m.*

▶ **Lemma 8.** *If $R_i \in \mathcal{R}$ U-receives m from P and $R_j \in \mathcal{R}$ V-receives m' from P, with $m \neq m'$, then $|U \cup V| \geq b$.*

4.1 Predicate LEVELS

Our protocols follow a specific pattern. They first allow the sender S to send its input m to each subset of $b-1$ recipients. From now on, only the recipients interact with each other. Whenever a condition C_1 is met for R_i, it sends a message via all available b-cast channels, and as soon as a (stricter) condition C_2 is met, it outputs its final message.

At the core of the conditions is the predicate `LEVELS`, which is reminiscent of the notion of *proxcast* [6]. Intuitively, the predicate `LEVELS` can be understood as an indicator of the consistency level achieved so far. Level 1 is the strongest, and indicates that from the view of R_i, the sender "looks honest". Level 2 indicates that there might be an honest receiver R_j for whom the sender looks honest, i.e., that R_j is at level 1. Level 3 indicates that there might be an honest R_k at level 2, and so on.

Roughly speaking, to achieve level 1, R_i checks that it 1-received a message m from the sender S, and that $n-t$ parties *confirm to be* at level 1 as well. Intuitively, a recipient R_i is at level $l > 1$ if from its point of view there could be an honest recipient that is at level $l-1$. Note that if an honest recipient is at level $l-1$, then all the $n-t$ honest recipients eventually receive enough messages from the sender and enough confirmations to place themselves on level $l-1$ or l. Hence, for level l, R_i needs to l-receive m from S and there must be a sequence of subsets of parties $L_1, \ldots, L_l$, where L_k can be interpreted as a set containing parties confirming to be at level k, that satisfies $|L_1| \geq n-t$ and $\forall\ 1 \leq k \leq l-1 : |L_k| + |L_{k+1}| \geq n-t$. For technical reasons, it will be useful to consider different sizes for each of the two conditions above (see Figure 1).

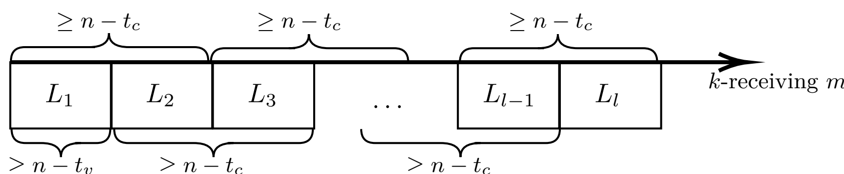


Figure 1 A visual representation of the `LEVELS` predicate.

More concretely, the predicate satisfies level l for parameters n, t_v and t_c if:

$$\begin{aligned}\texttt{LEVELS}_{n,t_v,t_c}(L_1,\ldots,L_l) = &(\forall\ 1 \leq k < k' \leq l : L_k \cap L_{k'} = \varnothing) \wedge \\ &\big(|L_1| \geq n-t_v\big) \wedge \big(\forall\ 2 \leq k \leq l : |L_k| \geq 1\big) \wedge \\ &\big(\forall\ 1 \leq k \leq l-1 : |L_k| + |L_{k+1}| \geq n-t_c\big)\end{aligned}$$

For $l = 0$, the predicate is `true` by default.

We add a few properties. Their proofs are enclosed in the full version of the paper.

▶ **Lemma 9.** *If $\texttt{LEVELS}_{n,t_v,t_c}(L_1,\ldots,L_l)$ holds, then $\texttt{LEVELS}_{n,t_v,t_c}(L_1,\ldots,L_k)$ holds for any $0 \leq k \leq l$.*

We denote by $\lambda(l)$ the minimum number of recipients that can be placed into sets $L_1, \ldots, L_l$ satisfying $\texttt{LEVELS}_{n,t_v,t_c}(L_1, \ldots, L_l)$. That is, the minimum $\left|\bigcup_{k=1}^{l} L_k\right|$, for sets $L_1, \ldots, L_l$ satisfying $\texttt{LEVELS}_{n,t_v,t_c}(L_1, \ldots, L_l)$. Naturally, $\lambda(0) = 0$. The next lemma computes $\lambda(l)$, $l > 0$.

▶ **Lemma 10.** *Given $0 \leq t_c \leq t_v < n$ and $l > 0$, $\lambda(l)$ can be computed as follows.*

If $t_v = t_c = t$:

$$\lambda(l) = \begin{cases} \frac{l+1}{2}(n-t) & \text{if } l \text{ is odd,} \\ \frac{l}{2}(n-t)+1 & \text{if } l \text{ is even.} \end{cases}$$

Otherwise, if $t_v > t_c$:

$$\lambda(l) = \begin{cases} (n-t_v) + \frac{l-1}{2}(n-t_c) & \text{if } l \text{ is odd,} \\ \frac{l}{2}(n-t_c) & \text{if } l \text{ is even.} \end{cases}$$

5 Asymptotically Optimal Reliable Broadcast Protocol

We present a protocol for any finite input space achieving reliable broadcast in $\mathcal{N}_b$ that is:

- for $3 \leq b \leq 4$, secure up to $t < n/2$ corruptions;
- for $b > 4$ even, secure up to $t < \left(\frac{b-4}{b-2}n + \frac{8}{b-2}\right)$ corruptions;
- for $b > 4$ odd, secure up to $t < \left(\frac{b-3}{b-1}n + \frac{6}{b-1}\right)$ corruptions.

5.1 Protocol Description

The protocol generalizes the simple protocol for 3-cast presented in Section 3. In the region of dishonest majority, $n - t < t$ confirmations are not sufficient to make a decision since all confirmations could come from dishonest parties. Instead, we store the recipients that send confirmations and make use of the `LEVELS` predicate to evaluate the consistency level.

Initially, the sender forwards its input m to every subset of $b-1$ recipients via all available b-cast channels. If S is honest, once a recipient R_i 1-receives m from S, it sends $(\texttt{READY}, m)$ via all b-cast channels to notify the other recipients. It outputs m when in addition it 1-receives $(\texttt{READY}, m)$ from $n - t - 1$ other recipients, completing the protocol at level 1.

If S is corrupted, it is possible that R_i is the only honest recipient that 1-receives m from S and 1-receives $(\texttt{READY}, m)$ from the other $n - t - 1$ recipients, which are corrupted. However, any other honest recipient R_j eventually 2-receives m from S, 2-receives $(\texttt{READY}, m)$ from the $n - t - 1$ corrupted recipients and 1-receives $(\texttt{READY}, m)$ from R_i. Once R_j receives these messages, it sends $(\texttt{READY}, m)$ and outputs m, completing the protocol at level 2.

Following this line of reasoning, for $l > 1$, an honest recipient R_i that l-receives m from S sends $(\texttt{READY}, m)$ when it believes that an honest recipient R_j completed the protocol on level $l - 1$. Then, R_i outputs m when it is sure that any honest recipient that eventually $(l + 1)$-receives m from S will have enough evidence that R_i terminated at level l. This guarantees that when an honest recipient R_i completes the protocol with output m, every honest recipient eventually sends $(\texttt{READY}, m)$. Additionally, we set the threshold so that it ensures that honest recipients cannot send `READY` for different messages. Moreover, if all honest recipients send $(\texttt{READY}, m)$, all honest recipients eventually output m.

Each recipient R_i keeps sets $\mathbf{R}^i(m, k)$, $1 \leq k \leq b$, where it stores the recipients from whom it k-received $(\texttt{READY}, m)$. We define two predicates which will be helpful when describing the protocol. Predicate $\texttt{DONE}^i(m, l)$ indicates that the l levels have been completed for the message m, and hence R_i can complete the protocol. Predicate $\texttt{NOTIFY}^i(m, l)$ indicates that there is a seemingly honest recipient R_j who satisfied the predicate $\texttt{DONE}^j(m, l - 1)$, meaning that R_i should send a notification for level l and message m. In the following, we formally describe the two predicates `NOTIFY`, `DONE` (see Figure 2).

$$
\begin{aligned}
\texttt{NOTIFY}^i(m,1) &= \texttt{true}\\
\texttt{NOTIFY}^i(m,l) &= \exists\, L_1^i \subseteq \mathbf{R}^i(m,l), \ldots, L_k^i \subseteq \mathbf{R}^i(m,l-k+1), \ldots, L_{l-1}^i \subseteq \mathbf{R}^i(m,2) \text{ s.t.}\\
&\qquad \big(\forall\, 1 \le k \le l-1 : L_k^i \cap \mathbf{R}^i(m,l-k) \neq \varnothing\big) \wedge\\
&\qquad \texttt{LEVELS}_{n,t,t}(L_1^i, \ldots, L_{l-1}^i) \text{ holds.}\\
\texttt{DONE}^i(m,l) &= \exists\, L_1^i \subseteq \mathbf{R}^i(m,l), \ldots, L_k^i \subseteq \mathbf{R}^i(m,l-k+1), \ldots, L_l^i \subseteq \mathbf{R}^i(m,1) \text{ s.t.}\\
&\qquad \big(\forall\, 1 \le k \le l-1 : L_k^i \cap \mathbf{R}^i(m,l-k) \neq \varnothing\big) \wedge\\
&\qquad \texttt{LEVELS}_{n,t,t}(L_1^i, \ldots, L_l^i) \text{ holds.}
\end{aligned}
$$

Figure 2 The condition $\texttt{NOTIFY}^i(m,l)$, in contrast to the condition $\texttt{DONE}^i(m,l)$.

Protocol $\Pi_{\texttt{RBC}}^{n,b}$

Code for the sender S

1: On input m, send m to every subset of $b-1$ recipients via b-cast and terminate with output m.

Code for recipient R_i

Initialize $\mathbf{R}^i(m,k) = \varnothing$ for any m and $1 \le k \le b$.

1: Upon k-receiving $(\texttt{READY}, m)$ from R_j, add R_j to $\mathbf{R}^i(m,k)$.
2: As soon as $\texttt{NOTIFY}^i(m,l)$ holds and m was l-received from S (for a message m), send $(\texttt{READY}, m)$ to every set of $b-1$ recipients via b-cast and add R_i to $\mathbf{R}^i(m,k)$ for $1 \le k \le b$.
3: As soon as $\texttt{DONE}^i(m,l)$ holds and m was l-received from S (for a message m), output m and terminate.

5.2 Resilience Proof

We divide the proof in two cases: when $3 \le b \le 4$, and when $b > 4$. We first state a number of intermediate lemmas that help in the proofs of both cases. Note that in our protocol, corrupted recipients can send inconsistent READY messages. However, they are limited by the following property implied by Lemmas 7 and 8.

▶ **Lemma 11.** *If $R \in \mathbf{R}^i(m,k)$ for an honest recipient R_i and for $k < b$, then for any honest recipient R_j, eventually $R \in \mathbf{R}^j(m,k+1)$ and $R \notin \mathbf{R}^j(m',k')$, where $m' \neq m$ and $k' < b-k$.*

Intuitively, the following property ensures that if no honest recipient can place itself on level $b-2$ for a message m, then the honest recipient cannot send $(\texttt{READY}, m)$ or output m. The proof is enclosed in the full version of the paper.

▶ **Lemma 12.** *Assume that $t < \lambda(b-2)$ and let U denote the smallest set containing an honest recipient such that the sender U-sends m. If $|U| \geq b-1$, then no honest recipient can send $(\texttt{READY}, m)$ or complete the protocol with output m.*

The next property ensures that if an honest recipient outputs m on level l, then all the honest recipients can place themselves on level at most $l+1$ for m and send $(\texttt{READY}, m)$. It is proven in the full version of the paper.

▶ **Lemma 13.** *If $t < \lambda(b-2)$ and an honest recipient R_i completes the protocol with output m, then any other honest recipient eventually sends $(\texttt{READY}, m)$.*

We now show that once an honest recipient has terminated with output m on level l, every honest recipient can output m, intuitively by placing itself on level at most $l+1$.

▶ **Lemma 14.** *If $t < \lambda(b-2)$ and every honest recipient sends $(\texttt{READY}, m)$, then every honest recipient can output m.*

Proof. Let R_i denote the first honest recipient that sends $(\texttt{READY}, m)$. Then, R_i l-receives m and $\texttt{NOTIFY}^i(m, l)$ holds for $l < b-1$, according to Lemma 12. According to the definition of $\texttt{NOTIFY}^i(m, l)$, $\exists\ L_1^i \subseteq \mathbf{R}^i(m, l)$, ..., $L_k^i \subseteq \mathbf{R}^i(m, l-k+1)$, ..., $L_{l-1}^i \subseteq \mathbf{R}^i(m, 2)$ such that $L_k^i \cap \mathbf{R}^i(m, l-k) \neq \varnothing$ for every $1 \leq k \leq l-1$ and $\texttt{LEVELS}_{n,t,t}(L_1^i, ..., L_{l-1}^i)$ holds. Since R_i is the first honest recipient to send $(\texttt{READY}, m)$, every recipient in $\bigcup_{k=1}^{l-1} L_k^i$ is corrupted.

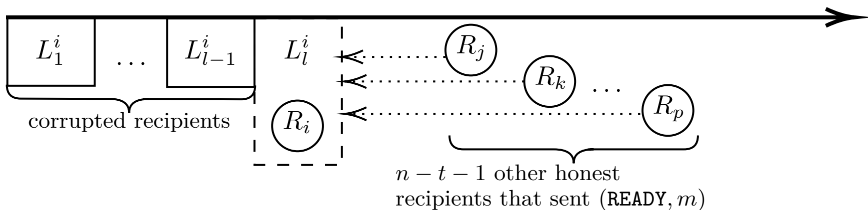


Figure 3 All the honest recipients have sent $(\texttt{READY}, m)$.

As shown in Figure 3, since every honest recipient sends $(\texttt{READY}, m)$, eventually $\left|\mathbf{R}^j(m,1) \setminus \left(\bigcup_{k=1}^{l-1} L_k^i\right)\right| \geq n-t$ for any honest recipient R_j. We can assign $n-t-|L_{l-1}^i|$ honest recipients to L_l^i. Then, $\texttt{DONE}^i(m, l)$ holds and R_i can output m. Every honest recipient R_j can achieve $\texttt{DONE}^j(m, l+1)$ by assigning $L_k^j = L_k^i$, according to Lemma 11, and by assigning to L_{l+1}^j the honest recipients that are not in L_l^i. Then, $|L_l^j \cup L_{l+1}^j| \geq n-t$ and $\texttt{DONE}^j(m, l+1)$ holds. Additionally, from Lemma 7, R_j $(l+1)$-receives m and therefore it can output m. ◀

5.2.1 Resilience for $3 \leq b \leq 4$

▶ **Lemma 15.** *If $t < \lambda(1)$, validity is satisfied.*

Proof. Let m denote the input of the honest sender and let $m' \neq m$. Since the recipients only b-receive m and $t < \lambda(1) \leq \lambda(b-2)$, no honest recipient sends $(\texttt{READY}, m')$ or outputs m', according to Lemma 12. Every recipient can 1-receive m from S, hence every honest recipient eventually sends $(\texttt{READY}, m)$. From Lemma 14, we obtain that every honest recipient eventually outputs m. ◀

▶ **Lemma 16.** *If $t < \lambda(1)$, consistency is satisfied.*

Proof. Assume that an honest recipient R_i completes the protocol with output m. Since $t < \lambda(1)$, we can assume without loss of generality that R_i has 1-received m.

According to Lemma 8, no honest recipient can $(b-2)$-receive $m' \neq m$ and since $t < \lambda(1) \leq \lambda(b-2)$, we obtain from Lemma 12 that no honest recipient sends $(\texttt{READY}, m')$ or outputs m'. According to Lemma 13, every honest recipient sends $(\texttt{READY}, m)$. It follows from Lemma 14 that every honest recipient outputs m. ◀

We immediately obtain the following result from lemmas 15 and 16.

▶ **Theorem 17.** *Let $3 \leq b \leq 4$. $\Pi^{n,b}_{\texttt{RBC}}$ is a reliable broadcast protocol secure up to $t < \lambda(1)$ corruptions in $\mathcal{N}_b$.*

From Lemma 10, which states that $t < \lambda(1) = n - t$, and Theorem 17 we obtain:

▶ **Corollary 18.** *If $3 \leq b \leq 4$, $\Pi^{n,b}_{\texttt{RBC}}$ is a reliable broadcast protocol secure up to $t < \frac{n}{2}$ corruptions in $\mathcal{N}_b$.*

5.2.2 Resilience for $b > 4$

We firstly add a few properties that will be useful in proving that our protocol achieves asynchronous broadcast. For certain thresholds, the READY messages the honest recipients send are unique and *consistent* with respect to the messages that the honest parties output upon termination. That is, if an honest recipient outputs m, then it sent $(\texttt{READY}, m)$, while if an honest party sends $(\texttt{READY}, m)$, no honest party can output $m' \neq m$.

The proofs of the following lemmas are enclosed in the full version of the paper.

▶ **Lemma 19.** *Assume that $t < \lambda(l-1) + \lambda(l'-1) - \lambda(l+l'-b+1) + 2$ for any $l > 0$, $l' > 0$ such that $l + l' \geq b$. Then, if $\texttt{NOTIFY}^i(m, l)$ holds for an honest recipient R_i, $\texttt{NOTIFY}^j(m', l')$ is false for any honest recipient R_j, where $m \neq m'$.*

▶ **Lemma 20.** *Assume that $t < \lambda(l-1) + \lambda(l'-1) - \lambda(l+l'-b+1) + 2$ for any $l > 0$, $l' > 0$ such that $l + l' \geq b$. Then, if an honest recipient R_i outputs m, it sent $(\texttt{READY}, m)$.*

▶ **Lemma 21.** *Assume that $t < \lambda(l-1) + \lambda(l'-1) - \lambda(l+l'-b+1) + 2$ for any $l > 0$, $l' > 0$ such that $l + l' \geq b$. Then, if an honest recipient R_i sends $(\texttt{READY}, m)$, no honest recipient completes the protocol with output $m' \neq m$.*

We now show the conditions for our protocol to achieve validity and consistency.

▶ **Lemma 22.** *If $t < \lambda(b-2)$, validity is satisfied.*

Proof. Let m denote the input of the honest sender and let $m' \neq m$. According to Lemma 12 and since the recipients only b-receive m', no honest recipient sends $(\texttt{READY}, m')$ or outputs m'. Every honest recipient eventually sends $(\texttt{READY}, m)$, as they can 1-receive m from S. From Lemma 14, we obtain that every honest recipient eventually outputs m. ◀

▶ **Lemma 23.** *If $t < \lambda(b-2)$ and $t < \lambda(l-1) + \lambda(l'-1) - \lambda(l+l'-b+1) + 2$ for every $l, l' > 0$ such that $l + l' \geq b$, consistency holds.*

Proof. Assume that an honest recipient R_i completes the protocol with output m. According to Lemmas 19 and 21, no honest recipient sends $(\texttt{READY}, m')$ or outputs m', where $m' \neq m$. Consequently, it follows from Lemmas 20 and 13, that every honest recipient sends $(\texttt{READY}, m)$. Then, we obtain from Lemma 14 that every honest recipient outputs m. ◀

The next result follows immediately from Lemmas 22 and 23.

▶ **Theorem 24.** *If $b > 4$, $t < \lambda(b-2)$ and $t < \lambda(l-1) + \lambda(l'-1) - \lambda(l+l'-b+1) + 2$ for any $l, l' > 0$ such that $l + l' \geq b$, $\Pi^{n,b}_{\texttt{RBC}}$ achieves reliable broadcast in $\mathcal{N}_b$.*

In the full version of the paper, we show that $t < \frac{b-4}{b-2}n + \frac{8}{b-2}$ (resp. $t < \frac{b-3}{b-1}n + \frac{6}{b-1}$) implies the hypothesis of Theorem 24. Hence, we obtain the following:

▶ **Corollary 25.** *Let $b > 4$. If b is even (resp. odd), $\Pi^{n,b}_{\texttt{RBC}}$ achieves resilient reliable broadcast in $\mathcal{N}_b$ secure against $t < \frac{b-4}{b-2}n + \frac{8}{b-2}$ (resp. $t < \frac{b-3}{b-1}n + \frac{6}{b-1}$) corruptions.*

6 Optimal Nonstop Broadcast

In this section, we present a protocol that achieves nonstop reliable broadcast secure against $t < \frac{b-1}{b+1}n$ corruptions. In the full version of the paper, we show that the construction is optimal with respect to the corruption threshold.

The construction follows an information gathering approach, where parties recursively invoke a two-threshold nonstop reliable broadcast, along the lines of [6]. The main difference relies on the fact that our construction needs to handle the fact that not all partial channels need to give output. As a consequence, instead of relying on the intermediate abstraction *proxcast*, which provides a fixed level of consistency at a known point in time, parties need to keep track of the received messages, and continue sending messages whenever *the consistency level increases*, which makes the overall combinatorial analysis substantially more complex.

We denote a (t_v, t_c)-nonstop reliable broadcast a protocol achieving validity (resp. consistency) up to t_v (resp. t_c) corrupted recipients. For simplicity, in this section, we focus on protocols with binary input domain. One can always extend it to any finite domain by invoking the protocol in parallel for each bit of the message to be sent.

6.1 Protocol Description

Initially, the sender forwards his input via b-cast to every subset of $b-1$ recipients. Each recipient R_i, now as sender, recursively invokes the two-threshold broadcast protocol with parameters $t'_v = \min(t_v, n-2)$ and $t'_c = t_c - 1$ towards the $n-1$ left recipients, to distribute messages. The idea is that if R_i is honest, then validity holds in the recursive calls up to t_v corruptions if $t_v < n-1$, and up to $n-2$ corruptions out of the $n-1$ recipients otherwise. On the other hand, if R_i is dishonest, then among the $n-1$ recipients there is one less corrupted party, and so resilience up to t'_c corruptions is enough.

The message that each R_i distributes indicates a level l of confidence for a message m, meaning that it l-receives m from the sender, and did not send a message for $(b-l)$-receiving $m' \neq m$. In particular, R_i sends $(\texttt{READY}, k, m_k, l_k)$ to announce that it l_k-received m_k from the sender and that this is the k-th message that it sends. We think about the protocol as the recipients having available a designated channel for each level, so that any receiver can verify the order of the messages. Following the reasoning presented in Section 4.1, a recipient outputs once it receives enough confirmations that other recipients achieved the same consistency level. Given that the recursive broadcast works, it is then guaranteed that when an honest recipient outputs m with level l, all the other honest recipients confirm with level $l+1$, and eventually every honest recipient outputs at level $l+1$. Intuitively, the protocol does not allow parties to terminate because recipients do not know whether they will need to confirm messages for other recursive calls of the protocol (there might be some honest recipient still waiting for a confirmation message to terminate).

Formal condition to output. Formally, a recipient R_i outputs m on level l when it l-receives m from the sender and the following predicate is satisfied:

$$\mathtt{DONE}^i(m,l) = \ \exists\ L_1 \subseteq \mathbf{R}^i(m,1), \ldots, L_l \subseteq \mathbf{R}^i(m,l) \text{ s.t. } \mathtt{LEVELS}_{n,t_v,t_c}(L_1,\ldots,L_l) \text{ holds.}$$

▶ **Definition 26.** *A message* $(\mathtt{READY}, k, m_k, l_k)$ *is consistent with respect to the set of messages* $\{(\mathtt{READY}, k', m_{k'}, l_{k'}) \mid 1 \le k' < k\}$ *if for every such* k', $l_k + l_{k'} > b$ *when* $m_{k'} \neq m_k$ *and* $l_k \neq l_{k'}$ *when* $m_{k'} = m_k$.

▶ **Definition 27.** *A recipient* R_i *accepts a message* $(\mathtt{READY}, k, m_k, l_k)$ *from a recipient* R_j *if it has received the messages* $\{(\mathtt{READY}, k', m_{k'}, l_{k'}) \mid 1 \le k' \le k\}$ *from* R_j *and* $(\mathtt{READY}, k, m_k, l_k)$ *is consistent with respect to them.*

▶ **Definition 28.** $\mathbf{R}^i(m,l) \subseteq \mathcal{R}$ *denotes the set of recipients that* R_i *has accepted a message* $(\mathtt{READY}, \cdot, m, l)$ *from. We use* $\mathbf{R}^i(m, \le l)$ *to denote* $\bigcup_{j=1}^{l} \mathbf{R}^i(m,j)$.

We now formally present our protocol $\Pi^{n,b}_{\mathtt{nRBC}}(t_v, t_c, S, \mathcal{R})$ for n recipients in the communication network $\mathcal{N}_b$, where S is the sender, and $\mathcal{R}$ is the set of recipients.

Protocol $\Pi^{n,b}_{\mathtt{nRBC}}(t_v, t_c, S, \mathcal{R})$

Code for the sender *S*

1: On input m, send m via b-cast to every subset of $b-1$ recipients.

Code for recipient $R_i \in \mathcal{R}$

Initialize $k = 0$ and $\mathbf{R}^i(m,l) = \varnothing$ for every m and $1 \le l \le b$.

1: **if** $n = b$ **then**
2: Upon receiving a value from the b-cast with sender S and recipient set $\mathcal{R}$, output m.
3: **end if**
4: When receiving $(\mathtt{READY}, p, m_p, l_p)$ from R_j, wait until receiving $M_j = \{(\mathtt{READY}, p', m_{p'}, l_{p'}) \mid 1 \le p' < p\}$ from R_j. If $(\mathtt{READY}, p, m_p, l_p)$ is consistent with respect to M_j, add R_j to $\mathbf{R}^i(m_p, l_p)$.
5: When l-receiving m from the sender such that $(\mathtt{READY}, k+1, m, l)$ is consistent with respect to your previously sent messages, send $(\mathtt{READY}, k+1, m, l)$ to the other recipients by invoking $\Pi^{n-1,b}_{\mathtt{nRBC}}(\min(t_v, |\mathcal{R} \setminus \{R_i\}| - 1), t_c - 1, R_i, \mathcal{R} \setminus \{R_i\})$, add R_i to $\mathbf{R}^i(m,l)$, and increment k.
6: When observing for the first time that $\mathtt{DONE}^i(m,l)$ holds and you have l-received m from the sender, output m.

6.2 Resilience Proof

In this section, we define a predicate $\mathtt{Q}^b_n(t_v, t_c)$ which reflects the conditions that t_v and t_c must satisfy such that $\Pi^{n,b}_{\mathtt{nRBC}}$ achieves (t_v, t_c)-nonstop reliable broadcast.

$\mathtt{Q}^b_b(t_v, t_c) = \mathtt{true}$ since $\Pi^{n,b}_{\mathtt{nRBC}}$ implements an ideal b-cast. Then, for $n > b$, $\mathtt{Q}^b_n(t_v, t_c) = \bigwedge_{k=0}^{n-b-1} \mathtt{P}^b_{n-k}(\min(t_v, n-k-1), t_c)$, where $\mathtt{P}^b_n(t_v, t_c)$ denotes a *local* predicate enclosing the conditions that t_v and t_c must satisfy assuming that $\Pi^{n-1,b}_{\mathtt{nRBC}}$ achieves (t_v, t_c)-nonstop reliable broadcast. We prove that $\mathtt{P}^b_n(t_v, t_c)$ can be defined as follows.

$$\begin{aligned}\mathtt{P}^b_n(t_v, t_c) = \ & [t_v < \lambda(b-1) \lor n < \lambda(b)] \ \land \\ & \big[\forall 1 \le l \le b: \ (t_c < \lambda(l-1) + \lambda(b-l-1) \lor n < \lambda(l) + \lambda(b-l))\big]\end{aligned}$$

6.2.1 Validity

In each result enclosed in this subsection, we assume that the sender is honest, that there are at most t_v corrupted recipients, and that $\mathtt{Q}^b_{n-1}(\min(t_v, |\mathcal{R}| - 2), t_c - 1)$ holds, i.e., that $\Pi^{n-1,b}_{\mathtt{nRBC}}(\min(t_v, |\mathcal{R} \setminus \{R_i\}| - 1), t_c - 1, R_i, \mathcal{R} \setminus \{R_i\})$ achieves validity (resp. consistency) up to $\min(t_v, |\mathcal{R} \setminus \{R_i\}| - 1)$ (resp. $t_c - 1$) corruptions. We show that satisfying $\mathtt{P}^b_n(t_v, t_c)$ suffices for $\Pi^{n,b}_{\mathtt{nRBC}}$ to achieve validity.

▶ **Lemma 29.** *If an honest recipient R_i sends $(\mathtt{READY}, i, m, l)$, then eventually $R_i \in \mathbf{R}^j(m, l)$ for any honest recipient R_j.*

Proof. R_i invokes $\Pi^{n-1,b}_{\mathtt{nRBC}}(\min(t_v, |\mathcal{R}\setminus\{R_i\}|-1), t_c-1, R_i, \mathcal{R}\setminus\{R_i\})$ with at most $\min(t_v, |\mathcal{R}|-2)$ corrupted recipients. It follows that $\Pi^{n-1,b}_{\mathtt{nRBC}}$ achieves validity. Hence, every honest recipient R_j eventually receives the messages sent by R_i and, since R_i is honest, R_j accepts each such message and adds R_i to $\mathbf{R}^j(m, l)$. ◀

▶ **Lemma 30.** *If m is the input of the honest sender, then every honest recipient can output m.*

Proof. Let m denote the input of the honest sender. Hence, the recipients eventually 1-receive m from S and never l-receive $m' \neq m$ for $l < b$. Then, the honest recipients can send $(\mathtt{READY}, \cdot, m, 1)$ as it is consistent with respect to any messages they sent beforehand. Eventually, $\left|\mathbf{R}^i(m, 1)\right| \geq n - t_v$ and therefore $\mathtt{DONE}^i(m, 1)$ holds according to Lemma 29 for every honest recipient R_i. It follows that every honest recipient R_i eventually outputs m. ◀

▶ **Lemma 31.** *If $\mathtt{P}^b_n(t_v, t_c)$ holds and m is the input of the honest sender, then no honest recipient outputs $m' \neq m$.*

Proof. Assume that an honest recipient R_i outputs m'. $\mathtt{DONE}^i(m', b)$ must hold since no recipient l-receives m' from the sender for $l < b$, implying that $n \geq \lambda(b)$. Additionally, $\mathbf{R}^i(m', \leq b - 1)$ consists entirely of corrupted recipients, and since $\mathtt{DONE}^i(m', b)$ implies $\mathtt{DONE}^i(m', b - 1)$, it follows that $t_v \geq \lambda(b - 1)$. Hence, $t_v \geq \lambda(b - 1) \wedge n \geq \lambda(b)$, which contradicts $\mathtt{P}^b_n(t_v, t_c)$. ◀

Finally, using Lemmas 30 and 31, we conclude the following.

▶ **Lemma 32.** *If $\mathtt{P}^b_n(t_v, t_c)$ holds, then $\Pi^{n,b}_{\mathtt{nRBC}}(t_v, t_c, S, \mathcal{R})$ achieves validity.*

6.2.2 Consistency

In each result enclosed in this section, we assume that there are at most t_c corrupted recipients, and that $\mathtt{Q}^b_{n-1}(\min(t_v, |\mathcal{R}| - 2), t_c - 1)$ holds. We show that $\mathtt{P}^b_n(t_v, t_c)$ suffices for $\Pi^{n,b}_{\mathtt{nRBC}}$ to achieve consistency.

Properties of READY Messages. Intuitively, the following lemma shows that the honest recipients eventually receive the same messages after an invocation of the subprotocol.

▶ **Lemma 33.** *If an honest recipient receives m from R_i, all the other honest recipients eventually receive m. Additionally, if R_i is honest, m is the message that R_i sent.*

Proof. R_i invokes $\Pi^{n-1,b}_{\mathtt{nRBC}}(\min(t_v, |\mathcal{R}\setminus\{R_i\}|-1), t_c-1, R_i, \mathcal{R}\setminus\{R_i\})$. There are at most $t_c \leq \min(t_v, |\mathcal{R}| - 2)$ corrupted recipients in $\mathcal{R} \setminus \{R_i\}$ if R_i is honest, and at most $t_c - 1$ otherwise. Since $\mathtt{Q}^b_{n-1}(\min(t_v, |\mathcal{R}| - 2), t_c - 1)$ holds, $\Pi^{n-1,b}_{\mathtt{nRBC}}$ achieves (t_v, t_c)-nonstop broadcast. ◀

Lemma 33 immediately implies that the honest parties eventually accept the same messages from the other recipients. The proof is enclosed in the full version of the paper.

▶ **Lemma 34.** *If $R_j \in \mathbf{R}^i(m,l)$ for an honest recipient R_i, then eventually $R_j \in \mathbf{R}^k(m,l)$ for any honest recipient R_k.*

The following lemma provides a range of levels on which the honest recipients can place themselves for a message m. The result is proven in the full version of the paper.

▶ **Lemma 35.** *Let $U \subseteq \mathcal{R}$ be the smallest set containing an honest recipient such that S U-sends m. Then, for any honest $R_i \in \mathcal{R}$, there is no honest party in $\mathbf{R}^i(m, \leq l-1) \cup \mathbf{R}^i(m', \leq b-l-1)$, where $l = |U|$.*

If l is the smallest level on which an honest recipient can place himself based on the messages of the sender, then eventually the $n - t_c$ honest recipients will place themselves on levels l and $l+1$, as they continue to run the protocol even after they obtain an output.

▶ **Lemma 36.** *Let $U \subseteq \mathcal{R}$ be the smallest set containing an honest recipient such that S U-sends m and let $l = |U|$. Then, it eventually holds that $|\mathbf{R}^i(m, \leq l+1) \setminus \mathbf{R}^i(m, \leq l-1)| \geq n - t_c$ for any honest recipient R_i.*

Proof. Let R_j denote an arbitrary honest recipient. From Lemma 35, $R_j \notin \mathbf{R}^i(m, \leq l-1) \cup \mathbf{R}^i(m', \leq b-l-1)$. R_j eventually $(l+1)$-receives m according to Lemma 7, and it sends $(\texttt{READY}, \cdot, m, l+1)$ since it is consistent with respect to any messages it previously sent. Consequently, $R_j \in \mathbf{R}^i(m, \leq l+1) \setminus \mathbf{R}^i(m, \leq l-1)$ since according to Lemma 33, R_i eventually receives each message that R_j sends. ◀

Our protocol ensures that the notifications of the recipients satisfy the following condition. The proof is enclosed in the full version of the paper.

▶ **Lemma 37.** *If $m \neq m'$ and $l + l' \leq b$, $\mathbf{R}^i(m, \leq l) \cap \mathbf{R}^i(m', \leq l') = \varnothing$ for any honest R_i.*

Properties of the DONE Predicate.

▶ **Lemma 38.** *If $\texttt{DONE}^i(m,l)$ holds for an honest recipient R_i, then $\texttt{DONE}^j(m,l)$ eventually holds for any other honest R_j.*

Proof. Follows from Lemma 34, as R_j eventually receives the same messages as R_i. ◀

▶ **Lemma 39.** *Let $U \subseteq \mathcal{R}$ be the smallest set containing an honest recipient such that S U-sends m and let $l = |U|$. Assume that $\texttt{DONE}^i(m,l)$ holds for an honest R_i.*

If $\texttt{P}^b_n(t_v, t_c)$ holds, then $\texttt{DONE}^i(m', b-l)$ cannot be satisfied for $m' \neq m$.

Proof. Assume that $\texttt{DONE}^i(m', b-l)$ holds. Note that, according to Lemma 37, $\mathbf{R}^i(m, \leq l) \cap \mathbf{R}^i(m', \leq b-l) = \varnothing$. Then, $n \geq |\mathbf{R}^i(m, \leq l) \cup \mathbf{R}^i(m', \leq b-l)| \geq \lambda(l) + \lambda(b-l)$, as shown in Figure 4. Additionally, using Lemma 35, we obtain that every recipient in $\mathbf{R}^i(m, \leq l-1) \cup \mathbf{R}^i(m', \leq b-l-1)$ is corrupted. It follows that $t \geq |\mathbf{R}^i(m, \leq l-1) \cup \mathbf{R}^i(m', \leq b-l-1)| \geq \lambda(l-1) + \lambda(b-l-1)$, contradicting $\texttt{P}^b_n(t_v, t_c)$.

◀

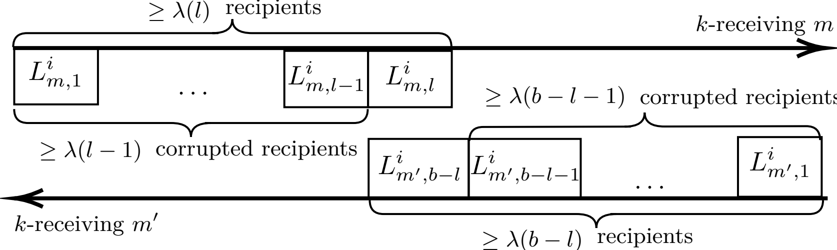


Figure 4 Levels if both $\texttt{DONE}^i(m,l)$ and $\texttt{DONE}^i(m',b-l)$ hold.

Achieving Consistency

▶ **Lemma 40.** *If an honest recipient R_i outputs m, every honest recipient eventually outputs m.*

Proof. Since R_i has output m, it l-received m such that $\texttt{DONE}^i(m,l)$ holds. If every honest recipient can l-receive m, $\texttt{DONE}^i(m,l)$ is enough to guarantee their output, by Lemma 38.

Otherwise, l must be the size of the smallest set $U \subseteq \mathcal{R}$ containing an honest recipient such that S U-sends m, and there is at least one honest recipient that does not belong to any $V \subseteq \mathcal{R}$ such that S V-sends m and $|V| = l$. This recipient however eventually $(l+1)$-receives m, by Lemma 7.

According to Lemma 36, $|\mathbf{R}^j(m, \leq l+1) \setminus \mathbf{R}^j(m, \leq l-1)| \geq n-t$ eventually holds for any honest R_j, and, since $|\mathbf{R}^j(m,l+1)| \geq 1$, $\texttt{DONE}^j(m,l+1)$ holds. It follows that every honest recipient eventually outputs m. ◀

▶ **Lemma 41.** *If $\texttt{P}^b_n(t_v,t_c)$ and an honest recipient R_i outputs m, then no honest recipient outputs $m' \neq m$.*

Proof. Assume that an honest recipient R_j outputs $m' \neq m$. Let $U \subseteq \mathcal{R}$ be the smallest set containing an honest recipient such that S U-sends m and let $l = |U|$. Since R_i completed the protocol, and according to Lemma 9, $\texttt{DONE}^i(m,l)$ holds. It follows from Lemma 38 that $\texttt{DONE}^j(m,l)$ eventually holds as well. According to Lemma 39, $\texttt{DONE}^j(m,b-l)$ must be false. Then, R_j has completed the protocol by $(b-l-1)$-receiving m', contradicting Lemma 8. ◀

Using Lemmas 40 and 41, we conclude the following.

▶ **Lemma 42.** *If $\texttt{P}^b_n(t_v,t_c)$ holds, then $\Pi^{n,b}_{\texttt{nRBC}}(t_v,t_c,S,\mathcal{R})$ achieves consistency.*

6.2.3 Corruption Threshold

We assemble the results proved in the sections 6.2.1 and 6.2.2. The next result follows immediately from Lemmas 32 and 42.

▶ **Theorem 43.** *If $\texttt{Q}^b_n(t_v,t_c)$ holds, then $\Pi^{n,b}_{\texttt{nRBC}}$ achieves (t_v,t_c)-nonstop reliable broadcast.*

In the full version of the paper, we show the technical lemmas that prove that if $2t_v + (b-1)t_c < (b-1)n$, then predicate $\texttt{Q}^b_n(t_v,t_c)$ holds, leaving us with the next theorem.

▶ **Theorem 44.** *If $2t_v + (b-1)t_c < (b-1)n$, then $\Pi^{n,b}_{\texttt{nRBC}}(t_v,t_c)$ achieves (t_v,t_c)-nonstop reliable broadcast in $\mathcal{N}_b$. Additionally, $\Pi^{n,b}_{\texttt{nRBC}}(t,t)$ is a t-resilient nonstop reliable broadcast protocol in $\mathcal{N}_b$, for any $t < \left(\frac{b-1}{b+1}n\right)$.*

References

1. Michael Ben-Or, Ran Canetti, and Oded Goldreich. Asynchronous secure computation. In *25th ACM STOC*, pages 52–61. ACM Press, May 1993. doi:10.1145/167088.167109.
2. Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *20th ACM STOC*, pages 1–10. ACM Press, May 1988. doi:10.1145/62212.62213.
3. Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.
4. Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM (JACM)*, 32(4):824–840, 1985.
5. David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (extended abstract). In *20th ACM STOC*, pages 11–19. ACM Press, May 1988. doi:10.1145/62212.62214.
6. Jeffrey Considine, Matthias Fitzi, Matthew K. Franklin, Leonid A. Levin, Ueli M. Maurer, and David Metcalf. Byzantine agreement given partial broadcast. *Journal of Cryptology*, 18(3):191–217, July 2005. doi:10.1007/s00145-005-0308-x.
7. Danny Dolev and H. Raymond Strong. Authenticated algorithms for byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983.
8. Matthias Fitzi and Ueli M. Maurer. From partial consistency to global broadcast. In *32nd ACM STOC*, pages 494–503. ACM Press, May 2000. doi:10.1145/335305.335363.
9. Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred Aho, editor, *19th ACM STOC*, pages 218–229. ACM Press, May 1987. doi:10.1145/28395.28420.
10. Martin Hirt and Ueli M. Maurer. Player simulation and general adversary structures in perfect multiparty computation. *Journal of Cryptology*, 13(1):31–60, January 2000. doi:10.1007/s001459910003.
11. Alexander Jaffe, Thomas Moscibroda, and Siddhartha Sen. On the price of equivocation in byzantine agreement. In Darek Kowalski and Alessandro Panconesi, editors, *31st ACM PODC*, pages 309–318. ACM, July 2012. doi:10.1145/2332432.2332491.
12. Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 27(2):228–234, 1980.
13. Tal Rabin and Michael Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority (extended abstract). In *21st ACM STOC*, pages 73–85. ACM Press, May 1989. doi:10.1145/73007.73014.
14. D. V. S. Ravikant, M. Venkitasubramaniam, V. Srikanth, K. Srinathan, and C. P. Rangan. On byzantine agreement over (2,3)-uniform hypergraphs. In R. Guerraoui, editor, *Distributed Computing, 18th International Conference, DISC 2004, Amsterdam, The Netherlands, October 4-7, 2004, Proceedings*, volume 3274 of *Lecture Notes in Computer Science*, pages 450–464. Springer, 2004.
15. Pavel Raykov. Broadcast from minicast secure against general adversaries. In Magnús M. Halldórsson, Kazuo Iwama, Naoki Kobayashi, and Bettina Speckmann, editors, *ICALP 2015, Part II*, volume 9135 of *LNCS*, pages 701–712. Springer, Heidelberg, July 2015. doi:10.1007/978-3-662-47666-6_56.

Fast Agreement in Networks with Byzantine Nodes

Bogdan S. Chlebus
School of Computer and Cyber Sciences, Augusta University, GA, USA

Dariusz R. Kowalski
School of Computer and Cyber Sciences, Augusta University, GA, USA
SWPS Uniwersytet Humanistycznospołeczny, Warsaw, Poland

Jan Olkowski
Wydział Matematyki, Informatyki i Mechaniki, Uniwersytet Warszawski, Warsaw, Poland

Abstract

We study Consensus in synchronous networks with arbitrary connected topologies. Nodes may be faulty, in the sense of either Byzantine or proneness to crashing. Let t denote a known upper bound on the number of faulty nodes, and D_s denote a maximum diameter of a network obtained by removing up to s nodes, assuming the network is $(s+1)$-connected. We give an algorithm for Consensus running in time $t + D_{2t}$ with nodes subject to Byzantine faults. We show that, for any algorithm solving Consensus for Byzantine nodes, there is a network G and an execution of the algorithm on this network that takes $\Omega(t + D_{2t})$ rounds. We give an algorithm solving Consensus in $t + D_t$ communication rounds with Byzantine nodes using authenticated messages of polynomial size. We show that for any numbers t and $d > 4$, there exists a network G and an algorithm solving Consensus with Byzantine nodes using authenticated messages in fewer than $t + 3$ rounds on G, but all algorithms solving Consensus without message authentication require at least $t + d$ rounds on G. This separates Consensus with Byzantine nodes from Consensus with Byzantine nodes using message authentication, with respect to asymptotic time performance in networks of arbitrary connected topologies, which is unlike complete networks. Let f denote the number of failures actually occurring in an execution and unknown to the nodes. We develop an algorithm solving Consensus against crash failures and running in time $\mathcal{O}(f + D_f)$, assuming only that nodes know their names and can differentiate among ports; this algorithm is also communication-efficient, by using messages of size $\mathcal{O}(m \log n)$, where n is the number of nodes and m is the number of edges. We give a lower bound $t + D_t - 2$ on the running time of any deterministic solution to Consensus in $(t+1)$-connected networks, if t nodes may crash.

2012 ACM Subject Classification Computing methodologies → Distributed algorithms

Keywords and phrases distributed algorithm, network, Consensus, Byzantine fault, message authentication, node crash, lower bound

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.30

Funding *Dariusz R. Kowalski*: Supported by the Polish National Science Center (NCN) grant UMO-2017/25/B/ST6/02553.

1 Introduction

We consider distributed algorithms solving Consensus in synchronous networks of arbitrary connected topologies. A network has n nodes, each with a unique name. Links connecting nodes are reliable and can transmit messages in both directions. Nodes are prone to failures. We want a distributed algorithm to produce an agreement on a common decision value across the whole network. The feasibility of reaching agreement in a network with faulty nodes depends on its connectivity properties, as showed by Dolev et al. [14]. We consider time

© Bogdan S. Chlebus, Dariusz R. Kowalski, and Jan Olkowski;
licensed under Creative Commons License CC-BY

performance of distributed and deterministic algorithms solving Consensus in such networks for which Consensus solutions exist. The ultimate goal is to optimize time performance, but next to minimize the initial knowledge of nodes and message size.

A network is $(s+1)$-connected if removing at most s nodes does not break it into multiple connected components. We let D_s denote a maximum diameter of a network obtained by removing up to $s > 0$ nodes, assuming the network is $(s+1)$-connected. An upper bound on the number of faulty nodes is denoted either by t or by f. The difference is that t denotes an upper bound on the number of faulty nodes that is known, in being usable in codes of algorithms, while f is a number of faulty nodes actually occurring in an execution and unknown. This convention extends to network properties determined by the numbers of faults, with respect to either being known or unknown. In particular, properties depending on t, like the magnitude of D_{2t}, are known and can be a part of code, while properties depending on f, like the magnitude of D_f, are not known and cannot be referred to directly.

Bounds on the running time of Consensus solutions have been extensively studied in complete networks. It was shown in [1, 20] that $t+1$ rounds are sufficient and necessary to solve Consensus in the case of node crashes. The number of crashes f actually occurring in an execution could be less than t. This leads to the postulate of early stopping, see [17], which can be interpreted as scaling running time to the number of faults: we want all nodes to decide and halt as early as possible, in a number of rounds that depends on f, and possibly on t as well. Early stopping was studied for complete networks under various models of failures, see [7, 8, 17, 24, 28, 35]; it was established that $\min\{f+2, t+1\}$ rounds are sufficient and necessary for all the nodes to reach agreement and halt. We extend to arbitrary connected topologies the concept of scalability of time of a Consensus algorithm to the number of failures in an execution.

A summary of the results. We give an algorithm solving Consensus with Byzantine modes in $t+D_{2t}$ rounds, which is presented in Section 3. The algorithm works under the assumption that node degrees are greater than $3t$. (In complete graphs, the condition on degrees to be greater than $3t$ is equivalent to having $t < n/3$, which is necessary for solvability of Consensus with Byzantine nodes in such networks.) We show that, for any algorithm solving Consensus for Byzantine nodes, there is a network G and an execution of the algorithm on this network that takes $\Omega(t+D_{2t})$ rounds, which is presented in Section 4. We give a Consensus solution with Byzantine nodes using authentication of messages that runs in $t+D_t$ rounds, in networks with node degrees at least $2t$ and using messages of size polynomial in n, see Section 5. We show that, for any t and $d > 4$, there exists a network G and an algorithm solving Consensus with Byzantine nodes using authenticated messages in fewer than $t+3$ rounds on G, but every algorithm solving Consensus without message authentication requires at least $t+d$ rounds on this network. This separates the model of networks with Byzantine nodes from the model with Byzantine nodes using message authentication, with respect to asymptotic time performance of Consensus solutions in networks of suitably connected topologies; such a difference does not hold for complete networks. We develop an early-stopping algorithm solving Consensus against crash failures and running in time $\mathcal{O}(f+D_f)$, where nodes know their names and can differentiate among ports, see Section 6. This algorithm is communication-efficient, in that nodes use messages of $\mathcal{O}(m \log n)$ size, where n is the number of nodes and m is the number of edges. We give a lower bound $t+D_t-2$ on the running time of any deterministic solution to Consensus in $(t+1)$-connected networks, if up to t nodes may crash, see Section 7. Our algorithms for Consensus with Byzantine nodes and Consensus with Byzantine nodes along with message authentication rely only on nodes knowing their names and the parameters

t and also either the bound D_t or D_{2t}, unlike previously known solutions in these models, which assumed knowledge of the whole network's topology. The results of this paper, along with some other relevant facts, are summarized in Table 1.

Table 1 A summary of algorithms and lower bounds. The meaning of notations is as follows: letter n denotes the number of nodes, m is the number of links, t denotes a known upper bound on the number of faulty nodes, and f an actual number of node failures in an execution. The asterisk $*$ marks the contributions of this paper.

model	algorithm	time performance	message size	connectivity	remarks	lower bound on time
Byzantine faults	Dolev et al. [14]	$t \cdot D_{2t}$	$\mathcal{O}(n^5)$	$2t+1$	topology is known	-
	FAST-BYZANTINE Section 3 $*$	$t + D_{2t}$ $*$	exponential	$2t+1$	node degrees $\geq 3t$ D_{2t} is known	$\Omega(t + D_{2t})$ Section 4 $*$
Byzantine local broadcast	Khan et al [25]	exponential	exponential	$\lfloor \frac{3}{2}t \rfloor + 1$	node degrees $\geq 2t$ is necessary	-
	Khan et al. [25]	$\mathcal{O}(n)$	exponential	$2t$	-	-
Byzantine message authentication	Bansal et al. [4]	-	-	-	topology is known monitoring model	-
	FAST-AUTHENTICATED Section 5 $*$	$t + D_t$ $*$	polynomial in n $*$	$t+1$	node degrees $\geq 2t$ D_t is known	$t + D_t - 2$ $*$
crash failures	EARLY-STOPPING -CRASHES Section 6 $*$	$\mathcal{O}(f + D_f + 3)$ $*$	$\mathcal{O}(m \log n)$ $*$	$f+1$	f actual number of failures f, D_f not known	$f + D_f - 2$ Section 7 $*$

Previous and related work. Consensus has been among most popular algorithmic problems studied in distributed systems and communication networks, see [3, 23, 27, 30]. The problem of distributed agreement in systems prone to faults was first considered by Pease at al. [29], Dolev [14] and Lamport et al. [26]. Pease at al. [29] proposed an algorithm for Byzantine faults that has nodes share all their information acquired over time, known as "exponential information gathering," with a further modification given by Bar-Noy et al. [5]. This approach to algorithm design requires nodes to send exponentially long messages and process an exponential amount of information, in the number of nodes n. Pease at al. [29] and Lamport et al. [26] considered reaching agreement with Byzantine faults in message-passing systems with authenticated messages. Dolev and Strong [18] gave a simple algorithm for Byzantine nodes with authentication of messages; Sirikanth and Toueg [31] showed how to implement that algorithm in the model of Byzantine faults and local broadcast, in which every node sends identical messages to every neighbor in each round.

A bound $t < n/3$ on the number of Byzantine nodes t was shown to be necessary for solvability of Consensus by Pease at al. [29], Dolev [14] and Lamport et al. [26]. Fisher and Lynch [19] proved a lower bound $t+1$ on the number of communication rounds, which holds for crashes. Dolev and Reischuk [16] gave a lower bound $\Omega(nt)$ on the number of communication bits necessary to solve Consensus with Byzantine faults, which becomes $\Omega(n^2)$ if the number of Byzantine nodes satisfies $t = \Omega(n)$. Methods to show impossibilities and lower bounds for distributed-computing problems, including reaching distributed agreement, are reviewed in [2]. Garay and Moses [22] showed how to reach agreement with a polynomial number of communication bits and a polynomial local computation, subject only to the

bound of $t < n/3$ on the number of Byzantine nodes while staying within $t+1$ communication rounds. Next, we review previous work on distributed agreement solutions that scale well with respect to performance metrics. We consider an algorithm scaling its running time well if it is either fast, by running in time $\mathcal{O}(t+1)$, or early stopping, by running in time $\mathcal{O}(f+1)$. Berman and Garay [6] developed an algorithm for Byzantine faults which uses messages carrying just one input value, so a message is of constant size if the range of input values is of constant size; the algorithm works for $t < n/4$. Galil et al. [21] developed an algorithm for crashes using $\mathcal{O}(n)$ messages, thus showing that this number of messages is optimal; this algorithm runs in over-linear time $\mathcal{O}(n^{1+\varepsilon})$, for a parameter $0 < \varepsilon < 1$; that paper gave an early-stopping algorithm of message complexity $\mathcal{O}(n + fn^{\varepsilon})$, for any $0 < \varepsilon < 1$. Chlebus and Kowalski [9] developed a gossiping algorithm coping with crashes and applied it to develop a fast solution to Consensus which sends $\mathcal{O}(n \log^2 t)$ messages, provided that $n - t = \Omega(n)$. Chlebus and Kowalski [10] developed a deterministic algorithm which is early-stopping and globally scales communication by sending $\mathcal{O}(n \log^5 n)$ messages. Chlebus at al. [12] gave a fast deterministic Consensus algorithm that sends $\mathcal{O}(n \log^4 n)$ bits, and showed that no deterministic Consensus algorithm can be locally scalable with respect to message complexity. Chlebus and Kowalski [11] gave a randomized Consensus solution terminating in $\mathcal{O}(\log n)$ expected time, while the expected number of bits that each process sends and receives against oblivious adversaries is $\mathcal{O}(\log n)$, assuming that a bound t on the number of crashes is a constant fraction of the number of nodes n. Dolev and Lenzen [15] showed that any crash-resilient Consensus algorithm deciding in $f + 1$ rounds has worst-case message complexity $\Omega(n^2 f)$. Chlebus et al. [13] gave a scalable quantum algorithm to solve binary Consensus, in a system of n crash-prone quantum processes, which works in $\mathcal{O}(\text{polylog } n)$ time sending $\mathcal{O}(n \text{ polylog } n)$ qubits against the adaptive adversary.

Here is a brief review of previous work on reaching agreement in networks beyond the complete network topologies. The following assumptions about networks and distribution of faults were shown to be necessary and sufficient for solvability of the problem: for Byzantine nodes, networks need to be $(2f+1)$-connected with the number of faulty nodes less than $n/3$, while for Byzantine nodes with authentication of messages and for crash failures they need to be $(f+1)$-connected, see [3, 27]. Dolev [14] showed that solving Consensus with t Byzantine nodes requires at least $3t + 1$ nodes in total and network connectivity at least $(2t + 1)$; see also [20]; that paper [14] gave an algorithm relying on knowing the network's topology and solving Consensus in any network satisfying these conditions. Khan at al. [25] considered Consensus in networks in the local-broadcast model, in which every node sends the same message to every neighbor in a round, including Byzantine nodes. They showed that in order for Consensus to be solvable in that model, a network needs to be connected and with each node's degree of at least $2t$. Their algorithms rely on knowing a network's topology; one algorithm has an exponential running time, and another has $\mathcal{O}(n)$ running time, in networks that are $2f$-connected. Bansai et al. [4] considered Consensus in arbitrary networks with Byzantine faults such that nodes can authenticate messages, subject to allowing an adversary to monitor up to k nodes to forge their messages; the paper gave tight conditions on network connectivity referring to the values of t and k to make Consensus solvable, assuming additionally that the network's topology is known. Tseng and Vaidya [33] studied solvability of Consensus in networks with Byzantine nodes and uni-directional links; such networks are modeled as directed graphs. Surveys of related work on reaching distributed agreement are given in [32, 34].

2 Preliminaries

A distributed system is modeled as a simple graph, in which vertices represent nodes and edges represent bi-directional links connecting pairs of nodes. We use letter n to denote the number of vertices and letter m for the number of edges. A graph G is $(s+1)$-*connected* if removing up to s nodes from G does not produce a subgraph of G with at least two connected components. For an $(s+1)$-connected graph G, the notation $D(G,s)$, for an integer $s>0$, denotes the s-*diameter of* G, which is a maximum diameter of a graph obtained by removing s nodes from G. If an $(s+1)$-connected graph G is understood from context, then we use the notation D_s for $D(G,s)$.

Networks are synchronous, in that an execution of a communication algorithm is partitioned into global rounds; an execution starts for all nodes at the same round. During a round, a node may send messages to all neighbors, including itself, and receive all messages sent to it in this round. Links are considered fully reliable, in that no messages are lost, duplicated, nor otherwise modified or corrupted.

Nodes are prone to failures. A node *crashes* at a round when it stops all activity beyond this round and never resumes it again in an execution; some messages sent by a node in a round it crashes may be delivered. A node fault is *arbitrary* or *Byzantine* if the node may undergo arbitrary state transitions in an execution.

We say that a network is equipped with a mechanism to *authenticate messages* if a copy of a message received by a node can be embedded in its future messages such that the authenticity of the contents of the included copy can be verified beyond doubt. We abstract from a mechanism to implement authentication of messages, simply assuming that it is available to every node in a network. If Byzantine faults of nodes are combined with authentication of messages, this is understood such that faulty nodes cannot forge messages, which imposes a restriction on how "arbitrary" their state transitions can be.

In the problem of *Consensus*, each node p is given an *input value* denoted $\texttt{input}_p$. We say that a node *decides on a value* x if it sets a dedicated private variable to this value x; such a decision is considered irrevocable in an execution. Informally, the goal for all nodes is to eventually decide on one common value. We use standard specifications of Consensus expressed as agreement, validity, and termination, depending on the kind of faults, which are either Byzantine or crashes; see [3, 27, 30].

Our algorithms are deterministic and their performance is measured in the worst-case sense. One performance metric is *time*, understood as a number of communication rounds until termination. Another performance metric is *message size*, meant to be an upper bound on the number of bits a node transmits to a neighbor in a round.

We say that some aspect of a distributed system is *known* if nodes can use it in the code of an algorithm. Each node is equipped with a unique *name*, which it knows. We assume a name can be encoded by $\mathcal{O}(\log n)$ bits if transmitted in messages. A node communicates with its neighbors by transmitting messages via ports, one port per neighbor. Ports at a node are distinguishable in the following sense: if a node wants to send a message then it can specify by which port the message is to be transmitted, and if a node receives a message then the node can identify which port delivered the message.

Letter t denotes a known upper bound on the number of faulty nodes; if an algorithm refers to t then its correctness needs to hold only in executions in which up to t nodes are faulty. Letter f denotes a number of faults actually occurring in an execution; this parameter f is not known. By analogy, if we refer to network parameters like D_t or D_f then D_t is assumed to be known but D_f is not assumed to be known. If nodes only know their

names and can distinguish ports then this is the model of *minimal knowledge*. We say that a node *knows its neighbors* if it can map ports on the names of nodes that receive messages transmitted via each of these ports. If neighbors are not initially known, then this can be discovered by having every node send its name to every neighbor, which takes one round but contributes $\mathcal{O}(m \log n)$ bits to total communication.

3 Fast Byzantine Consensus

We present a distributed algorithm Fast-Byzantine solving Consensus in arbitrary networks with Byzantine node faults in asymptotically-optimal time. Consensus in arbitrary networks with bi-directional links was already studied by Lamport et al. [26] and Dolev in [14]. These papers concentrated on solvability of Consensus, as determined by network's connectivity, and did not attempt to optimize time and communication performance; most importantly, they gave algorithms relying on knowing network's topology. We propose an approach to solve Consensus in a deterministic and distributed manner that does not require knowing network's topology, and works assuming sufficiently strong connectivity, while each node's degree is at least $3t$. We use a paradigm to solve Consensus in complete networks with Byzantine nodes, which is known as "exponential information gathering," see [5, 29]. An execution is partitioned into two parts. In the first part, which takes $t+1$ rounds, information is exchanged among the nodes. In a round i, nodes store the information gained so far in a a tree of height i. Leaves store the input value that passes trough subsequent nodes with names belonging to the path from the root to a respective leaf. When the first part is over, the trees have height $t+1$. In the second part, the tree is evaluated from the lowest level to the highest one by a local computation in each node such that the value at the root represents a decision. We divide executions of our algorithm into four stages, described next, referring to notations used in the pseudocode of the algorithm Fast-Byzantine given in Algorithm 1. All non-faulty nodes communicate by sending messages of the format $(\overline{p_1 p_2 \dots p_i}, W)$, which is an ordered pair such that $p_1, p_2, \dots, p_i$ is a sequence of names of nodes that forwarded W from p_1 through p_i. A message of this format is *well-formed* if all node names in the sequence $p_1, p_2, \dots, p_i$ are distinct.

The initialization stage. This stage initializes variables $\texttt{Paths}[i]$ representing sets and is performed in the very beginning of the first round. Each set $\texttt{Paths}[i]$ at p, for $2 \le i \le t+1$, will store well-formed pairs $(\overline{s_1 s_2 \dots s_{i-1} s_i}, W)$, where $s_i = p$. The set $\texttt{Paths}[1]$ at node p is initialized to one ordered pair, which has a single-vertex path p as the first component and the input value $\texttt{input}_p$ of node p as the second component. The sets $\texttt{Paths}[i]$, for $2 \le i \le t+1$, are initialized to an empty set each.

The local authorization stage. In this stage, nodes work to deliver their input values to other nodes at distance at most t. A node p considers only messages that passed through $t+1$ different nodes, including itself at the end: these are pairs of the form $(\overline{s_1 s_2 \dots s_{t+1}}, W)$ with all distinct vertices on the path. To collect such pairs, each node tries to propagate its input value along paths of length $t+1$: each node forwards all the received messages it considers legitimate to its neighbors during t consecutive rounds. More precisely, whenever a node p receives a pair $(\overline{s_1 s_2 \dots s_{i-1}}, W)$ at round $i-1$, then it checks two conditions: (1) did the last node on the path s_{i-1} send the message, and (2) is the path $\overline{s_1 s_2 \dots s_{i-1}, p}$ well-formed. If both conditions are met, node p forwards the pair $(\overline{s_1 s_2 \dots s_{i-1}, p}, W)$ to its neighbors in the next round. An upper bound $3t$ on a node's degree ensures that each non-faulty node can propagate its input value in a verifiable manner to other nodes via sufficiently many well-formed sequences of nodes.

Algorithm 1 A pseudocode of algorithm FAST-BYZANTINE for a node p, structured into four stages. A pseudocode of procedure DELIVER is in Algorithm 2. Variable `Leaves`$[q]$ stores paths of nodes, each path starting with node q.

`algorithm` FAST-BYZANTINE

—— initialization ——

1. initialize $\texttt{Paths}[1] = \{(\bar{p}, \texttt{input})\}$
2. initialize $\texttt{Paths}[2], \ldots, \texttt{Paths}[t+1]$ as empty sets

—— local authorization ——

3. **for** $i \leftarrow 1$ **to** t **do**
  a. send $\texttt{Paths}[i]$ to each neighbor
  b. **foreach** neighbor q **do**
    i. receive $\texttt{Paths}_q[i]$ from q
    ii. **foreach** $(\overline{s_1 s_2 \ldots s_i}, W)$ in $\texttt{Paths}_q[i]$ **do**
      if $s_i = q$ and $(\overline{s_1 s_2 \ldots s_i p}, W)$ is well-formed \\ *q is just before p on the path*
        add $(\overline{s_1 s_2 \ldots s_i p}, W)$ to $\texttt{Paths}[i+1]$

—— global communication ——

4. $(\texttt{Nodes}, M_1, \ldots, M_{|\texttt{Nodes}|}) \leftarrow$ DELIVER$(\texttt{Paths}[t+1])$
5. **foreach** q in `Nodes` **do**
  a. construct set $\texttt{Leaves}[q]$ based on sets of paths $M_1, \ldots, M_{|\texttt{Nodes}|}$.

—— local computation ——

6. **foreach** q in `Nodes` **do**
  a. construct $\texttt{Tree}[q]$ based on the set $\texttt{Leaves}[q]$
  b. **for** $i \leftarrow t$ **to** 1 **do** \\ *evaluation of tree for node q*
    i. **foreach** vertex $\overline{s_1 s_2 \ldots s_i}$ of $\texttt{Tree}[q]$ **do**
      A. **if** $\overline{s_1 s_2 \ldots s_i}$ is active `then`
        set $\texttt{resolve}(\overline{s_1 s_2 \ldots s_i})$ to majority among values $\texttt{resolve}(\overline{s_1 s_2 \ldots s_i s_{i+1}})$
        such that $\overline{s_1 s_2 \ldots s_i s_{i+1}}$ is an active vertex of $\texttt{Tree}[q]$
      B. **else** $\texttt{resolve}(\overline{s_1 s_2 \ldots s_i}) \leftarrow \perp$
7. `decision` $\leftarrow$ the majority among values $\texttt{resolve}(\texttt{Tree}[q].\texttt{root})$ for q in `Nodes`

The global communication stage. In this stage, each node p tries to propagate all legitimate messages it has accumulated in the previous stage to all other nodes in the network. To this end, node p invokes procedure DELIVER, which returns a set `Nodes` of nodes that p heard from along with sets $M_1, \ldots, M_{|\texttt{Nodes}|}$ that node p could validate as information they wanted to propagate. Node p discovers what a node q in `Nodes` wants to send by working with a set of paths of nodes traversed by messages that start at node q and arrive at p after suitable forwards. This stage begins by invoking procedure DELIVER, which has its pseudocode in Algorithm 2. It takes a node's name and the information to be sent as parameters, and returns a set `Nodes` of names of nodes along with the information that these nodes wanted to propagate. In order to achieve this, nodes work to propagate messages between any two non-faulty nodes trough all possible paths of length D_{2t}: in consecutive D_{2t} rounds, each node propagates all received valid paths, by first appending its name at the end of a received path to form a new path. The assumed connectivity $2t+1$ ensures that at least $t+1$ paths

Algorithm 2 A pseudocode of procedure DELIVER for a node p. Parameter I denotes information to be sent. Variable M_q stores paths of nodes, each ending with node q. Variable `Relay` denotes a set of pairs received from neighbors to be forwarded. An element $(\overline{s_1 s_2 \ldots q}, W)$ in `Relay` is considered *confirmed* if there are at least $t+1$ received pairs such that all the pairs carry value W and the paths by which they reached node p are disjoint, except for the endpoints s_1 and p.

`procedure` DELIVER (I)

---

1. initialize sets `Relay` $\leftarrow \{(\overline{p}, I)\}$; `Nodes` $\leftarrow \emptyset$
2. **for** $i \leftarrow 1$ **to** D_{2t} **do**
   a. **foreach** x in `Relay` **do** send x to each neighbor unless x was already sent
   b. **foreach** neighbor q **do**
      i. **foreach** $(\overline{s_1 s_2 \ldots s_i}, W)$ received from q **do**
         A. **if** $s_i = q$ and $(\overline{s_1 s_2 \ldots s_i p}, W)$ is well-formed \\ *q is last node on received path*
            then add $(\overline{s_1 s_2 \ldots s_i p}, W)$ to `Relay`
3. **foreach** confirmed $(\overline{s_1 s_2 \ldots s_k}, W)$ in `Relay` **do**
   add s_1 to set `Nodes` ; assign $\mathtt{M}_{s_1} \leftarrow W$
4. `return` $(\mathtt{Nodes}, M_1, \ldots, M_{|\mathtt{Nodes}|})$

---

are free of faulty nodes. Thus, the information the other node q wanted to propagate could be computed by node p by considering the maximum set of received disjoint paths starting at q with the same propagated value, with at least $t+1$ such disjoint paths. A node p that received $(\overline{s_1 s_2 \ldots q}, W)$ from a neighbor considers W a *confirmed value from* s_1 if there are at least $t+1$ pairs received from neighbors such that all the pairs carry value W and the paths by which they reached node p are disjoint, except for the endpoints s_1 and p. When the global communication stage is over, then each node p wants to store a record of the preceding communication about all nodes q in the set `Leaves`$[q]$, such that the input value of q could be retrieved from this set. This is indeed doable in non-faulty nodes q.

The local computation stage. For a node q in `Nodes`, we treat paths in the set `Leaves`$[q]$ as leaves of a tree. More precisely, these are paths of the form $\overline{s_1 s_2 \ldots s_{t+1}}$ such that a pair $(\overline{s_1 s_2 \ldots s_{t+1}}, ?)$ is in `Leaves`$[q]$, where notation ? is a wildcard character. The tree for a node q in `Nodes` is denoted `Tree`$[q]$ and referred to as the *tree for* q. If a sequence x is a prefix of a sequence y then we say that a vertex x is a ancestor of a vertex y and y is a descendant of x; immediate ancestors and descendants are parents and children. Vertex $\overline{q}$ is a root of tree `Tree`$[q]$, denoted `Tree`$[q]$`.root`. The property to be an *active vertex* is defined recursively as follows: each leaf is active and a vertex with at least $t+1$ active children is such as well. Node p associates a value `resolve`$(\overline{s_1 s_2 \ldots s_k})$ with each vertex $\overline{s_1 s_2 \ldots s_k}$ of tree `Tree`$[q]$. This is a unique value such that node p believes node s_k received it from node s_{k-1}, who received this value from node s_{k-2}, who in turn received the value from node s_{k-3}, and so on. A systematic approach to compute `resolve` values is as follows. If $\overline{s_1 s_2 \ldots s_{t+1}}$ is a leaf then `resolve`$(\overline{s_1 s_2 \ldots s_{t+1}})$ equals W taken from the pair $(\overline{s_1 s_2 \ldots s_{t+1}}, W)$ in the set `Leaves`$[q]$. The value of `resolve` for an inner node is the majority of values `resolve` of its children. Finally, a node determines the input value of a node q to `resolve`(`Tree`$[q]$`.root`). A node decides on the majority of the input values of all nodes q in `Nodes`.

▶ **Theorem 1.** *Algorithm* FAST-BYZANTINE *solves Byzantine Agreement in* $t + D_{2t}$ *communication rounds, provided* $n > 3t$.

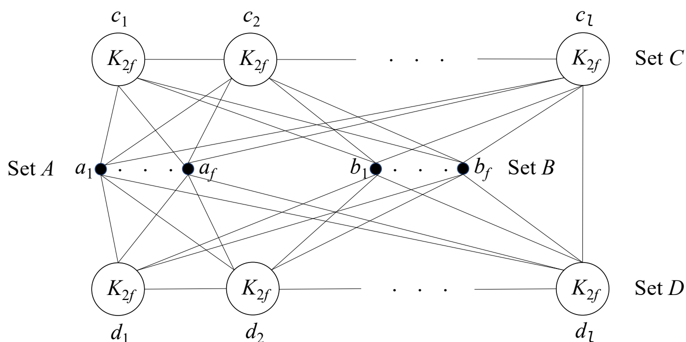


Figure 1 A visualization of graph $G_{t,\ell}$. The parameter t is an upper bound on the number of faulty nodes, while the parameter ℓ denotes a number of cliques in the sets C and D. An edge between either two cliques or a node and a clique indicates that every node at one end is connected with all nodes at the other end.

4 A Lower Bound for Byzantine Faults

In this section, we present a lower bound on time performance of algorithms solving Consensus with Byzantine nodes. The performance of algorithm FAST-BYZANTINE from Section 3 matches this lower bound. This shows that the parameter $D(G, 2t)$ of a network G captures time-optimality of algorithms solving Consensus with Byzantine faults of nodes in networks of general topologies. Let $G_{t,\ell}$ denote a graph with topology as depicted in Figure 1. Its nodes are partitioned into four sets A, B, C, D. Each of the two sets A and B has t elements: $A = \{a_1, \ldots, a_t\}$ and $B = \{b_1, \ldots, b_t\}$. The set C is partitioned into ℓ disjoint subsets, denoted by $c_1, \ldots, c_\ell$, each of $2t$ nodes, and analogously, the set D is also partitioned into ℓ disjoint subsets, denoted by $d_1, \ldots, d_\ell$, each of $2t$ nodes. The edges of the graph are defined as follows:

- for every i such that $1 \leq i \leq \ell$, all pairs of nodes in c_i are connected to produce a clique, and all nodes in d_i are connected to produce a clique;
- for every i such that $1 \leq i \leq \ell - 1$, every node in c_i is connected to all nodes in c_{i+1}, and every node in d_i are connected with all nodes in d_{i+1};
- every node in clique c_ℓ is connected with all nodes in clique d_ℓ;
- for every i such that $1 \leq i \leq \ell$, node a_i and node b_i is connected with all nodes in $C \cup D$.

Graph $G_{t,\ell}$ is $2t + 1$ connected, the parameter $D(G_{t,\ell}, t)$ of such a graph equals 2, regardless of t and ℓ, while $D(G_{t,\ell}, 2t)$ equals 2ℓ, regardless of t.

▶ **Lemma 2.** *For every deterministic algorithm solving Consensus with at most t Byzantine nodes and for every $\ell \geq t$, there exists an execution of the algorithm on some graph $G_{t,\ell}$ with t faulty nodes that takes more than ℓ rounds.*

▶ **Theorem 3.** *For every $t \geq 1$ and $\ell \geq 1$ and any algorithm $\mathcal{A}$ solving Consensus in networks with Byzantine faults there is a network G such that $\ell \geq D(G, 2t)/2$ and an execution of algorithm $\mathcal{A}$ on this network G, with some t faulty nodes, that takes more than $(t + \ell)/2$ rounds to terminate.*

Proof. We consider two cases: either $t \geq \ell$ or $t < \ell$. If $t > \ell$, then we take a a clique of $3t + 1$ nodes as G. A graph obtained from G by removing some $2t$ nodes has diameter 1, so that $D(G, 2t) = 1 \leq \ell$. Any algorithm requires at least $t + 1$ rounds to solve Consensus in this G with t faulty nodes, see [19]. Observe that $(t + \ell)/2 \leq t < t + 1$. If $t \leq \ell$ then we

take graph $G_{t,\ell}$ as G. We rely on the property $D(G, 2t) = 2\ell$, which follows by inspection of the topology of G. By Lemma 2, some execution of algorithm $\mathcal{A}$ takes at least $\ell + 1$ rounds. Observe that $(t + \ell)/2 \leq \ell < \ell + 1$. ◀

5 Fast Authenticated Consensus

We show a separation of the Consensus problem with Byzantine nodes from its version with the same model augmented by authentication of messages. The last part of this section discusses a class of graphs in which time performance of algorithm solving Consensus can be greater, by an arbitrarily large amount, than time performance of an algorithm solving Consensus when supported by authentication of messages, subject to a given number of Byzantine faults.

We use the concept ot authentication as originally proposed by Pease et al. [29]. It assumes the existence of *authenticators* and an oracle creating such authenticators. The authenticator of data d calculated at a node p will be denoted $A_p[d]$. It is considered infeasible for a node q different from p to be able to forge $A_p[d]$. At the same time, each node should be able to check if $d = A_p[d]$ for any other node p. Algorithm Fast-Authenticated has its pseudocode in Algorithm 3. In local authorizing stage the node p tries to make exchange of input values dependable. An input value becomes reliable if passes through a well-formed path of length exactly $f+1$. In order to accomplish this, nodes authenticate received messages and send it to all neighbors in the next round. To avoid exponential message complexity, nodes keep at most one message from every non-faulty source and at most two messages from every node that decided to equivocate. Then the node p analyses the knowledge that it receives in the previous stage and determines a decision.

The algorithm works in four stages. All messages sent by non-faulty nodes are of the following format: $(p_i, a_i, (p_{i-1}, a_{i-1}, \ldots (p_1, a_1, v) \ldots))$, where the node p_1 is the original source of the transmitted value v. The nodes $p_i, \ldots, p_1$ are the ones that propagated the message in consecutive rounds. The value a_j is the authenticator of the message $(p_{j-1}, a_{j-1}, (p_{j-2}, a_{j-2}, \ldots (p_1, a_1, v) \ldots))$, for $2 \leq j \leq i$, while the value a_1 is the authenticator of value v. This format ensures, that consecutive nodes that propagated the message could not fabricate value v. If all nodes $p_1 \ldots p_i$ are different, then the message $(p_i, a_i, (p_{i-1}, a_{i-1}, \ldots (p_1, a_1, v) \ldots))$ is *well-formed*. Non-faulty nodes in a network need to exchange messages. If messages can be authenticated, then network connectivity is the main constraint. Assuming $(f + 1)$-connectivity allows for all non-faulty nodes to communicate in principle without faulty nodes interfering. Procedure Send-To-All implements such communication, its pseudocode is in Algorithm 4. We assume an authentication mechanism with an overhead of a polynomial number of bits per authenticated message. This allows achieving $\mathcal{O}(|M|n^2 \log n)$ message-size complexity, which is polynomial in n, where $|M|$ denotes an upper bound on the number of bits in an authenticated input value.

The local authorizing stage. The nodes work to disseminate the inputs dependably. The mechanism of authentication prevents forging messages, but this does not provide consistency of knowledge about inputs of faulty nodes. To handle this, nodes do not disseminate their input values directly, but instead work also to validate knowledge of their inputs. A validation is provided by passing information through a path of $f + 1$ different nodes. - Such a path contains at least one non-faulty node, which guarantees that later the node spreads the value reliably among other non-faulty nodes. We require each node on this path to confirm forwarding the message.

Algorithm 3 A pseudocode of algorithm Fast-Authenticated for a node p structured into four stages. Procedure Send-To-All has its pseudocode in Algorithm 4.

```
algorithm FAST-AUTHENTICATED

———————————— initialization ————————————
1. initialize set ReceivedMessages[1] = {(p, input, A_p[input])}
2. initialize ReceivedMessages[2], ..., ReceivedMessages[f + 1] to empty sets

———————————— local authorization ————————————
3. for i ← 1 to f do
   a. initialize mapping Inputs to empty
   b. send ReceivedMessages[i] to each neighbor
   c. foreach neighbor q do
      i. Let ReceivedMessages[i]_q be the set of messages received from the node q
      ii. foreach well-formed message (p_i, a_i, (p_{i-1}, a_{i-2}, ... (p_1, a_1, v) ...)) received from q
          and such that p ∉ {p_1, ..., p_i} do
            if a_1 = A_{p_1}[v] and a_j = A_{p_j}[(p_{j-1}, a_{j-1}, ... (p_1, a_1, v) ...)]  for all 2 ≤ j ≤ i
              encrypted-message ←
              (p, (p_i, a_i, (p_{i-1}, a_{i-1}, ... (p_1, a_1, v) ...)), A_p[(p_i, a_i, (p_{i-1}, a_{i-1}, ... (p_1, a_1, v) ...))])
              if Inputs[p_1] = ∅                 \\ this is the first message from the node p_k
                ReceivedMessages[i + 1] ← encrypted-message, Inputs[p_1] ← v
              elseif Inputs[p_1] ≠ v             \\ message does not match the previous one
                ReceivedMessages[i + 1] ← encrypted-message, Inputs[p_1] ←⊥

———————————— global communication ————————————
4. {Nodes, M_1, ..., M_|Nodes|} ← SEND-TO-ALL(p, ReceivedMessage[f + 1])

———————————— local computation ————————————
5. set map Inputs to empty ; set Inputs[p] ← input
6. foreach (p_{f+1}, a_{f+1}, (p_f, a_f, ... (p_1, a_1, v) ...)) ∈ ⊎_{i∈Nodes} M_i do
     if a_1 = A_{p_1}[v] and a_i = A_{p_i}[(p_{i-1}, a_{i-1}, ... (p_1, a_1, v) ...)] : 2 ≤ i ≤ f + 1
       if Inputs[p_1] = ∅ then Inputs[p_1] ← v
       elseif Inputs[p_1] ≠ v then Inputs[p_1] ←⊥
7. decide on the majority of {Inputs[q] : Inputs[q] ≠⊥}
```

The mechanism given above takes f rounds. A node p maintains sets ReceivedMessages$[i]$, for $1 \leq i \leq f+1$, with the messages received after rounds $0, 1, \cdots, f$ respectively. In a round i, the node p sends the set ReceivedMessages$[i]$ to its neighbors. Consider a genuine message $(p_i, a_i, (p_{i-1}, a_{i-2}, \ldots (p_1, a_1, v) \ldots))$ received by a node p in round i. The message is processed as follows: if node p has marked the node p_1 as faulty, it skips the message; if in *this* round the node p has not received yet a message carrying the p_1's input value, it adds the message to the set ReceivedMessages$[i+1]$; if the node p has received a message carrying the p_1's input value before, but the new value does not match the old one, it still adds the message to the set ReceivedMessages$[i+1]$, but also marks the node p_1 as a faulty one (indicated by symbol $\perp$ in the pseudocodes); otherwise node p omitts the message. In case of adding the message to the set ReceivedMessages$[i+1]$, node p confirms the authenticity of communication by adding its name to the list of nodes traversed by the forwarded message

Algorithm 4 A pseudocode of procedure SEND-TO-ALL for a node p. The variable m denotes the information to be sent. The node p checks the genuine of an message by verifying the authenticator of the sender.

```
procedure SEND-TO-ALL(p,m)
```

1. initialize set `ReceivedMessages` $\leftarrow \{m\}$
2. `for` $i \leftarrow 1$ `to` D_f `do`
   a. `EncryptedMessages` $\leftarrow \{(p, A_p[v], v) : v \in$ `ReceivedMessages`$\}$
   b. send `EncryptedMessages` to each neighbor
   c. `foreach` neighbor q `do`
      `foreach` $(p_1, a_1, (p_2, a_2, \ldots (p_k, a_k, v) \ldots))$ received from q `do`
      \\checking if the message is genuine
      `if` $a_k = A_{p_k}[v]$ `and` $a_i = A_{p_i}[(p_{i+1}, a_{i+1}, \ldots (p_k, a_k, v) \ldots))] : 1 \leq i \leq k-1$
      `and` $p_k \notin$ `Nodes` `then`
         A. add $(p_1, a_1, (p_2, a_2, \ldots (p_k, a_k, v) \ldots)))$ to `ReceivedMessages`
         B. add p_k to `Nodes`
         C. $M_{p_k} \leftarrow v$
3. `return` (`Nodes`, $M_1, \ldots, M_{|\texttt{Nodes}|}$)

and authenticating it. If a node p adheres to this scheme of communication, then it stores in each round at most one message from a non-faulty node and at most two messages from a faulty node. This gives a polynomial bits complexity for this stage.

The global communication stage. Once nonfaulty nodes deliver their input values dependably to all other nodes, by executing the previous stage, the messages containing input values are scattered among the network nodes. To distribute the information among all nodes, the nodes perform an all-to-all communication procedure called SEND-TO-ALL. It takes a node's name and the information to be sent as parameters, and returns the set `Nodes` of the names of the other nodes together with the information the nodes from set `Nodes` wanted to propagate; in the case of Byzantine nodes, such a message may be missing. During the following D_f rounds, each node sends its entire knowledge to all the neighbors, using messages that can be verified for their authenticity. A node that obtains a message from a neighbor, verifies if the message is authentic. If this is the case, then the recipient learns new knowledge by adding it to its private repository. Such knowledge includes, for each node that has been learned about, the input value of each node along with the path that this piece of knowledge traversed from its originator node.

The local computation stage. Once the global communication stage is over, the information about input values is distributed among the non-faulty nodes. Each non faulty node authenticate all received messages. Based on all received authenticated messages, a node p discovers input values of another node q. This information is stored array $\texttt{Input}_p[q]$. It may occur that two different input values of the same node q appear even after authenticating messages. Each node that sent such ambiguous information is considered as faulty by the node p. Once this is detected, the corresponding value in the set $\texttt{Input}_p[q]$ is set to a special symbol $\perp$. The decision at node p is made on a majority from the values in the array $\texttt{Inputs}_p$ that are different from $\perp$.

▶ **Theorem 4.** *Algorithm* Fast-Authenticated *solves Consensus with authenticated messages in $f + D_f$ communication rounds while using messages with a polynomial number of authenticators and a polynomial number of auxiliary bits.*

The following corollary combines Theorem 3 from Section 4 with properties of algorithm Fast-Authenticated to establish a separation between Byzantine Agreement and Authenticated Byzantine Agreement based on time performance.

▶ **Corollary 5.** *For every $d \geq f \geq 4$ there exists a $(2f + 1)$-connected graph G on which solving Consensus in the presence of at most f Byzantine nodes requires at least d rounds, while solving Consensus with authenticated messages in the presence of at most f Byzantine nodes is possible within $f + 2$ rounds.*

6 Early Stopping for Node Crashes

We propose an algorithm that is more efficient in terms of message size than algorithm Fast-Authenticated, assuming nodes are only prone to crashes. We give a Consensus algorithm that operates in time proportional to $f + D_f$, it uses messages of size $\mathcal{O}(m \log n)$, and relies on limited initial knowledge of nodes. Each node knows only its own name and could locally distinguish ports; in particular, knowing either $f + D_f$ or n is not assumed.

The algorithm is structured into three stages: discovery, testing and deciding. An execution starts with discovering the neighbors in one round. It is followed by the stage of testing, which is the main part of the algorithm, and completed by the stage of deciding. The algorithm is called Early-Stopping-Crashes, its pseudocode is given in Algorithm 5.

Algorithm 5 A pseudocode of algorithms Early-Stopping-Crashes for a node p . The pseudocode of procedure Send-And-Receive is in Algorithm 6. The main while loop implements testing and deciding. The variable `Grasp` represents the state of a node. The variable `Inputs` is the set of input values known to the node, and max(`Inputs`) is the maximum value in this set. The variable `Faulty` stores the set of crashed nodes known to p.

`algorithm` Early-Stopping-Crashes

---

1. `tentative` $\leftarrow$ `null`, `Inputs` $\leftarrow$ $\{$`input`$_p\}$, `Faulty` $\leftarrow \emptyset$, `Grasp` $\leftarrow \emptyset$, $i \leftarrow 1$
2. `for` each port `do`
  send message with `name`$_p$ to each neighbor
  `if` `name`$_q$ received through this port `then` assign `name`$_q$ to this port
3. `while tentative` $=$ `null do`
  a. $j \leftarrow 1$, `checkpoint` $\leftarrow i$, `previousInputs` $\leftarrow$ `Inputs`, `previousFaulty` $\leftarrow$ `Faulty`
  b. `while` (`tentative` $=$ `null`) and ($j < 2($`checkpoint` $+ 1))$
  and ($|$`Faulty` $\setminus$ `PreviousFaulty`$|$ $<$ `checkpoint`)
  and (for every `input`$_q \in$ `Input` $\setminus$ `Faulty` : $(q, ?, ?, j) \in$ `Grasp`) do
   set (i, `Inputs`, `Faulty`, `Grasp`, `tentative`) to the output returned by
    Send-And-Receive(v, i, `Inputs`, `Faulty`, `Grasp`, `tentative`)
   $j \leftarrow j + 1$
  c. if $j = 2($`checkpoint` $+ 1)$ and (`tentative` $=$ `null`) then `tentative` $\leftarrow$ max(`Inputs`)
4. send `tentative` to each neighbor as the decision
5. `decide` on `tentative`

In the beginning, each node initiates its variables by instruction (1.) in the pseudocode. The variable `Inputs` stores the set of the known input values represented as pairs of a node's name and the input value of this node. Initially it contains the input value of the node. If a

node knows an input value of some other node, then this is a correct value, because nodes are prone to crashes only. The variable `Faulty` stores nodes that are known to have crashed. These are nodes from whom some of their neighbors failed to receive a message in some round, and this information has been forwarded to other nodes in the network. A set `Grasp` is a digest of current knowledge. It is exchanged among neighbors until it stabilizes. More precisely, the set `Grasp` at node p consists of tuples $(q, \texttt{Inputs}, \texttt{Faulty}, i)$, each interpreted such that p has learned the set of input values `Inputs` and crashed nodes `Faulty` as known by q at the end of round i. Here q is a node's name, which could be also p, while `Inputs` and `Faulty` contain the content of these variables taken at the end of round i.

The stage of discovery is implemented by instruction (2.) of the pseudocode given in Algorithm 5. It consists of sending a node's name to all neighbors and collecting their names in return to assign neighbors' names to ports. Testing occurs in instruction (3.) and is structured as a loop. The stage of deciding begins in instruction (3c.) and continues through the last two lines of the pseudocode in Algorithm 5.

Algorithm 6 A pseudocode of procedure Send-And-Receive for a node p. It implements communicating the essential components of its state in a round to all the neighbors and updating the state by collecting similar information from the neighbors. Letter i denotes the current round number.

`procedure` Send-And-Receive $(p, i, \texttt{Inputs}, \texttt{Faulty}, \texttt{Grasp}, \texttt{decision})$

---

1. $i \leftarrow i + 1$; add a tuple $(p, \texttt{Inputs}, \texttt{Faulty}, i - 1)$ to set `Grasp`
2. send message with `Grasp` to each neighbor of p
3. for each neighbor q of p do
  a. if p received message $\texttt{Grasp}_q$ from q then
    for each tuple $(r, \texttt{Inputs}_r, \texttt{Faulty}_r, i_r)$ from $\texttt{Grasp}_q$ do
      $\texttt{Inputs} \leftarrow \texttt{Inputs} \cup \texttt{Inputs}_r$
      $\texttt{Faulty} \leftarrow \texttt{Faulty} \cup \texttt{Faulty}_r$
    add each tuple in $\texttt{Grasp}_q$ to `Grasp`
  b. if p received message with $\texttt{decision}_q$ from q then
    $\texttt{decision} \leftarrow \texttt{decision}_q$
  c. if p did not received anything from q then
    add node q to set `Faulty`
4. return (i, `Inputs`, `Faulty`, `Grasp`, `decision`)

The conditions controlling the testing loop (3.) allow for the next iteration if we have not heard from a neighbor about its decision yet and if the set `Inputs` has not just been updated and if the current estimate of the number of crashes is less than `checkpoint`. When a new testing phase begins at a round `checkpoint`, a node starts monitoring the changes of its `Grasp` in the subsequent period of $2(\texttt{checkpoint} + 1)$ rounds spent on executing the inner loop (3b.). If it finds too many changes in `Grasp` in the course of this testing period, it aborts this testing phase and starts a new one with an updated `Grasp`. Otherwise, if a node considers `Grasp` stable enough at this point, it decides on the maximum of the known input values, broadcasts its decision value to its neighbors and halts, which is what makes the stage of deciding. Similarly, if a node receives a decision value, it decides on it, broadcasts it to its neighbors and halts.

What a node tests is if its set `Grasp` is stable enough to make a decision. It does this in subsequent testing phases. Suppose that a node starts a new testing phase in round `checkpoint`. Then, in the course of $2(\texttt{checkpoint} + 1)$ following rounds, a node observes

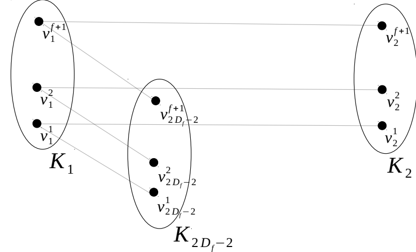


Figure 2 A visualization of graph G_{f,D_f}, for given values of f and D_f. It consists of $2(D_f - 1)$ cliques of size $f+1$ each, denoted $K_1, K_2, \ldots, K_{2(D_f-1)}$, with additional edges between corresponding nodes of subsequent cliques.

but also transmits further the set `Grasp` it has received. A node must receive consistent information about the `Grasp` of each round of every non-faulty node it learned so far, up to this round – not present in the current set `Faulty` but present in some pair in `Inputs`. If any of the received sets `Grasp` provides information about a new node in the network or increases the number of known crashes learned by the node during this testing phase beyond t, this is interpreted that `Grasp` is not stable enough. This results in aborting this testing phase and a new one starts with respect to the current `Grasp`. If such an event does not occur, a node proceeds to decide at the end of round $2(\texttt{checkpoint} + 1)$ in the phase. The rules to update `Grasp` at a node p at the end of round i are as follows. Initially, a tuple $(p, \texttt{Inputs}, \texttt{Faulty}, i-1)$ is created. Corresponding to the digest of the state of node p at the end of round $i - 1$. The current sets `Inputs` and `Faulty` are updated by incorporating the contents of the sets `Inputs` and `Faulty` in the sets `Grasp` relayed by the received messages. Each neighbor from which a message has not been received at a round is deemed crashed and added to the set `Faulty`.

▶ **Theorem 6.** *Algorithm* Early-Stopping-Crashes *solves Consensus in* $\mathcal{O}(f+D_f)$ *rounds.*

7 A Lower Bound for Node Crashes

We present a lower bound that applies to node crashes. This bound matches the performance of algorithm `Early-Stopping-Crashes` given in Section 6 and algorithm `Fast-Authenticated` given in Section 5. For any values of $f \geq 3$ and $D_f \geq 4$, we define a corresponding $(f+1)$-connected graph G_{f,D_f}, which consist of $2(D_f - 1)$ cliques of size $f + 1$ each, in which additionally the ith node in clique j is connected to the respective ith nodes in cliques $j-1$ and $j+1$ taken modulo $2(D_f - 1)$, for any $1 \leq i \leq f+1$ and $1 \leq j \leq 2(D_f - 1)$; see Figure 2 for an illustration. The value of D_f for graph G_{f,D_f} is exactly D_f. Additionally, for each node p, if some other f nodes get removed, there is still a node of distance at least $D_f - 1$ from p in the remaining graph.

▶ **Theorem 7.** *For every deterministic distributed algorithm solving Consensus there is an execution in which this algorithm terminates after least* $f + D_f - 2$ *rounds.*

Proof. Suppose that input values are binary: 0 and 1, to simplify the exposition. For every graph G and every deterministic algorithm $\mathcal{A}$ there exists a bivalent initial configuration, see [3]. There is an execution $\mathcal{E}$ of $f - 1$ rounds that ends in a bivalent configuration, see [3]. If there is an extension of $\mathcal{E}$ to some execution $\mathcal{E}'$ of f rounds, ending in a bivalent configuration, then take $\mathcal{E}'$ and continue similarly through rounds $f + 1, f + 2, \ldots,$ until

reaching an execution $\mathcal{E}''$ of $r \geq f-1$ rounds such that each of its possible extensions leads to a univalent configuration. Such $\mathcal{E}''$ exists because otherwise agreement would not be achieved for some unbounded extension of $\mathcal{E}$, contradicting the fact that $\mathcal{A}$ solves Consensus. Let β be an extension of $\mathcal{E}''$ by one round such that there is no crash in round $r+1$. Since β is univalent, it determines a decision value v_β. By the choice of $\mathcal{E}''$, there is another extension of it to some execution γ of $r+1$ rounds resulting in a different decision $1-v_\beta$. Since $\gamma \neq \beta$, there must be a crash in round $r+1$ of γ; denote a crashed node by p. There is at least one node q of distance at least $D_f - 1$ from p in a subgraph of G_{f,D_f} induced by the nodes that are non-faulty at round $r+1$. Let β_1, respectively γ_1, be an extension of β, respectively γ, to the following $D_f - 3$ rounds, with no crashes occurring. The relation $\beta_1 \overset{q}{\sim} \gamma_1$ holds. Indeed, the only difference between these two executions occurs at round $r+1$ and takes place in the non-faulty neighbors of node p in graph G_{f,D_f}. Since p and q are of distance $D_f - 1$, the neighbors of p are of distance at least $D_f - 2$ from q. The information about the only difference between these two executions could be recorded in a state of q at round $r+1+D_f-2$. Both executions take $r+1+D_f-3$ rounds, therefore $\beta_1 \overset{p}{\sim} \gamma_1$. Configurations β_1 and γ_1 are univalent and result in different decisions, therefore node q cannot decide by round $r+1+D_f-3 \geq f+D_f-3$. So at least $f+D_f-2$ rounds are needed for algorithm $\mathcal{A}$ to terminate. ◀

References

1. Marcos Kawazoe Aguilera and Sam Toueg. A simple bivalency proof that t-resilient consensus requires $t+1$ rounds. *Information Processing Letters*, 71(3-4):155–158, 1999.
2. Hagit Attiya and Faith Ellen. *Impossibility Results for Distributed Computing*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2014.
3. Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. Wiley, Second edition, 2004.
4. Piyush Bansal, Prasant Gopal, Anuj Gupta, Kannan Srinathan, and Pranav K. Vasishta. Byzantine agreement using partial authentication. In *Proceedings of the 25th International Symposium on Distributed Computing (DISC)*, volume 6950 of *Lecture Notes in Computer Science*, pages 389–403. Springer, 2011.
5. Amotz Bar-Noy, Danny Dolev, Cynthia Dwork, and H. Raymond Strong. Shifting gears: Changing algorithms on the fly to expedite Byzantine agreement. *Information and Computation*, 97(2):205–233, 1992.
6. Piotr Berman and Juan A. Garay. Cloture votes: $n/4$-resilient distributed consensus in $t+1$ rounds. *Mathematical Systems Theory*, 26(1):3–19, 1993.
7. Piotr Berman, Juan A. Garay, and Kenneth J. Perry. Optimal early stopping in distributed consensus (extended abstract). In *Proceedings of the 6th International Workshop on Distributed Algorithms (WDAG)*, volume 647 of *Lecture Notes in Computer Science*, pages 221–237. Springer, 1992.
8. Armando Castañeda, Yoram Moses, Michel Raynal, and Matthieu Roy. Early decision and stopping in synchronous consensus: A predicate-based guided tour. In *Proceedings of 5th International Conference on Networked Systems (NETYS)*, volume 10299 of *Lecture Notes in Computer Science*, pages 206–221. Springer, 2017.
9. Bogdan S. Chlebus and Dariusz R. Kowalski. Robust gossiping with an application to consensus. *Journal of Computer System and Sciences*, 72(8):1262–1281, 2006.
10. Bogdan S. Chlebus and Dariusz R. Kowalski. Time and communication efficient consensus for crash failures. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC)*, volume 4167 of *Lecture Notes in Computer Science*, pages 314–328. Springer, 2006.

11 Bogdan S. Chlebus and Dariusz R. Kowalski. Locally scalable randomized consensus for synchronous crash failures. In *Proceedings of the* 21*st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 290–299. ACM, 2009.

12 Bogdan S. Chlebus, Dariusz R. Kowalski, and Michał Strojnowski. Fast scalable deterministic consensus for crash failures. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 111–120. ACM, 2009.

13 Bogdan S. Chlebus, Dariusz R. Kowalski, and Michał Strojnowski. Scalable quantum consensus for crash failures. In *Proceedings of the* 24*th International Symposium on Distributed Computing (DISC)*, volume 6343 of *Lecture Notes in Computer Science*, pages 236–250. Springer, 2010.

14 Danny Dolev. The Byzantine generals strike again. *Journal of Algorithms*, 3(1):14–30, 1982.

15 Danny Dolev and Christoph Lenzen. Early-deciding consensus is expensive. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 270–279. ACM, 2013.

16 Danny Dolev and Rüdiger Reischuk. Bounds on information exchange for Byzantine agreement. *Journal of the ACM*, 32(1):191–204, 1985.

17 Danny Dolev, Rüdiger Reischuk, and H. Raymond Strong. Early stopping in Byzantine agreement. *Journal of the ACM*, 37(4):720–741, 1990.

18 Danny Dolev and H. Raymond Strong. Authenticated algorithms for Byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983.

19 Michael J. Fischer and Nancy A. Lynch. A lower bound for the time to assure interactive consistency. *Information Processing Letters*, 14(4):183–186, 1982.

20 Michael J. Fischer, Nancy A. Lynch, and Michael Merritt. Easy impossibility proofs for distributed consensus problems. *Distributed Computing*, 1(1):26–39, 1986.

21 Zvi Galil, Alain J. Mayer, and Moti Yung. Resolving message complexity of Byzantine agreement and beyond. In *Proceedings of the* 36*th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 724–733. IEEE, 1995.

22 Juan A. Garay and Yoram Moses. Fully polynomial Byzantine agreement for $n > 3t$ processors in $t + 1$ rounds. *SIAM Journal on Computing*, 27(1):247–290, 1998.

23 Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2012.

24 Idit Keidar and Sergio Rajsbaum. A simple proof of the uniform consensus synchronous lower bound. *Information Processing Letters*, 85(1):47–52, 2003.

25 Muhammad Samir Khan, Syed Shalan Naqvi, and Nitin H. Vaidya. Exact Byzantine consensus on undirected graphs under local broadcast model. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 327–336. ACM, 2019.

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

27 Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

28 Philippe Raipin Parvédy and Michel Raynal. Optimal early stopping uniform consensus in synchronous systems with process omission failures. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 302–310. ACM, 2004.

29 Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980.

30 Michel Raynal. *Fault-tolerant Agreement in Synchronous Message-passing Systems*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2010.

31 T. K. Srikanth and Sam Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Computing*, 2(2):80–94, 1987.

32 Lewis Tseng. Recent results on fault-tolerant consensus in message-passing networks. In *Proceedings of the* 23*rd International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, volume 9988 of *Lecture Notes in Computer Science*, pages 92–108. Springer, 2016.

33 Lewis Tseng and Nitin H. Vaidya. Fault-tolerant consensus in directed graphs. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 451–460. ACM, 2015.

34 Lewis Tseng and Nitin H. Vaidya. A note on fault-tolerant consensus in directed networks. *SIGACT News*, 47(3):70–91, 2016.

35 Xianbing Wang, Yong Meng Teo, and Jiannong Cao. A bivalency proof of the lower bound for uniform consensus. *Information Processing Letters*, 96(5):167–174, 2005.

Scalable and Secure Computation Among Strangers: Message-Competitive Byzantine Protocols

John Augustine
Dept. of Computer Science & Engg, Indian Institute of Technology Madras, Chennai 600036, India
augustine@iitm.ac.in

Valerie King
Department of Computer Science, University of Victoria, Vancouver BC V8P 5C2, Canada
val@uvic.edu

Anisur Rahaman Molla
Computer and Communication Sciences, Indian Statistical Institute, Kolkata 700108, India
molla@isical.ac.in

Gopal Pandurangan
Department of Computer Science, University of Houston, Houston, TX 77204, USA
gopalpandurangan@gmail.com

Jared Saia
Department of Computer Science, University of New Mexico, NM 87131, USA
saia@cs.unm.edu

Abstract

The last decade has seen substantial progress on designing Byzantine agreement algorithms which do not require all-to-all communication. However, these protocols do require each node to play a particular role determined by its ID. Motivated by the rise of permissionless systems such as Bitcoin, where nodes can join and leave at will, we extend this research to a more practical model where initially, each node does not know the identity of its neighbors. In particular, a node can send to new destinations only by sending to random (or arbitrary) nodes, or responding to messages received from those destinations. We assume a synchronous and fully-connected network, with a full-information, but static Byzantine adversary. A major drawback of existing Byzantine protocols in this setting is that they have at least $\Omega(n^2)$ message complexity, where n is the total number of nodes. In particular, the communication cost incurred by the honest nodes is $\Omega(n^2)$, even when Byzantine node send no messages. In this paper, we design protocols for fundamental problems which are *message-competitive*, i.e., the total number of bits sent by honest nodes is not significantly more than the total sent by Byzantine nodes.

We describe a message-competitive algorithm to solve Byzantine agreement, leader election, and committee election. Our algorithm sends an expected $O((T+n)\log n)$ bits and has latency $O(\text{polylog}(n))$ (even in the CONGEST model), where $T = O(n^2)$ is the number of bits sent by Byzantine nodes.[1] The algorithm is resilient to $(\frac{1}{4}-\epsilon)n$ Byzantine nodes for any fixed $\epsilon > 0$, and succeeds with high probability.[2] Our message bounds are essentially optimal up to polylagarithmic factors, for algorithms that run in polylogarithmic rounds in the CONGEST model.

We also show lower bounds for message-competitive Byzantine agreement regardless of rounds. We prove that, in general, one cannot hope to design Byzantine protocols that have communication cost that is significantly smaller than the cost of the Byzantine adversary.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Mathematics of computing → Probabilistic algorithms; Mathematics of computing → Discrete mathematics

[1] Our algorithm will never send more $O(n^2 \log n)$ bits if $T \geq n^2$.
[2] Throughout, "with high probability" means with probability at least $1 - n^{-c}$ for some constant $c > 0$.

© John Augustine, Valerie King, Anisur Rahaman Molla, Gopal Pandurangan, and Jared Saia; licensed under Creative Commons License CC-BY

Keywords and phrases Byzantine protocols, Byzantine agreement, Leader election, Committee election, Message-competitive protocol, Randomized protocol

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.31

Related Version A full version of the paper is available at https://arxiv.org/abs/1907.10308.

Funding *John Augustine*: Research supported in part by an Extra-Mural Research Grant (file number EMR/2016/003016) and a MATRICS grant (file number MTR/2018/001198), both funded by the Science and Engineering Research Board, Department of Science and Technology, Government of India and by the VAJRA faculty program of the Government of India.
Valerie King: This work is supported by an NSERC Discovery Grant.
Anisur Rahaman Molla: Research supported by DST Inspire Faculty research grant DST/INSPIRE/04/2015/002801 and ISI DCSW/TAC Project (file number E5412).
Gopal Pandurangan: Research supported, in part, by NSF grants CCF-1527867, CCF-1540512, IIS-1633720, CCF-BSF-1717075, BSF award 2016419, and by the VAJRA faculty program of the Government of India.
Jared Saia: This work is supported by the National Science Foundation grants CNS-1318880 and CCF-1320994.

1 Introduction

What happens when you don't know your neighbors? Permissionless systems, such as cryptocurrency [12, 23], anonymous communication [27, 63], and wireless [44, 48, 64, 61], allow nodes to join and leave with little or no admission control. In such systems, nodes are generally known only by self-generated identifiers[3]; and communication primitives may be limited to: sending a message to all nodes, sending a message to a random (or arbitrary) node, and responding to a message sent directly. Unfortunately, all algorithms to coordinate such networks in the presence of malicious faults seem either to require all-to-all communication, or make cryptographic assumptions.

A major challenge in permissionless systems is dealing with malicious, or *Byzantine* nodes, which can try to foil the protocols executed by honest, or *good* nodes. Byzantine-resistant protocols are at the heart of secure systems. Consider the example of Bitcoin – a decentralized, digital currency [12]. A crucial problem faced by Bitcoin is fault-tolerant agreement on a set of ordered transactions.

The problem of achieving agreement under Byzantine faults, *Byzantine agreement*, is a fundamental and long-studied problem in distributed computing [56, 7, 50]. In this problem, all good nodes start with an input bit, and we must ensure: (1) *All* good nodes output the same input bit (*consensus condition*); and (2) this common bit is the input bit of some good node (*validity condition*). This must be done despite the presence of a constant fraction of Byzantine nodes that can deviate arbitrarily from the protocol executed by the good nodes. Byzantine agreement is a "keystone" problem in distributed computing, in that it provides a critical building block for creating attack-resistant distributed systems. It has been used in many domains including: sensor networks [60], grid computing [6], peer-to-peer networks [59] and cloud computing [65]. However, despite intensive research, there is still no practical solution to Byzantine agreement for large networks. A main reason for this is the *large message complexity* of currently known protocols, as has been suggested by

[3] Such as the public key for a digital signature.

many systems papers [3, 5, 17, 51, 66]. The best known Byzantine protocols have (at least) quadratic message complexity, i.e., $\Theta(n^2)$, where n is the number of nodes in the network (e.g., [32, 9, 26, 46]). This is especially true of protocols that run fast, i.e., in $O(\text{polylog}\, n)$ rounds (e.g.,[32, 9]). As noted in many papers [1, 38, 34] the message complexity plays an important role in performance.

King and Saia [40] described the first Byzantine agreement algorithm for *synchronous, complete networks* that *breaks* the quadratic message barrier *under the assumption that nodes a priori know the identities of all their neighbors.* This assumption is called the KT_1 model [57]. A more challenging model is KT_0 (also called the *clean network* model [57]), where nodes do not know the identity of their neighbors a priori, but do learn a node's identity upon receiving a message from it. In the KT_1 model, [40] presented an algorithm where *each* good node sends only $\tilde{O}(\sqrt{n})$ messages, and thus total message complexity is $\tilde{O}(n^{1.5})$. Braud-Santoni et al. [14] improved this to $O(n \text{ polylog}(n))$ total message complexity, however, their protocol might require some node to send $O(n)$ messages.

The KT_0 model is more applicable to permissionless networks, where nodes enter and leave at will, and hence it is not reasonable to assume that nodes a priori know the identities of all other nodes in the system. We can convert algorithms for KT_1 to KT_0 by including an initial step where each node communicates with all its neighbors to obtain their identities, but this incurs a $\Theta(n^2)$ message cost. It is better to avoid such costly, potentially all-to-all, communication cost. Hence a fundamental question is:

Can we design Byzantine protocols that require sub-quadratic messages in KT_0?

In this paper, we address the above question. Our focus is on the fundamental problems of Byzantine agreement, leader election and committee election. Our main result (Theorem 1) is an algorithm to solve these problems while sending a number of bits that is $O((T + n) \log n)$, where T is *the number of bits sent by Byzantine nodes*, and n is the network size.[4] We show (Theorem 2) that this is essentially the best possible bound if one desires fast (i.e., polylogarithmic rounds) algorithms.

To the best of our knowledge, our result introduces *message-competitive analysis* to the study of Byzantine protocols. In particular, our algorithm is *message-competitive* in the sense that the number of messages sent by good nodes competes well with the number sent by Byzantine nodes; if Byzantine nodes send fewer messages then our algorithm also sends fewer. An alternate way to interpret our result is that Byzantine nodes have to incur significant message complexity (up to quadratic in n) in order to make the good nodes to have large message complexity. This kind of result where algorithmic cost is measured with respect to adversarial cost, is an example of *resource-competitive analysis* [11, 30].

Our work can be considered as an improvement in a long line of work that focuses on designing message-optimal Byzantine protocols, see e.g., [38] and the references therein and the recent work of [1]. We note that prior work on Byzantine protocols all incurred at least quadratic message complexity (in KT_0), regardless of the behavior of the Byzantine nodes. One exception is the result of Hadzilacos and Halpern [38] (see also [22]) that gives a message bound of $O(nt)$ (deterministically) when there are t Byzantine nodes; however this protocol takes $O(n)$ rounds. This protocol's message complexity is proportional to the number of Byzantine nodes, but not to the total number of messages sent by them. Our protocol is more general and (essentially) subsumes[5] the protocol of [38], while being significantly faster. It is significantly more message-efficient when t is large and T is small.

[4] Our algorithm sends at most $O(n^2 \log n)$ bits if $T \geq n^2$.

[5] Note that if there are t Byzantine nodes our protocol has a message complexity of $O(nt \log n)$ with high probability, which is a logarithmic factor larger than the $O(nt)$ bound of [38].

The lower bound results of Hadzilacos and Halpern [38] imply that our protocol is the best possible if one wants fast algorithms (i.e., finishing in polylogarithmic rounds) in the CONGEST model, where good nodes send only small-sized messages, i.e., $O(\log n)$ bits per edge per round. Our protocol is also lightweight and fast (has low latency) and can be used as a building block for designing secure and scalable (where communication and latency scales efficiently with network size) systems.

1.1 Model

We consider a network of n nodes. There are $t \leq (\frac{1}{4} - \epsilon_0)n$ *bad* nodes which are controlled by the adversary, for fixed $\epsilon_0 > 0$. The remaining nodes are *good* and follow our algorithm.

We consider a synchronous, fully-connected network in the KT_0 model [55, 57]. In particular, each node has ports to every other node in the network, but learns the identity of each node reachable through a port only by receiving a communication from that node. Thus a node sends to a new destination only by selecting a port, or by responding to messages received. We note that our algorithm technically only requires two primitives to send to unknown nodes: the ability to write to (1) a random unknown ID; and (2) all unknown IDs. Thus, it may be useful for models beyond KT_0, such as a gossiping-based communication model [47].

The n nodes are assumed to have distinct ID's which lie in $[1, n^k]$ for k is a (large) constant.[6] Our adversary is full-information in that it knows the states of all nodes at any time, is assumed to be computationally unbounded, and is also *rushing* in the sense that it can read messages sent by good nodes before sending out its own messages. However, the adversary is *static*, so that it must decide which nodes are bad prior to the start of the algorithm. A Byzantine node can choose its identity initially, but once chosen, that identity is presented to all nodes which receive messages from the Byzantine node. We expect this last requirement to useful in conjunction with algorithms in [35, 36, 37] that require some effort, such as solving a computational puzzle, in order to create a new identity.

We seek to design algorithms with low latency, i.e., the number of rounds until termination, and low message complexity, i.e., the total number of messages sent by good nodes.

1.2 Our Contributions

We solve three classic problems in this model. In *Byzantine agreement*, all good nodes must output the same bit, which is the input bit of some good node. In *leader election*, all good nodes must agree on a leader, and this leader must be good with constant probability. In *committee election*, all nodes must agree on a subset of $O(\log n)$ nodes where the fraction of bad nodes in the subset is within a small ϵ fraction of the overall fraction of bad nodes.

Our main result is as follows.

▶ **Theorem 1.** *There exists a randomized algorithm that solves Byzantine agreement, leader election and committee election in the above model. This algorithm sends an expected $O((T + n) \log n)$ messages, and has latency $O(\text{polylog}(n))$, where T is the minimum of n^2 and the total number of bits sent by the bad nodes to good nodes. (If $T \geq n^2$, then the algorithm sends $O(n^2 \log n)$ bits.) It is resilient to $t \leq (\frac{1}{4} - \epsilon_0)n$ Byzantine faults for any fixed $\epsilon_0 > 0$, and succeeds with probability $1 - 1/n^c$ for any constant c.*

[6] This means that an ID can be represented using $O(\log n)$ bits, which can be sent in a message.

We note that our $O(\text{polylog}(n))$ latency bound holds even in the CONGEST model, where each message is $O(\log n)$ bits. The algorithm *KT0-ByzantineAgreement* described in Section 3 achieves the result in Theorem 1, and the proof of this theorem is in Section 4.

The interesting regime for T is subquadratic, where our algorithm sends only subquadratic messages (actually proportional to T), unlike prior works (cf. Section 1) that incurred quadratic message complexity in general. Our algorithm is resilient up to a *constant* fraction of nodes – up to essentially $n/4$ – being bad. It is a open question whether one can improve this tolerance further up to $n/3$ bad nodes, which is the best possible[56].

As mentioned in Section 1, the work of Hadzilacos and Halpern [38] shows a tight bound of $\Theta(nt)$ for message complexity[7] of Byzantine agreement with $t \leq n$ Byzantine nodes (t is unknown). It gives a deterministic algorithm and a lower bound proof that holds for *Monte Carlo randomized algorithms that succeed with high probability* as well. If one desires fast, i.e., polylog(n) rounds algorithms (as is the case with our randomized algorithm), then the above $\Theta(nt)$ bound shows that our message complexity is essentially the best possible in general. This is because, since in each round at most $\tilde{O}(n)$ bits can be sent by a Byzantine node to good nodes in the CONGEST model[8], and since the number of rounds is bounded by $O(\text{polylog}\, n)$, the number of Byzantine nodes t is $\tilde{\Omega}(T/n)$, where T is the total number of bits sent by Byzantine nodes to good nodes.[9] By the $\Omega(nt)$ lower bound of [38], we have the following lower bound theorem.

▶ **Theorem 2** (follows from [38]). *Let $T \in [\Theta(n), \Theta(n^2)]$ be the total number of bits sent by the Byzantine nodes to good nodes. Then any Byzantine agreement algorithm (including randomized Monte Carlo algorithms that succeed with high probability) that finishes in* polylog(n) *rounds in the CONGEST model needs, in general, at least $\tilde{\Omega}(T)$ messages in expectation.*

The above theorem implies that our randomized algorithm with message complexity of $O((T + n) \log n)$ and $O(\text{polylog}\, n)$ latency is essentially optimal in CONGEST.

To the best of our knowledge, our protocol is the first that is message-competitive. As discussed above, all prior protocols (excepting [38] and [22]) require at least quadratic messages, regardless of the behavior of Byzantine nodes. This is especially true for protocols that take small number of rounds (e.g., [9, 32]). Algorithms sending $O(nt)$ messages [22, 38] have linear latency. One can view these upper bounds that depend on t as a special case of our result. Our message-competitive bound is more general, in the sense, it is proportional to the total number of messages sent by Byzantine nodes. The case when a large number of Byzantine nodes send a small number of messages is not captured in the prior bounds.

We also show lower bounds for message-competitive Byzantine agreement that hold regardless of rounds (see Section 6). We prove that, in general, one cannot hope to design Byzantine protocols that have communication cost that is significantly smaller than the communication cost of the Byzantine adversary. We first show a lower bound for *deterministic* BA protocols which is essentially tight with respect to our randomized algorithm (see Section 6.1). We show that if $T = O(n^2)$ is the budget on the message bits of the Byzantine nodes, then for any deterministic protocol, the total number of messages sent by the good nodes is $\Omega(T)$ (see Theorem 12). The deterministic lower bound holds even in the KT_1 model. We

[7] This bound holds even for messages of size one bit.

[8] We only consider algorithms where Byzantine nodes follow the CONGEST bound. Otherwise, if a Byzantine node sends $\omega(\log n)$ bits to a good node, it will be ignored.

[9] The $\tilde{O}$ notation hides a polylog(n) factor and $\tilde{\Omega}$ notation hides a $1/\text{polylog}(n)$ factor.

then show a somewhat weaker lower bound on the message competitiveness of randomized Las Vegas (that always succeed) BA protocols (see Section 6.2) where we assume Byzantine nodes can fake their IDs. The argument for the randomized case is more involved compared to the deterministic case, as the algorithm's (future) random choices are unknown to the Byzantine adversary. We show that if $T = n^{1+\alpha}$ for some $\alpha \in (0, 1]$ is the budget of the Byzantine nodes, then for any (randomized) BA algorithm in the KT_0 setting, the expected number of messages sent by good nodes, is at least $\Omega(n^{1+\frac{\alpha}{2}})$ (see Theorem 14).

All omitted proofs and additional details will be given in the full paper [8].

1.3 Related Work

Message-Competitive Analysis. This paper introduces *message-competitive analysis* which can be considered as a special case of the more general *resource-competitive* analysis [11, 30] to the study of Byzantine agreement. In resource-competitive analysis, the computational cost of the attacker, T, is incorporated as a parameter in performance analysis. That is, the cost of executing an algorithm over a network of n nodes is measured not only as a function of n, but also as a function of T. Messages are an important resources and as mentioned in [38], "the number of messages used by a protocol is important, possibly the most important, factor that determines its performance."

Resource competitive analysis has been applied to designing algorithms for: jamming-resistant wireless communication [29, 31, 43]; attack-resistance on multiple access channels [10], tolerating adversarial channel noise [4, 20, 21], and efficiently distributing bridges for anonymity networks such as TOR [67]. See [11, 30] for detailed surveys.

Communication Efficient Byzantine Agreement and Leader Election. Byzantine agreement enables participants in a distributed network to reach agreement on a decision, even in the presence of a malicious minority. Thus, it is a fundamental building block for many applications including: cryptocurrencies [13, 24, 28, 33]; trustworthy computing [15, 16, 17, 18, 19, 45, 62]; peer-to-peer networks [2, 54]; and databases [53, 58, 68].

In 2006, King, Saia, Sanwalani, and Vee [41] gave a (randomized) algorithm to solve Byzantine agreement, leader election and committee election problems in a model differing from the one in this paper only in the assumption of KT_1 communication. This was the first algorithm to use only $\tilde{O}(1)$ bits of communication per node, and $\tilde{O}(1)$ time to bring almost all processors to agreement. This result can also be achieved in a particular sparse network [42]. This initial work produced agreement among all but $o(n)$ nodes. Further work extended this result to achieve everywhere agreement, while using a number of bits that is $\tilde{O}(n^{3/2})$ (load-balanced) [39]; and $\tilde{O}(n)$ (not load-balanced) [14]. All of these algorithms required each node to play a particular role as determined by its unique ID in $[1, n]$, and to send to specific neighbors. In other words, these algorithms critically rely on the KT_1 model. These bounds hold even if the bad nodes send any number of bits. Establishing Byzantine agreement via the use of committees is a common approach; for examples, see [28, 41, 49]. Recent work by Abraham et al. [1] revisits the problem of communication efficient Byzantine agreement in the KT_1 model. They show that achieving sub-quadratic message cost with an adaptive adversary in this model requires that the adversary not have the ability to erase messages already sent by the nodes it adaptively takes over. More related work is given in full paper.

2 High-level Overview of Algorithms and Techniques

We focus first on Byzantine agreement, our solutions to leader and committee election use similar techniques. Our algorithm depends on solutions to two new problems: *Implicit Agreement* and *Promise Agreement*. In the *Implicit Agreement* problem, success means that strictly greater than a t/n fraction of good nodes decide on the same (correct) bit and the remaining good nodes do not decide; and failure means that no good nodes decide. Next, the *Promise Agreement* problem assumes there has first been either success or failure in *Implicit Agreement*. In the case of success, *Promise Agreement* ensures all nodes decide on the same value and terminate; in the case of failure, no nodes decide.

KT0-ByzantineAgreement runs in epochs. In each epoch, we (1) run an algorithm for *Implicit Agreement*; (2) run an algorithm for *Promise Agreement*; and (3) terminate in the case of success, or increase the number of messages sent in the case of failure.

The number of messages sent for *Implicit Agreement* is tuned by increasing the number of *active* nodes. In particular, during a run of *Implicit Agreement*, the active nodes first attempt to solve Byzantine agreement among themselves, and then to communicate the output to all other nodes in the network. Our *Implicit Agreement* algorithm ensures that, unless the bad nodes send a number of messages that is n times the number of active nodes, then *Implicit Agreement* will succeed. Next, we solve *Promise Agreement*. This ensures that if *Implicit Agreement* succeeded, then all nodes will decide on the same value and terminate; and if *Implicit Agreement* failed, then no nodes decide. In the latter case, all nodes proceed to the next epoch, where the number of active nodes doubles in expectation.

When there is partial knowledge of participants. We say that a node x has a view of node y if x knows y's ID and the port to y. With a fair amount of technical work, we show that it is possible to modify an algorithm by King et al. [41] to ensure agreement even among nodes whose views only "mostly" overlap, provided that the range of all IDs is only polynomially large. We call this modified algorithm *LargeCoreBA*, and summarize its properties in Lemma 3 below; we believe the result may be of independent interest.

▶ **Lemma 3.** *Let G be a set of good nodes which wish to come to agreement. For each $x \in G$, let S_x be the set of nodes in the view of x. Let B be the set of bad nodes in $\bigcup_{x\in G} S_x$. Assume $G \subseteq \bigcap_x S_x$; $|B| \leq (1-\epsilon)|G|/2$ for some fixed constant $\epsilon > 0$; and all nodes have distinct ID's in $[1, n^k]$. Then there is an algorithm* LargeCoreBA *which computes almost everywhere agreement (i.e., computes agreement among $(1-1/\log n)$ fraction of nodes in G) with high probability in time and communication per node which is polylogarithmic in $|G|+|B|$. In one more round, if each good node broadcasts to all other nodes, and then each node takes the majority, all nodes will come to agreement using $|G|(|G|+|B|)$ total messages, and latency polylogarithmic in $|G|+|B|$.*

Implicit Agreement. Our solution to *Implicit Agreement* is given in Steps 1 to 6 of our main algorithm in Section 3.1. There are two key technical problems that must be addressed.

First, how do we ensure that each active node x maintains a set S_x so that the conditions of Lemma 3 are matched? Also, in order to achieve a good competitive ratio, we need the conditions of Lemma 3 to hold unless the adversary sends $\Omega(nA)$ messages, where A is the number of active nodes. If each active node x naively adds to S_x all nodes y that it receives an initial message from, then the adversary can add A Byzantine nodes to each S_x while sending only A^2 messages. Thus, we must enlist the aid of non-active nodes to establish the S_x sets. Initially, each active node sends its ID to *all* nodes. Call a good node *light* if it has

received a number of IDs approximately equal to A. Then the light nodes convey information about their S_x sets to the nodes in S_x. They cannot send out *all* the IDs in S_x, since that would be too many bits. Instead, they just send out a single random ID, and a node y adds an ID to S_y if it was received from at least enough (β) nodes that claim to be light.

Unfortunately, an adversary can still cause problems by making the size of the union of the bad nodes in each S_x large, so that $|B|$ is large in Lemma 3, even when the adversary does not send out too many messages. To solve this problem, we use a "validation" step, whereby each active node, for each ID in S_x, queries $\Theta(\log n)$ random nodes about whether they have the ID in their S_x sets, and filters out the ID unless enough of these queries are answered affirmatively. Based on information obtained during this process (Step 1 through Step 3c in Section 3.1), the active nodes determine if the number of light nodes is sufficient for favorable success in this epoch.

This brings us to the second problem. How can the active nodes agree on one of two options for this epoch: (1) conditions are favorable for agreement; or (2) conditions are not favorable? We can make use of *LargeCoreBA* in coming to agreement on an option. However, this is still challenging given that, under certain conditions, some active nodes may run *LargeCoreBA*, while other active nodes may not even have a small enough S_x set to run it. To address this issue requires careful decisions about whether a node will run *LargeCoreBA*, what its input will be, and whether or not it will trust the output, all based on the node's estimate of the number of light nodes (See Step 4, Section 3.1 for details). In particular, nodes will sometimes run *LargeCoreBA*, because other nodes are relying on them to do so, even when they plan to ignore the output. If active nodes decide conditions are favorable via the first call to *LargeCoreBA* (Step 4), they will all run it again (Step 5) to decide on a bit. Lemma 7 in Section 4 shows that no matter what the number of light nodes, these two steps ensure all active nodes come to agreement on the same decision.

Finally, in Step 6, active nodes send their decision to all other nodes. Nodes that have small S_x sets take the majority of the messages received in this step, whereas other nodes default to a decision to wait for the next epoch. We can thereby guarantee the post-condition for *Implicit Agreement*: either (1) a strictly greater than t/n fraction of good nodes decide, or (2) no good nodes decide. We obtain this result even when the adversary floods some good nodes but not others.

Promise Agreement. A final technical challenge is to determine whether or not we need to run another epoch. After solving *Implicit Agreement*, either (1) strictly greater than a t/n fraction of the good nodes have decided on the same correct bit; or (2) no good nodes have decided. We must then ensure that *all* good nodes decide either to terminate or to run another epoch. To do this, we run an algorithm, *PromiseAgreement* that solves the *Promise Agreement* problem (see Section 5.2). The solution simply has each node sample a logarithmic number of other nodes, and take a majority vote. It does not increase the overall asymptotic number of messages sent, but some non-active nodes can be forced by the adversary to respond to $O(n)$ requests. If the outcome of *PromiseAgreement* is not agreement then all nodes proceed to the next epoch, where the number of active nodes doubles in expectation. In this way, we can guarantee that that *KT0-ByzantineAgreement* succeeds within $\log(n)$ expected epochs.

3 KT0-ByzantineAgreement

The overview and intuition for the steps of the main algorithm *KT0-ByzantineAgreement* are described in Section 2. Here, we give its pseudocode and define the problem *Promise Agreement*. *KT0-ByzantineAgreement* calls *LargeCoreBA* and an algorithm *PromiseAgreement* that solves *Promise Agreement*. A node x calls *LargeCoreBA* with a set of possible participants S_x, which may include nodes which do not themselves participate.

The algorithm below runs correctly with probability $1 - 1/n^c$ for any constant c, when constant C below is chosen to be sufficiently large, depending on c. We let ϵ be a small constant such that $0 < \epsilon < \epsilon_0^2$. We set $max_a = (1+\epsilon)p(n-t)$ and $min_a = (1-\epsilon)p(n-t)$ so that w.h.p. the number of *active* nodes lies in this range.

We call a good node *active* if it sets its state to active in Step 2. We call a good node *light* if the number of IDs received by it from alleged *active* nodes in Step 2 is less than $max_a + \epsilon pn$. We use bounds $Low = n - 2t - \epsilon n$ and $High = Low + t$ to describe the number of light and purported light nodes. For $p > 1/(C \log n)$, if there are at least $Low - t$ light nodes and each sends a random ID from their list of nodes that reported being active in Step 2, then w.h.p., at least $\beta = \frac{(1-\epsilon)(Low-t)}{max_a+\epsilon pn}$ copies of all their common IDs, in particular, the IDs of all *active* nodes, will be received by every *active* node. Finally, an element in an *active* node x's set S_x is validated when x queries a random set of $C \log n$ nodes and $\delta C \log n$ nodes respond yes. $\delta = \frac{(1-\epsilon)(Low-t)}{n}$ is chosen so that w.h.p., every ID in *active* will be validated but not many ID's of nodes which are bad.

3.1 Pseudocode for KT0-ByzantineAgreement

1. **Initialize:** Every node x sets $p \leftarrow (C \log n)/n$. Each node x sets $ready\text{-}out_x \leftarrow 0$, $ready\text{-}in_x \leftarrow 0$, and sets its state to $\neg active$ and $\neg light$.
2. **Nodes become *active* and notify others:** With probability p, x sets its state to *active* and sends its ID to all nodes. Every node x sets S_x to the set of IDs received. A node sets its state to *light* if $|S_x| \leq max_a + \epsilon pn$.
3. ***Active* nodes learn of other *active* nodes:**[10]
   a. Every *light* node x randomly selects an ID in S_x and sends it to the nodes in S_x.
   b. Every active node x sets n_x to be the number of nodes which sent to x in Step 3a. If $n_x \geq Low - t$ then x resets S_x to be the set of IDs which were received from at least β nodes in step 3a. For each ID in S_x, x sends the query $< ID? >$ to a random set of $C \log n$ nodes.
   c. Every light node x answers a query $< ID? >$ if ID is in S_x and the query is sent by a node in S_x. An ID in S_x is considered *validated* if x received at least $\delta C \log n$ responses to the query for ID. Each *active* node x that sent queries removes from S_x all IDs which are not validated.
4. **Can we proceed?** Each *active* node x with $n_x \geq Low - t$ runs *LargeCoreBA* with the other nodes in S_x. The input bit to *LargeCoreBA*, $ready\text{-}in_x \leftarrow 1$ iff $n_x \geq High$. If $n_x \geq Low$ then $ready\text{-}out_x \leftarrow$ output of *LargeCoreBA*.
5. **Compute Byzantine Agreement** Each *active* node x with $ready\text{-}out_x = 1$ runs *LargeCoreBA* with nodes in S_x, with input bit, $value_x$, set to the node's initial input bit.
   Node x then sets $value_x$ to the output of this *LargeCoreBA*.

[10] Note: The S_x for inactive nodes x are unaffected by this step

6. **Take Majority:** Each active node, x, sends (*ready-out*$_x$, *value*$_x$) to all nodes. Then, each node x with $n_x \geq Low - t$ sets *ready-out*$_x$ to the majority *ready-out* bit received from nodes in S_x. If this bit is 1, then *value*$_x$ is set to the majority *value* bit received from nodes in S_x.
7. **Promise Agreement***:* Each node x runs *PromiseAgreement* with the tuple (*ready-out*$_x$, *value*$_x$), and resets the tuple based on the outcome.
   a. If *ready-out*$_x = 1$, then node x terminates and outputs value *value*$_x$;
   b. Else if $p < 1/(C \log n)$, then p doubles and x repeats from Step 2.
   c. Else (i.e., when $pn \geq n/(C \log n)$), every node sends to all other nodes to determine their IDs, and then the protocol resorts to running *LargeCoreBA*.

3.2 Promise Agreement

Here we define a variant of the almost-everywhere to everywhere Byzantine agreement problem, which we call *Promise Agreement*. In Section 5.2, we describe an algorithm, *PromiseAgreement*, to solve this problem.

▶ **Definition 4.** *An algorithm is said to solve the* Promise Agreement *problem if it has the following properties.*

1. *If (i) there is at least a $t/n+2\epsilon$ fraction of good nodes with tuple (ready-out, value) $= (1, v)$, for the same bit v; and (ii) all remaining good nodes have ready-out value of 0, then all nodes terminate with tuple (ready-out, value) $= (1, v)$.*
2. *If all good nodes have ready-out $= 0$, then all nodes terminate with ready-out $= 0$.*

4 Analysis of KT0-ByzantineAgreement

4.1 Correctness

We call one run of all the steps in the *KT0-ByzantineAgreement* algorithm an epoch. We assume $t \leq (\frac{1}{4} - \epsilon_0)n$ for a fixed $\epsilon_0 > 0$. We note that $p < 1/(C \log n)$ except in Step 7c.

▶ **Lemma 5.** *The following events occur w.h.p. in n.*

1. *The number of* active *nodes is between min_a and max_a.*
2. *If there are at least $Low - t$ light nodes, then all active nodes receive at least β copies of the ID of every* active *node in Step 3a.*
3. *If there are at least $Low - t$ light nodes, then all* active *nodes will consider all IDs of* active *nodes validated after Step 3c.*
4. *If an ID is contained in the S_x sets of at most $(1 - \epsilon)(\delta - t/n)n$ light nodes in Step 3b, then that ID will not be validated.*

Proof. For each of these items there is a random variable X which is the number of successful independent trials. In each case, we will show that $E[X] \geq C' \log n$ for some constant C'. Then, Chernoff bounds imply that $Pr(|X - E[X]| \geq \lambda E[X]) \leq n^{-c}$, for any fixed $\lambda < 1$, and any fixed c, for C' sufficiently large [52]. The details are deferred to the full paper [8]. ◀

For a fixed epoch, let *CORE* be the set of active nodes that run *LargeCoreBA* in Step 4. We show that the nodes participating in *LargeCoreBA* have the desired properties to successfully complete it when there are at least $Low - t$ light nodes. (See Lemma 3.)

From Lemma 5, we can observe the following.

▶ **Lemma 6.** *If there are at least $Low - t$ light nodes then w.h.p., we have the following*

1. *Every* active *node is in the CORE, and therefore $|CORE| \geq min_a$.*
2. *For all $x \in CORE$, $CORE \subseteq S_x$ after Step 3c.*
3. *Let B be the bad nodes in $\bigcup_{x \in CORE} S_x$. At the conclusion of Step 3, if there are at least $Low - t$ light nodes, $|B| \leq \epsilon' pn$ for any $\epsilon' > 0$, and $\frac{|B|}{|CORE|} \leq 1/2 - \epsilon''$ for any $\epsilon'' > 0$.*

The proof of this lemma is deferred to the full paper [8]. Lemma 6 and Lemma 3 imply that LargeCoreBA can be successfully run when there are at least $Low - t$ light nodes.

▶ **Lemma 7.** *Let L be the number of light nodes in an epoch of* KT0-ByzantineAgreement*. Then w.h.p.,*

1. *If $High \leq L$,*
   1) All active *nodes have ready-in $= 1$, they run* LargeCoreBA *and decide on ready-out $=$ 1 when run in Step 4; and*
   2) All active *nodes y run* LargeCoreBA *in Step 5 and set their value bit to the input bit $value_x$ of some* active *node x.*
2. *If $Low \leq L < High$,*
   1) All active *nodes successfully run* LargeCoreBA *but they may start with differing values for ready-out in Step 4.*
   2) If the output is a 1, all active *nodes y set ready-out $= 1$ and they will successfully run* LargeCoreBA *in Step 5 and set $value_y$ to the input $value_x$ for some* active *node x.*
   3) If the output is a 0, all active *nodes set ready-out $= 0$.*
3. *If $Low - t \leq L < Low$, all* active *nodes will successfully run* LargeCoreBA *in Step 4, though some nodes will disregard the output. All* active *nodes will start with ready-in $= 0$ and all* active *nodes will have ready-out $= 0$.*
4. *If $L < Low - t$, some* active *nodes may run a possibly flawed* LargeCoreBA *in Step 4, though all* active *nodes will disregard the output. All* active *nodes will start with ready-in $= 0$ and end with ready-out $= 0$.*

Proof. If $L < Low - t$, then $n_x < Low$ for all nodes x, thus in Step 4, all *active* nodes have *ready-in* $= 0$, disregard the output of *LargeCoreBA*, and set *ready-out* $= 0$.

If $Low > L \geq Low - t$, by Lemmas 6 and 3, *LargeCoreBA* will run successfully. All *active* nodes x have $n_x < High = Low + t$, so in Step 4, all *active* nodes x have $ready\text{-}in_x = 0$. Thus, by the consistency property of *LargeCoreBA*, all *active* nodes x have $ready\text{-}out_x = 0$.

If $Low \leq L < High$, then all active nodes running *LargeCoreBA* in Step 4 may start with different *ready-in* values, but by the correctness of *LargeCoreBA*, they will all end with the same *ready-out* value. If the *ready-out* vale is 1, in Step 5, *LargeCoreBA* will run correctly and they will all set their *value* bit to the input bit, $value_x$ of some *active* node x.

If $L \geq High$, then any *active* node x has $n_x \geq High$, and so has $ready\text{-}in_x = 1$. Thus, after Step 4, by the validity of *LargeCoreBA*, all active nodes will have *ready-out* $= 1$. Thus, they will all run *LargeCoreBA* in Step 5 and will all set their value bit to the input value bit of some *active* node. ◀

▶ **Lemma 8.** *At the end of each epoch, w.h.p., all nodes either terminate and output the same value or they all go to the next epoch.*

Proof. By Lemma 7, if any *active* node x sets *ready-out* $= 1$ after Step 5, all *active* nodes will set their tuple $(ready\text{-}out, value)$ to the value $(1, v)$, and v will be the input bit of some node in *CORE*. Moreover, there must be at least *Low* light nodes. Since every light node y has at least min_a IDs of *active* nodes in S_y, and $|S_y| \leq max_a + \epsilon n$, in Step 6, the majority of the messages received from nodes with IDs in S_y will be $(1, v)$ and y will set *ready-out* $= 1$

and $value_y = v$. Since $Low = n - 2t + \epsilon \geq t + 2\epsilon$, all good nodes will come to agreement on $(1, v)$ in Step 7, when the Promise Agreement problem is solved correctly (by Lemma 11 in Section 5.2).

On the other hand, if any *active* node x sets their value $\textit{ready-out}_x$ to 0, then we must be in Case 2, 3 or 4 of Lemma 7. In these cases, all *active* nodes have $\textit{ready-out} = 0$, at the end of Step 5. Thus, all light nodes set $\textit{ready-out} = 0$ since it is the majority value received in Step 6, and all nodes which are not light do not change their initial *ready-out* value from 0. Therefore, all nodes agree on $\textit{ready-out} = 0$. With $\textit{ready-out} = 0$, all nodes execute Steps 7b or 7c, depending on the value of p. ◀

4.2 Message Costs

▶ **Lemma 9.** *In any epoch, w.h.p., the algorithm sends $O((pn)^2 \log n + pn^2 + n \log n + T_e)$ messages, where T_e is the minimum of n^2 and the number of messages sent by bad nodes in that epoch. Moreover, in any epoch, the algorithm takes time polylogarithmic in n.*

Proof. There are $O(pn)$ *active* nodes which send to all nodes and each light node sends one message to $O(pn)$ nodes, for a total of $O(pn^2)$ messages. When S_x is reset, it is reset to be no larger than $n/\beta = O(np)$. To validate its S_x, each *active* node sends $O(\log n)$ messages for each element in S_x. There are $O(pn)$ *active* nodes, each with $|S_x| = O(n/\beta) = O(pn)$. Hence, issuing queries requires $O((pn)^2 \log n)$ messages by good nodes. There are at most T_e queries sent by bad nodes, so responding to queries requires $O(T_e + (pn)^2 \log n)$ messages.

Computing *LargeCoreBA* in Steps 4 and 5, requires $O((pn)^2)$ messages by Lemma 3. Then in Step 6, all *active* nodes send to all nodes for $O(pn^2)$ messages. Finally, in Step 7, all nodes send $O(n \log n)$ messages to solve *PromiseAgreement*, as shown in Lemma 11. Thus, the total number of messages sent in the epoch is $O((pn)^2 \log n + pn^2 + n \log n + T_e)$.

The time to perform all steps in an epoch is dominated by the cost of performing *LargeCoreBA* which is polylogarithmic. ◀

▶ **Lemma 10.** *The algorithm terminates in a decision in a given epoch, unless the adversary sends $\Omega(pn^2)$ messages.*

Proof. There are at least $High$ light nodes unless the adversary causes bad nodes to send more than $pn\epsilon$ messages to $n\epsilon$ nodes, for a total of $\Omega(pn^2)$ messages. If there are at least $High$ light nodes in an epoch, then by Lemma 7 the algorithm terminates with a decision. ◀

Note that $O((pn)^2 \log n + pn^2) = O(pn^2)$ except when $p > 1/\log n$, in which case our algorithm runs *LargeCoreBA* on all the nodes, by messaging all n of their neighbors, for a total cost of $O(n^2)$. This is the bottleneck in the algorithm which causes it to be $O(\log n)$-competitive instead of $O(1)$-competitive.

Let T be the minimum of n^2 and the total number of messages sent by the adversary, and n be the number of nodes in the network. We can now prove Theorem 1.

4.3 Proof of Theorem 1

Proof. By Lemma 10, the algorithm will terminate in an epoch, unless the adversary sends cpn^2 messages in that epoch, for some constant c. In epoch i, $p = (2^{i-1} \log n)/n$. If we do not terminate in epoch i, then $T \geq c2^{i-1} n \log n$. In epoch i, by Lemma 9, the total number of messages sent is $O((pn)^2 \log n + pn^2 + n \log n + T_e)$.

We first consider the case where it's always true that $p \leq 1/\log n$, and note that $O((pn)^2 \log n + pn^2) = O(pn^2)$. Thus, the message cost in epoch i is $O(n2^i \log n + T_i)$, where T_i is the number of messages sent by the adversary in epoch i. The T_i terms clearly sum to $O(T)$. If ℓ is the last epoch, then $O(\sum_{i=1}^{\ell} 2^i n \log n) = O(2^\ell n \log n) = O(T + n \log n)$. Thus the total number of messages sent in this case is $O(T + n \log n)$.

We next consider the case where $p > 1/\log n$. In this case, our algorithm runs *LargeCoreBA* on all the nodes, by messaging all n of their neighbors, for a total cost of $O(n^2)$. The value of T in this case is $\Omega(n^2/\log n)$, so our total message cost is $O(T \log n)$.

Since epoch i has latency polylogarithmic in n (by Lemma 3), and there are at most $\log n$ epochs, the total latency is $O(\text{polylog}(n))$. Additionally, we note that when the algorithm terminates, by Lemma 10, all good nodes come to agreement on an input bit of some node in *CORE*.

Finally we note that we can also solve the leader election and committee election problems. To do this, the active nodes use Feige's leader election algorithm [25] to elect a committee in one step, or a leader in $\log^* n$ steps among the $CORE_x$ sets for every active node x. This is done instead of selecting an agreement value as in the KSSV algorithm [41]. ◀

5 Additional Algorithms

5.1 LargeCoreBA

Here we prove Lemma 3. We do this by adapting the algorithm from [41]. In that paper, all nodes have a view of all of other nodes and nodes are numbered $[1, n]$.

The main idea of our adaptation is to show that for any s, $\log^{10} n \leq s \leq n$, there exists a deterministic assignment of IDs in $[1, n^k]$ to a set of $s/\ln n$ committees, so that for every subset of size s IDs, a $1 - 1/\ln^2 n$ fraction of committees are (1) "sufficiently large"; and (2) contain a nearly representative fraction of both good and bad nodes.

The algorithm in [41] is built upon a family of bipartite graphs with expansion-like properties. The existence of such graphs are proved using the probabilistic method (see Section 3 of [41]). We need the same properties here, but for a possibly much smaller subset of $s \leq n$ identities, which come from a much larger name space ($[1, n^k]$). We show that we can start with identities in the range $[1, n^k]$, of which s are active and generate a set of committees which have the required properties with respect to the active nodes, as is needed in each layer of the "election graph" in [41], Corollary 3.2. The details are deferred to the full paper [8].

5.2 PromiseAgreement

We now present a simple algorithm to solve the *Promise Agreement* problem, defined in Section 3.2.

PromiseAgreement

1. Each node y sends a request to a random set of $c \log n$ nodes.
2. Each node x, upon receiving a request from a node y, responds to the request by reporting ($\textit{ready-out}_x$, $\textit{value}_x$).
3. If greater than a $t/n + \epsilon$ fraction of nodes sampled by x respond with $\textit{ready-out} = 1$, then x sets $\textit{ready-out} \leftarrow 1$ and sets $\textit{value}_x$ to the majority of the *value* bits sent by sampled nodes. Else $\textit{ready-out}_x \leftarrow 0$.

▶ **Lemma 11.** PROMISEAGREEMENT *solves the* Promise Agreement *problem (Definition 4), with $O(1)$ latency, and sending $O(T' + n \log n)$ bits, where T' is the minimum of n^2 and the number of messages sent by the adversary during this algorithm.*

Proof. Assume there are at least a $t/n + 2\epsilon$ fraction of good nodes with (*ready-out*, *value*) $= (1, v)$ for the same bit v, and all remaining good nodes have *ready-out* values of 0. By Chernoff and union bounds, every good node then has greater than a $t/n + \epsilon$ fraction of good nodes with *ready-out* values of 1, and less than a $t/n + \epsilon$ fraction of bad nodes in their sample. Hence, all good nodes will terminate with tuple values of (*ready-out*, *value*) $= (1, v)$.

Assume that all good nodes have *ready-out* values of 0. Then by Chernoff and union bounds, each sample has less than a $t/n + \epsilon$ fraction of bad nodes. Hence, all good nodes will terminate with *ready-out* values of 0.

The number of bits sent is just the number of queries sent which is $O(T' + n \log n)$. ◀

6 Lower Bounds for Message-Competitive Byzantine Agreement

We now study lower bounds for message-competitive Byzantine agreement (BA). We first show a tight lower bound on the message competitiveness of deterministic BA protocols. Then we show a lower bound on the message competitiveness of randomized BA protocols.

6.1 Deterministic Lower Bound

As per our model in Section 1.1 we assume a complete n-node network with $\hat{\epsilon}n$ Byzantine nodes and $(1 - \hat{\epsilon})n$ good nodes (i.e., non-Byzantine) for some small constant $\hat{\epsilon}$. We assume the KT_0 model. The Byzantine nodes are controlled by a non-adaptive rushing adversary. It is assumed that Byzantine nodes cannot fake their own identities.

In the above setting, the goal is to show a lower bound on the message bits spent by the good nodes in any deterministic algorithm solving Byzantine everywhere agreement. The lower bound also holds in the KT_1 model, in which a node knows the ID of its neighbors.

Suppose there is a deterministic algorithm solving BA. The output of the algorithm, i.e., the agreed value depends on the ID, input distribution of the nodes and the information exchanged among the nodes during the execution of the algorithm. More precisely, the output of a node u (with id ID_u) is a function $f_u(ID_u, b_u, X_u) \to \{0, 1\}$, where the argument b_u is the input bit of u and X_u is the set of received message bits during the execution of the algorithm. Let us call this information (ID_u, b_u, X_u) as the "transcript" of u. The algorithm is deterministic and known to the adversary which controls the Byzantine nodes. Further, the algorithm should work for any input distribution (i.e., the $0 - 1$ value distribution). Given an input distribution over the nodes, the complete execution of the algorithm is known to the adversary. Based on the execution, the adversary selects Byzantine nodes (in the beginning) in such a way that the algorithm fails to achieve agreement everywhere unless it spends enough messages. In fact, we prove the following result.

▶ **Theorem 12.** *Suppose the budget of messages of the Byzantine nodes is $T \leq cn^2$ bits, for some constant c. Then any deterministic algorithm, which solves Byzantine everywhere agreement, incurs an expected $\Omega(\min\{T, n^2\})$ bits of messages.*

Proof. We give a proof-sketch here. The detailed proof can be found in the full version [8]. Let there be a deterministic algorithm $\mathcal{A}$ that solves the Byzantine agreement everywhere and incurs only $o(T)$ messages. We show a contradiction, that the agreement is wrong in the sense that there exists two nodes with two different output values for some input distribution.

Consider an arbitrary input distribution $\mathcal{I}$ over n nodes. Since the total messages sent by the good nodes is $o(T)$, there must exist a good node, say, u that exchanges (sends and receives) less than $\delta T/((1-\hat{\epsilon})n)$ message bits in total for some small constant $\delta < 1$ (the actual value of δ to be fixed later); otherwise the sum of the messages of all the good nodes would be $\Omega(T)$.Let S_u be the set of nodes which exchange messages with u throughout the execution of $\mathcal{A}$ on the given input $\mathcal{I}$. Note that, given $\mathcal{A}$ and $\mathcal{I}$, u and S_u are fixed and known to the adversary in the beginning. (Further, for different input $\mathcal{I}$, the pair (u, S_u) might be different.) The adversary then selects all the nodes in S_u as Byzantine nodes before the execution starts. Thus the *transcript* of u is fully controlled by the Byzantine nodes as X_u is determined by the nodes in S_u. The transcript of u is the total history of messages between u and the rest of the nodes. Clearly, the decision of u depends on the choice of u's input value (0 or 1), u's ID and its transcript (which might also include the IDs of the nodes that it communicated with). Also, in a valid protocol, every node (with every distinct ID and input value) will have a distinct transcript for deciding 0 or 1, respectively. Essentially, the adversary can decide a transcript for u (depending on its input value and ID) such that the output value of u would be different than the output value of all other good nodes (assuming all other good nodes execute the algorithm without any influence from the Byzantine nodes). This will give a contradiction to the everywhere agreement. ◀

6.2 Randomized Lower Bound

Let us first consider the anonymous KT_0 setting, i.e., nodes do not have any identifiers. Later, we extend this to the non-anonymous KT_0 setting, where good nodes have unique identities and Byzantine nodes can fake their identities (in full version). Each node u has $n-1$ ports through which it connects to the $n-1$ other nodes. Thus, if a node u sends a message through a port $p \in [n-1]$ to another node v, then any message u receives through p is guaranteed to be from v. For our lower bound purpose, we assume that the ports for each node u are assigned uniformly at random and independent of port assignments for other nodes. As before, among the n nodes, a small fraction $\hat{\epsilon}n$ (assumed to be integral) for a fixed $\hat{\epsilon} > 0$ are Byzantine and denoted V^b; let $V^g = V \setminus V^b$. Nodes can individually generate uniform and independent random bits, but availability of common coins is not assumed.

We assume that our Byzantine adversary is full-information (i.e., knows the states of all nodes at all times), computationally unbounded, and is also *rushing* (i.e., it can read messages sent by good nodes before sending out its own messages). The adversary is limited to being *static*, so that it must decide which nodes are bad prior to the start of the algorithm. We formally define message complexity as follows.

▶ **Definition 13.** *For a given BA algorithm $\mathcal{A}$, the message complexity $M_{\mathcal{A}}$ (or just M when clear from context) is defined as the maximum expected number of the sum of the bits sent by good nodes. The maximum is taken over all possible adversarial strategies (i.e., choice of IDs, port assignments, input bits, and the behaviour of the Byzantine nodes) and the expectation is over the random bits used by the nodes.*

Overview of Our Approach. We show that if bad nodes can send $\Omega(n^{1+\alpha})$ messages, then the good nodes must send at least $\Omega(n^{1+\alpha/2})$ messages, for any $\alpha \in (0,1]$. If we assume not (for the sake of contradiction), then, good nodes can reach agreement while only sending $o(n^{\alpha/2})$ messages on average. Under this situation, when any good node u sends a message to any other good node v, the bad nodes can bombard v with $n^{\alpha/2}$ messages intended for denial of service (DoS). Node v will be unable to distinguish between the legitimate message

from u and these DoS messages from bad nodes. As a result, v will have to respond, on average, to $\Omega(n^{\alpha/2})$ messages from bad nodes first. This is a greater number of messages than what good nodes can afford on average. Thus, several good nodes will not be able to establish two-way contact with any other good node, which we then exploit to show the impossibility via an indistinguishability argument. The proof has been deferred to the full version [8].

▶ **Theorem 14.** *Consider any BA algorithm $\mathcal{A}$ that guarantees that good nodes reach a valid agreement in the anonymous KT_0 setting as long as the number of messages sent by Byzantine nodes is at most $B = n^{1+\alpha}$ for some $\alpha \in (0, 1]$. Then, the message complexity $M_{\mathcal{A}}$ is at least $\Omega(n^{1+\frac{\alpha}{2}})$.*

7 Conclusion

We have described an efficient randomized message-competitive algorithm to solve Byzantine agreement, Leader election and Committee election, in the synchronous communication model, with a static and full-information adversary, where nodes don't know the IDs of other nodes a priori. Our algorithm is efficient in the sense that message cost and latency grow slowly with the number of messages sent by the adversary. We also show lower bounds on message-competitive Byzantine agreement algorithms. Our lower bounds show that in general, it is not possible to do significantly better than our algorithm with respect to the number of bits sent by Byzantine nodes. A key open problem is to close the gap between upper and lower bounds for randomized protocols across all budget values as well improve the fault-tolerance to 1/3-fraction of all nodes.

References

1 Ittai Abraham, TH Hubert Chan, Danny Dolev, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. Communication complexity of byzantine agreement, revisited. In *PODC*, pages 317–326, 2019.

2 Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. FARSITE: Federated, Available, and Reliable Storage for Incompletely Trusted Environment. In 5th *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–14, 2002.

3 A. Agbaria and R. Friedman. Overcoming byzantine failures using checkpointing. *University of Illinois at Urbana-Champaign Coordinated Science Laboratory technical report no. UILU-ENG-03-2228 (CRHC-03-14)*, 2003.

4 Abhinav Aggarwal, Varsha Dani, Thomas P. Hayes, and Jared Saia. Sending a message with unknown noise. In *Proceedings of the 19th International Conference on Distributed Computing and Networking (ICDCN)*, pages 8:1–8:10, 2018.

5 Yair Amir, Claudiu Danilov, Jonathan Kirsch, John Lane, Danny Dolev, Cristina Nita-Rotaru, Josh Olsen, and David John Zage. Scaling byzantine fault-tolerant replication towide area networks. In *Proceedings of International Conference on Dependable Systems and Networks (DSN)*, pages 105–114, 2006.

6 D. P. Anderson and J. Kubiatowicz. The worldwide computer. *Scientific American*, 286(3):28–35, 2002.

7 Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics (2nd edition).* John Wiley Interscience, 2004.

8 John Augustine, Valerie King, Anisur Rahaman Molla, Gopal Pandurangan, and Jared Saia. Scalable and secure computation among strangers: Message-competitive byzantine protocols. *CoRR*, abs/1907.10308, 2019.

9 Michael Ben-Or, Elan Pavlov, and Vinod Vaikuntanathan. Byzantine agreement in the full-information model in o(log n) rounds. In Jon M. Kleinberg, editor, *Proceedings of the 38th Annual ACM Symposium on Theory of Computing, Seattle, WA, USA, May 21-23, 2006*, pages 179–186. ACM, 2006.

10 Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Maxwell Young. How to Scale Exponential Backoff: Constant Throughput, Polylog Access Attempts, and Robustness. In *Proceedings of the* 27th *Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 636–654, 2016.

11 Michael A. Bender, Jeremy T. Fineman, Mahnush Movahedi, Jared Saia, Varsha Dani, Seth Gilbert, Seth Pettie, and Maxwell Young. Resource-Competitive Algorithms. *SIGACT News*, 46(3):57–71, September 2015.

12 Bitcoin. Bitcoin website https://bitcoin.org/.

13 Joseph Bonneau, Andrew Miller, Jeremy Clark, Arvind Narayanan, Joshua A. Kroll, and Edward W. Felten. SoK: Research Perspectives and Challenges for Bitcoin and Cryptocurrencies. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, pages 104–121, 2015.

14 Nicolas Braud-Santoni, Rachid Guerraoui, and Florian Huc. Fast byzantine agreement. In *Proceedings of the 2013 ACM symposium on Principles of distributed computing*, pages 57–64. ACM, 2013.

15 Christian Cachin and Jonathan A. Poritz. Secure Intrusion-Tolerant Replication on the Internet. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 167–176, 2002.

16 Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance. *Operating Systems Review*, 33:173–186, 1998.

17 Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance and Proactive Recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

18 Allen Clement, Mirco Marchetti, Edmund Wong, Lorenzo Alvisi, and Mike Dahlin. Byzantine Fault Tolerance: The Time is Now. In *Proceedings of the Second Workshop on Large-Scale Distributed Systems and Middleware (LADIS)*, pages 1–4, 2008.

19 Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults. In *Proceedings of the Sixth USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 153–168, 2009.

20 Varsha Dani, Tom Hayes, Mahnush Movahedi, Jared Saia, and Maxwell Young. Interactive Communication with Unknown Noise Rate. *Information and computation*, 261(Part):464–486, 2018.

21 Varsha Dani, Mahnush Movahedi, Jared Saia, and Maxwell Young. Interactive Communication with Unknown Noise Rate. In *Proceedings of the Colloquium on Automata, Languages, and Programming (ICALP)*, pages 575–587, 2015.

22 Danny Dolev and Rüdiger Reischuk. Bounds on information exchange for byzantine agreement. *J. ACM*, 32(1):191–204, 1985.

23 Ethereum. Ethereum website https://ethereum.org/.

24 Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert Van Renesse. Bitcoin-NG: A Scalable Blockchain Protocol. In *Proceedings of the* 13th *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 45–59, 2016.

25 Uriel Feige. Noncryptographic selection protocols. In *Proc. of the 40th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 142–153, 1999.

26 Paul Feldman and Silvio Micali. Byzantine agreement in constant expected time (and trusting no one). In *FOCS*, pages 267–276, 1985.

27 Freenet. Freenet website https://freenetproject.org/author/freenet-project-inc.html.

28 Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, pages 51–68, 2017.

29 Seth Gilbert, Valerie King, Seth Pettie, Ely Porat, Jared Saia, and Maxwell Young. (Near) Optimal Resource-Competitive Broadcast with Jamming. In *Proceedings of the* 26th *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 257–266, 2014.

30 Seth Gilbert, Valerie King, Jared Saia, and Maxwell Young. Resource-Competitive Analysis: A New Perspective on Attack-Resistant Distributed Computing. In *Proceedings of the* 8th *ACM International Workshop on Foundations of Mobile Computing*, page 1, 2012.

31 Seth Gilbert and Maxwell Young. Making Evildoers Pay: Resource-Competitive Broadcast in Sensor Networks. In *Proceedings of the* 31th *Symposium on Principles of Distributed Computing (PODC)*, pages 145–154, 2012.

32 Shafi Goldwasser, Elan Pavlov, and Vinod Vaikuntanathan. Fault-tolerant distributed computing in full-information networks. In *47th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2006), 21-24 October 2006, Berkeley, California, USA, Proceedings*, pages 15–26. IEEE Computer Society, 2006.

33 Sergey Gorbunov and Silvio Micali. Democoin: A Publicly Verifiable and Jointly Serviced Cryptocurrency. Cryptology ePrint Archive, Report 2015/521, 2015. http://eprint.iacr.org/2015/521.

34 Jim Gray. The cost of messages. In *PODC*, pages 1–7. ACM, 1988.

35 Diksha Gupta, Jared Saia, and Maxwell Young. Proof of work without all the work. In *Proceedings of the 19th International Conference on Distributed Computing and Networking*, pages 1–10, 2018.

36 Diksha Gupta, Jared Saia, and Maxwell Young. Peace through superior puzzling: An asymmetric sybil defense. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1083–1094. IEEE, 2019.

37 Diksha Gupta, Jared Saia, and Maxwell Young. Resource burning for permissionless systems. *arXiv preprint arXiv:2006.04865*, 2020.

38 Vassos Hadzilacos and Joseph Y. Halpern. Message-optimal protocols for byzantine agreement. *Mathematical Systems Theory*, 26(1):41–102, 1993. Conference version in PODC 1991.

39 Valerie King, Steven Lonargan, Jared Saia, and Amitabh Trehan. Load balanced scalable byzantine agreement through quorum building, with full information. In *International Conference on Distributed Computing and Networking*, pages 203–214. Springer, 2011.

40 Valerie King and Jared Saia. Breaking the $O(n^2)$ bit barrier: Scalable byzantine agreement with an adaptive adversary. *J. ACM*, 58(4):18:1–18:24, 2011.

41 Valerie King, Jared Saia, Vishal Sanwalani, and Erik Vee. Scalable leader election. In *Proceedings of the Seventeenth annual ACM-SIAM Symposium on Discrete Algorithm*, pages 990–999. Society for Industrial and Applied Mathematics, 2006.

42 Valerie King, Jared Saia, Vishal Sanwalani, and Erik Vee. Towards secure and scalable computation in peer-to-peer networks. In *Foundations of Computer Science, 2006. FOCS'06. 47th Annual IEEE Symposium on*, pages 87–98. IEEE, 2006.

43 Valerie King, Jared Saia, and Maxwell Young. Conflict on a Communication Channel. In *Proceedings of the* 30th *Symposium on Principles of Distributed Computing (PODC)*, pages 277–286, 2011.

44 Jiejun Kong. *Anonymous and untraceable communications in mobile wireless networks*. Citeseer, 2004.

45 Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine Fault Tolerance. In *Proceedings of* 21st *ACM SIGOPS Symposium on Operating Systems Principles*, pages 45–58, 2007.

46 Dariusz R. Kowalski and Achour Mostéfaoui. Synchronous byzantine agreement with nearly a cubic number of communication bits: synchronous byzantine agreement with nearly a cubic

number of communication bits. In Panagiota Fatourou and Gadi Taubenfeld, editors, *PODC*, pages 84–91. ACM, 2013.

47 Harry C Li, Allen Clement, Edmund L Wong, Jeff Napper, Indrajit Roy, Lorenzo Alvisi, and Michael Dahlin. Bar gossip. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 191–204, 2006.

48 Na Li, Nan Zhang, Sajal K Das, and Bhavani Thuraisingham. Privacy preservation in wireless sensor networks: A state-of-the-art survey. *Ad Hoc Networks*, 7(8):1501–1514, 2009.

49 Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A Secure Sharding Protocol For Open Blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 17–30, 2016.

50 Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

51 Dahlia Malkhi and Michael K. Reiter. Unreliable intrusion detection in distributed computations. In *Proceedings of the 10th Computer Security Foundations Workshop (CSFW)*, pages 116–125, 1997.

52 Michael Mitzenmacher and Eli Upfal. *Probability and computing: randomization and probabilistic techniques in algorithms and data analysis*. Cambridge university press, 2017.

53 Hector Garcia Molina, Frank Pittelli, and Susan Davidson. Applications of Byzantine Agreement in Database Systems. *ACM Transactions on Database Systems (TODS)*, 11:27–47, 1986.

54 Oceanstore. The Oceanstore Project. http://oceanstore.cs.berkeley.edu.

55 Gopal Pandurangan. *Distributed Network Algorithms*. https://sites.google.com/site/gopalpandurangan/dna, 2018.

56 Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980.

57 D. Peleg. *Distributed Computing: A Locality Sensitive Approach*. SIAM, 2000.

58 Nuno Preguiça, Rodrigo Rodrigues, Christóvão Honorato, and João Lourenço. Byzantium: Byzantine-Fault-Tolerant Database Replication Providing Snapshot Isolation. In *Proceedings of the Fourth Conference on Hot Topics in System Dependability*, page 9. USENIX Association, 2008.

59 Sean C. Rhea, Patrick R. Eaton, Dennis Geels, Hakim Weatherspoon, Ben Y. Zhao, and John Kubiatowicz. Pond: The oceanstore prototype. In *Proceedings of the FAST '03 Conference on File and Storage Technologies*, pages 1–14, 2003.

60 Elaine Shi and Adrian Perrig. Designing secure sensor networks. *IEEE Wireless Commun.*, 11(6):38–43, 2004.

61 Sabrina Sicari, Alessandra Rizzardi, Luigi Alfredo Grieco, and Alberto Coen-Porisini. Security, privacy and trust in internet of things: The road ahead. *Computer networks*, 76:146–164, 2015.

62 SINTRA. SINTRA - Distributed Trust on the Internet. http://www.zurich.ibm.com/security/dti/.

63 Tor. Tor website https://www.torproject.org/.

64 Rolf H Weber. Internet of things–new security and privacy challenges. *Computer law & security review*, 26(1):23–30, 2010.

65 Alex Wright. Contemporary approaches to fault tolerance. *Commun. ACM*, 52(7):13–15, 2009.

66 Hiroyuki Yoshino, Naohiro Hayashibara, Tomoya Enokido, and Makoto Takizawa. Byzantine agreement protocol using hierarchical groups. In *Proceedings of the 11th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 64–70, 2005.

67 Mahdi Zamani, Jared Saia, and Jedidiah Crandall. TorBricks: Blocking-Resistant Tor Bridge Distribution. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 426–440. Springer, 2017.

68 Wenbing Zhao. A Byzantine Fault Tolerant Distributed Commit Protocol. In *Proceedings of the 3rd IEEE International Symposium on Dependable, Autonomic and Secure Computing*, pages 37–46, 2007.

Byzantine Lattice Agreement in Synchronous Message Passing Systems

Xiong Zheng
Electrical and Computer Engineering, University of Texas at Austin, TX, USA

Vijay Garg
Electrical and Computer Engineering, University of Texas at Austin, TX, USA

Abstract

We propose three algorithms for the Byzantine lattice agreement problem in synchronous systems. The first algorithm runs in $\min\{3h(X)+6, 6\sqrt{f_a}+6\}$ rounds and takes $O(n^2 \min\{h(X), \sqrt{f_a}\})$ messages, where $h(X)$ is the height of the input lattice X, n is the total number of processes in the system, f is the maximum number of Byzantine processes such that $n \geq 3f+1$ and $f_a \leq f$ is the actual number of Byzantine processes in an execution. The second algorithm takes $3\log n + 3$ rounds and $O(n^2 \log n)$ messages. The third algorithm takes $4\log f + 3$ rounds and $O(n^2 \log f)$ messages. All algorithms can tolerate $f < \frac{n}{3}$ Byzantine failures. This is the first work for the Byzantine lattice agreement problem in synchronous systems which achieves logarithmic rounds. In our algorithms, we apply a slightly modified version of the Gradecast algorithm given by Feldman et al [10] as a building block. If we use the Gradecast algorithm for authenticated setting given by Katz et al [12], we obtain algorithms for the Byzantine lattice agreement problem in authenticated settings and tolerate $f < \frac{n}{2}$ failures.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Lattice agreement, Byzantine Failure, Gradecast

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.32

Related Version Full paper hosted on `https://arxiv.org/abs/1910.14141`.

Funding *Vijay Garg*: This work was supported in parts by the National Science Foundation Grants CNS-1812349, CNS-1563544, and the Cullen Trust Endowed Professorship.

1 Introduction

The lattice agreement problem, introduced by Attiya et al [2], is an important decision problem in shared-memory and message passing systems. In this problem, processes start with input values from a lattice and need to decide values which are comparable to each other. Specifically, suppose each process i has input x_i from a lattice $(X, \leq, \sqcup)$ with the partial order $\leq$ and the join operation $\sqcup$, it has to output a value y_i also in X such that the following properties are satisfied. 1) **Downward-Validity**: $x_i \leq y_i$ for each correct process i. 2) **Upward-Validity**: $y_i \leq \sqcup\{x_i \mid i \in [n]\}$. 3) **Comparability**: for any two correct processes i and j, either $y_i \leq y_j$ or $y_j \leq y_i$.

In shared-memory systems, algorithms for the lattice agreement problem can be directly applied to solve the atomic snapshot problem [1, 2]. This was the initial motivation for studying this problem. The application of lattice agreement in message passing systems has been explored only recently. Failero et al [9] were the first to apply lattice agreement for building a special class of linearizable replicated state machines, which can support query operation and update operation, but not mixed query and update operation. Traditionally, consensus based protocols are applied to build linearizable replicated state machines. However, consensus based protocols do not provide deterministic termination guarantee in the presence of failures in the system, since the consensus problem cannot be solved with even one failure

© Xiong Zheng and Vijay Garg;
licensed under Creative Commons License CC-BY
34th International Symposium on Distributed Computing (DISC 2020).
Editor: Hagit Attiya; Article No. 32; pp. 32:1–32:16
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

in an asynchronous system [11]. The lattice agreement problem instead has been shown to be a weaker decision problem than consensus. It can be solved in an asynchronous system when a majority of processes is correct. Thus, linearizable replicated state machines built based on lattice agreement protocols have the advantage of termination even with failures. Another application of lattice agreement in distributed systems is to build atomic snapshot objects. Efficient implementation of atomic snapshot objects in crash-prone asynchronous message passing systems is important because they can make design of algorithms in such systems easier. The paper [2] presents a general technique for applying algorithms for the lattice agreement problem to solve the atomic snapshot problem. By using the same technique, algorithms for lattice agreement problem in distributed systems can be directly applied to implement atomic snapshot objects in crash-prone message passing systems. Essentially, an atomic snapshot object needs to provide linearizabilty for all processes, which decides on some total ordering of operations. In the lattice agreement problem, processes need to output values which lie in a chain of the input lattice, which is also a total ordering.

The lattice agreement problem in crash failure model has been studied both in synchronous and asynchronous systems. In synchronous systems, a $\log n$ rounds recursive algorithm based on "branch-and-bound" approach is proposed by Attiya et al [2] with message complexity of $O(n^2)$. The basic idea of their algorithm is to divide processes into two groups based on ids and let processes in the first group send values to processes in the second group. Each process in the second group takes join of received values. Then, this procedure continues within each subgroup. Their algorithm can tolerate at most $n-1$ process failures. Later, the paper by Mavronicolasa et al [14] gave an algorithm with round complexity of $\min\{1 + h(X), \lfloor(3 + \sqrt{8f+1}/2)\rfloor\}$, for any execution where at most $f < n$ processes may crash and $h(X)$ denotes the height of the input lattice X. Their algorithm has the early-stopping property and is the first algorithm with round complexity that depends on the actual height of the input lattice. The best upper bound for the lattice agreement problem in crash-failure model is given by Xiong et al [21], which is $O(\log f)$ rounds. The basic idea of their algorithm is again to divide processes into two groups based height of received values at each round and trying to achieve agreement within each group recursively.

In asynchronous systems, the lattice agreement problem was first studied by Faleiro et al in [9]. They present a Paxos style protocol when a majority of processes are correct. Their algorithm needs $O(n)$ asynchronous rounds in the worst case. The basic idea of their algorithm is to let each process repeatedly broadcast its current value to all at each round and update its value to be the join of all received values until all received values at a certain round are comparable with its current value. Later, Xiong et al [21] propose an algorithm which improves the round complexity to $O(f)$ rounds. For the best upper bound, Xiong et al [20] present an algorithm for this problem with round complexity of $O(\log f)$, which applies similar idea as [21] but with extra work to take care of possible arbitrary delay of messages in asynchronous systems.

The Byzantine failure model was first considered by Lamport et al [13] for the study of the Byzantine general agreement problem. For the lattice agreement problem in Byzantine failure model, Nowak et al [15] give an algorithm for a variant of the lattice agreement problem on cycle-free lattices that tolerates up to $f < \frac{n}{(h(X)+1)}$ Byzantine faults in asynchronous systems, where $h(X)$ is the height of the input lattice X. In their problem, the original Downward-Validity and Upward-Validity requirement are replaced with a different validity definition, which only requires that for each output value y of a correct process, there must be some input value x of a correct process such that $x \leq y$. With their validity definition, however, corresponding algorithms are not suitable for applications in atomic snapshot

objects and linearizable replicated state machines, since each process would like to have their proposal value included its output value. A more closely related work is the preprint by Di Luna et al [8], which proposes a reasonable validity condition and presents the first algorithm for asynchronous systems. Their algorithm takes $O(f)$ rounds. The basic idea of their algorithm is to first use the asynchronous Byzantine reliable broadcast primitive [5, 17] to let all correct processes disclose their input values to each other, based on which each correct process constructs a set of safe values. This set of safe values are the only values a correct process will possibly deliver in future rounds. After the disclosure phase, the remaining steps are similar to the algorithms given in [9, 21] for the lattice agreement problem in crash failure model, except that each process delivers a message only if all the values included in it are contained in its safe value set. Our algorithms assume synchronous systems but achieve exponential improvement in terms of round complexity.

For related works on application of lattice agreement, Faleiro et al [9] give procedures to build a linearizable and serializable replicated state machine which only supports query operation and update operation but not mixed queryte operation, based on lattice agreement protocols. Later, Xiong et al [20] propose some optimizations for their procedure for implementing replicated state machines in practice, specifically, they proposed a method to truncate the logs maintained. The recent paper by Skrzypczak et al [16] improves the procedure given in [9] in terms of memory consumption, at the expense of progress, and also demonstrates higher throughput.

Our main contribution in this paper is summarized in Table 1. Our first algorithm is early stopping because its round complexity depends on f_a: the actual number of Byzantine processes in an execution. Its basic idea is to let processes communicate using a modified Gradecast primitive and detect Byzantine processes along the way. The second and third algorithm are not early stopping but take logarithmic number of rounds. A building block of both algorithms is to construct a classifier procedure as in [2, 3, 21, 20] using a variant of the Gradecast primitive, but now the classifier procedure is Byzantine-tolerant and needs to guarantee different properties.

Table 1 Our Results.

Problem	Reference	Rounds	Resilience
BLA	Concurrent work [7]	$O(\log f)$	$f < \frac{n}{4}$
	This paper	$\min\{3h(X)+6, 6\sqrt{f_a}+6\}$	$f < \frac{n}{3}$
		$3\log n + 3$	
		$4\log f + 3$	

In a concurrent work by Di Luna et al [7], they show that the Byzantine lattice agreement problem cannot be solved with $f \geq \frac{n}{3}$ failures in a synchronous systems. This shows that our algorithms achieve optimal resilience. In their paper, they give an algorithm for this problem which takes $O(\log f)$ rounds and tolerates $f < \frac{n}{4}$ failures, whereas our algorithms tolerate $f < \frac{n}{3}$ failures. With the assumption of digital signatures, they can improve the resilience to be $f < \frac{n}{3}$, whereas our algorithms tolerate $f < \frac{n}{2}$ failures.

2 System Model and Problem Definition

System Model: We assume a distributed message system with n processes in a completely connected topology, denoted as $p_1, ..., p_n$. Every process can send messages to every other process. We consider synchronous systems, which means that message delays and the duration

of the operations performed by the process have an upper bound on the time. We assume that processes can have Byzantine failures but at most $f < n/3$ processes can be Byzantine in any execution of the algorithm. We use parameter f_a to denote the actual number of Byzantine processes in a system. By our assumption, we must have $f_a \leq f$. Byzantine processes can deviate arbitrarily from the algorithm. We say a process is correct or non-faulty if it is not a Byzantine process. We use C to denote the set of correct processes in an execution. We assume that the underlying communication system is reliable.

The Byzantine Lattice Agreement (BLA) Problem: Let $(X, \leq, \sqcup)$ be a finite join semi-lattice with the partial order $\leq$ and the join operation $\sqcup$. Two values u and v in X are comparable iff $u \leq v$ or $v \leq u$. The join of u and v is denoted as $\sqcup\{u, v\}$. X is a *join semi-lattice* if the join exists for every nonempty finite subset of X. As customary in this area, we use the term *lattice* instead of *join semi-lattice* in this paper for simplicity. More background on join semi-lattices can be found in [6].

In the Byzantine lattice agreement problem, each process p_i can propose a value x_i in X and must decide on some output y_i also in X in the presence of at most f Byzantine processes in the system. Let C denote the set of correct processes. Let f_a denote the actual number of Byzantine processes in the system. An algorithm is said to solve the Byzantine lattice agreement problem if the following properties are satisfied:

Comparability: For all $i \in C$ and $j \in C$, either $y_i \leq y_j$ or $y_j \leq y_i$.

Downward-Validity: For all $i \in C$, $x_i \leq y_i$.

Upward-Validity: $\sqcup\{y_i \mid i \in C\} \leq \sqcup(\{x_i \mid i \in C\} \cup B)$, where $B \subset X$ and $|B| \leq f_a$.

Remark: The Upward-Validity given by Attiya et al [2] is not suitable in the presence of Byzantine processes, since the input value for a Byzantine process is not defined. Thus, the extra B set is used to accommodate for possible values from Byzantine processes. The above Upward-Validity is similar to the Non-Triviality defined in [8]. The only difference is that the extra B set in [8] is required to have size at most f, which is the resilience parameter. One may argue that if a Byzantine process proposes the largest element of the input lattice, then correct processes may always decide on the largest element. For applications, we can impose an additional constraint on the initial proposal of all processes. For example, in the case of a Boolean lattice, we can require that the initial proposal for any process must be a singleton. More generally, we can impose the requirement that the initial proposal of any process must have the height less than some constant.

In this paper, for a given set $V \subseteq X$, we use $\mathcal{L}(V)$ to denote the join-closed subset of X that includes all elements in V. Clearly, $\mathcal{L}(V)$ is also a join semi-lattice. The height of a value v in a lattice X is defined as the length of longest chain from any minimal value to v, denoted as $h_X(v)$ or $h(v)$ when it is clear. The height of a lattice X is the height of its largest value in this lattice, denoted as $h(X)$. For two lattices $\mathcal{L}_1$ and $\mathcal{L}_2$, we use $\mathcal{L}_1 \subseteq \mathcal{L}_2$ to mean that $\mathcal{L}_1$ is a sublattice of $\mathcal{L}_2$.

3 Early Stopping Algorithm for BLA

In this section, we present an early stopping algorithm for the BLA problem, which applies a slightly modified version of the Gradecast algorithm given by Feldman et al [10] as a building block. The algorithms takes $\min\{3h(X) + 6, 6\sqrt{f_a} + 6\}$ rounds. Our algorithm has two primary ingredients which are quite different from the algorithm given in [9, 21] for the crash failure model. In the Byzantine failure model, correct processes can receive arbitrary values from a Byzantine process. In order to guarantee **Upward-Validity**, we do not want correct

processes to accept arbitrarily many values sent from a Byzantine process. The idea in [8] is to construct a safe value set, which stores the values reliably broadcast by each process at the first round. Later on, each process only delivers a received message if the values included in this message are contained in its safe value set. In this way, correct processes would not deliver arbitrary values sent by Byzantine processes. However, this idea can only provide $O(f)$ rounds guarantee.

To obtain the $O(\sqrt{f_a})$ rounds guarantee, our first idea is to let each correct process in our algorithm keep track of a lattice, which we call the safe lattice, instead of just a set of values. At each round, each correct process ignores all values received which are not contained in this safe lattice. By carefully updating this safe lattice of each correct process, our algorithm ensures that the value sent from a correct process is always in the safe lattice of any other correct process and Byzantine process cannot introduce arbitrary values to break the **Upward-Validity** condition. To get the $O(\sqrt{f_a})$ bound, another crucial ingredient of our algorithm is to apply the Gradecast algorithm at each round to detect the Byzantine processes which sends different values to different correct processes and let each correct process ignores messages from these processes. This idea is used in [4] to solve the Byzantine consensus problem in synchronous systems.

3.1 The Modified Gradecast Algorithm

Gradecast [10] is a three-round distributed algorithm that ensures some properties that are similar to those of broadcast. The Gradecast procedure has two parameters. The first one specifies the leader of the Gradecast and the second one represents the value that the leader would like to send. The output of process i in the Gradecast of leader p is a triple $< p, v_p^i, c_p^i >$ where v_p^i is the value process i thinks the leader p has sent and c_p^i is the score assigned by i for the leader. The score assigned by process i for the leader is among $\{0, 1, 2\}$. We say c_p^i is the score assigned to the value or the leader by process i.

For our purpose, we do a slight modification of the Gradecast algorithm from [10] to enable processes to filter out some invalid values received and ignore messages from known Byzantine processes. The modified Gradecast algorithm can be found in out full paper [18]. We do the following modifications: 1) Let each process store a safe lattice to filter out all received values which are not in the lattice. We call this lattice: *the safe lattice*. Specifically, each process i keeps a safe value set, denoted as SV_i. This set is updated by process i at each round of the main algorithm. From SV_i, each process i constructs $\mathcal{L}(SV_i)$, the safe lattice as the join-closed subset of X which includes all values in SV_i. 2) Let each process store a bad set, which stores the Byzantine processes known by this process. Each process ignores the messages sent by processes in this set in the modified Gradecast algorithm. This bad set is also updated at each round of the main algorithm.

We assume that a correct leader always gradecasts a value which lies in the safe lattice of each correct process and the bad set of each correct process does not contain any correct process. We will show that these assumptions hold when we invoke the modified Gradecast algorithm as a substep in our main algorithm.

▶ **Lemma 1.** *Assume that a correct leader always gradecast some value v which lies in the safe lattice of each other correct process and the bad set of each correct process does not contain a correct process. Then the modified Gradecast algorithm satisfies the following properties.*

1. *If the leader p is non-faulty then $v_p^i = v$ and $c_p^i = 2$, for any non-faulty i;*
2. *For every non-faulty i and j: if $c_p^i > 0$ and $c_p^j > 0$ then $v_p^i = v_p^j$;*
3. *$|c_p^i - c_p^j| \leq 1$ for every non-faulty i and j.*

3.2 The Main Algorithm

The main algorithm, shown in Alg. 1, runs in synchronous rounds. For ease of presentation, we use rounds to mean the iterations in the **for** loop of the main algorithm. We call the rounds taken by the Gradecast step as sub-rounds. We assume for now that there is an upper bound F on the number of rounds of the main algorithm. We will establish the accurate value of F later and show that each process must decide on an output by round F. Initially, the bad set B_i and the safe value set SV_i for each process i are empty. Process i regards any value received as valid in the gradecast of the first round.

Algorithm 1 Early Stopping Algorithm for the BLA Problem.

Algorithm for Process i:

$v_i := x_i$ //value held by i during the algorithm $B_i := \emptyset$ //set of faulty processes known by process i

$SV_i := \emptyset$ //set of safe values for process i

1: **for** $r := 1$ to F // F is an upper bound on the number of rounds

2: **Gradecast**(i, v_i)

3: Let $< j, u_j, c_j >$ denote that process j gradecast u_j with score c_j

4: Define $U_i^1 := \{u_j \mid < j, u_j, c_j > \wedge c_j \geq 1, j \in [n]\}$,
$U_i^2 := \{u_j \mid < j, u_j, c_j > \wedge c_j = 2, j \in [n]\}$

5: Set $B_i := B_i \cup \{j \mid < j, *, c_j > \wedge c_j \leq 1, j \in [n]\}$

6: Set $SV_i := U_i^1$

7: **if** v_i comparable with each value in U_i^2 **then** decide on v_i, but continue execution

8: Set $v_i := \sqcup\{u \mid u \in U_i^2\}$

9: **endfor**

At each round, each process invokes the modified Gradecast algorithm with its current value and acts as the leader. So there are at least $n - f$ Gradecast instances running at each round, with each instance corresponding to one correct process. After the Gradecast phase, each process i has a set of triples, one for each process which invoked Gradecast as the leader. A triple consists of the leader id, the value sent by the leader, and the score assigned by i. From these triples, processes i updates its bad set B_i and safe value set SV_i as follows. At line 5, process i includes all processes which are assigned score at most one into its bad set B_i and ignores all messages sent from processes in B_i at future rounds. Process i also updates its safe value set SV_i to be the union of all values gradecast by processes with score at least one. By updating the safe value set in this way, we can ensure that the current value of a correct process is in the safe lattice of every other correct process. Thus, the value gradecast by a correct process in the next round is valid for every other correct process, which implies property 1 of Gradecast. On the other hand, this safe value set also prevents Byzantine processes from gradecasting an arbitrary value, i.e, Byzantine processes can only gradecast values that belong to the safe lattice.

For the deciding condition at line 7, each process decides on its current value at a certain round if all values gradecast by processes with score 2 are comparable with its current value. A process keeps executing the algorithm even if it has decided on an output. It updates its current value to be the join of all values gradecast with score 2 and starts the next round.

3.3 Correctness and Complexity

We now prove the correctness of our algorithm. Due to space limitation, we put the proof of most lemmas and theorems in our full paper [18]. The variables used for the proof are defined in Table 2. The main algorithm has the following properties.

Table 2 Notations for the Proof of Correctness of the Early Stopping Algorithm.

Variable	Definition
v_i^r	Value of process i at the end of round r
v^r	Auxiliary variable. The join of values of all correct processes at the end of round r, i.e., $v^r := \sqcup\{v_i^r \mid i \in C\}$
SV_i^r	The safe value set held by process i at the end of round r
S^r	Auxiliary variable. The union of all safe value sets held by correct processes at the end of round r, i.e, $S^r = \cup\{SV_i^r \mid i \in C\}$
s^r	Auxiliary variable. The join of all values in S^r, i.e, $s^r = \sqcup\{v \mid v \in S^r\}$
c_j^i	The score process i assigned to process j in the Gradecast with j as the leader.

▶ **Lemma 2.** *Let i and j be any two correct processes. For any $1 \leq r \leq F$, the main algorithm satisfies the properties below.*
(p1) $v_i^r \in \mathcal{L}(SV_j^r)$ (p2) $v^r \in \mathcal{L}(SV_i^r)$ (p3) $v^r \leq s^r$ (p4) $v_i^{r-1} < v_i^r$ if process i is undecided at the end of round r
(p5) $v^r < v^{r+1}$ if at least one correct process is not decided at the end of round r
(p6) $\mathcal{L}(S^{r+1}) \subseteq \mathcal{L}(S^r)$ (p7) $s^{r+1} \leq s^r$ (p8) For each correct process i, its bad set B_i never contains a correct process (p9) $v^r \leq v_i^{r+1}$

Property $(p1)$ and $(p8)$ immediately justify the assumption in Lemma 1. Thus, all properties of Gradecast are satisfied. Property $(p3)$ indicates that the join of all values of correct processes is less than the join of all values in the safe value sets of all correct processes at any round. Property $(p4)$ and $(p5)$ imply that the join of all values of correct processes is strictly increasing. Property $(p7)$, implied by $(p6)$, indicates that the join of all values in the safe value sets of all correct processes is non-increasing. Then, we must have $v^r = s^r$ at some round r. After this round, the deciding condition must be satisfied for each process.

Now we show the algorithm satisfies all the properties required by the BLA problem. For now we assume that each process decides within F rounds. We show the accurate value of F when we analyze the round complexity.

▶ **Lemma 3.** *The values decided by correct processes satisfy all the properties of BLA.*

Proof. (Sketch) **Comparability**. If two processes decide at the same round, their decision value must be comparable by the deciding condition. Otherwise, the process decides in a later round must receive the decision value of the other process.

Upward-Validity. The safe lattice kept by each process guarantees that each Byzantine process can introduce at most one value into the decision value of correct processes. ◀

We now analyze the round and message complexity of our algorithm. The following lemma along with property $(p5)$ and $(p7)$ of Lemma 2 guarantees the termination of our algorithm. It can be derived from $(p1)$, $(p7)$, and $(p9)$ of Lemma 2 and the decision condition.

▶ **Lemma 4.** *If $v^r = s^r$ at the end of some round r, then all undecided correct processes decide in at most 2 rounds.*

▶ **Lemma 5.** *$F \leq h(X) + 2$, where X is the input lattice and $h(X)$ is the height of X.*

Proof. (Sketch) $(p3)$, $(p5)$ and $(p7)$ of Lemma 2 implies that $v^r = s^r$ in at most $h(X)$ rounds. Then, Lemma 4 implies that $F \leq h(X) + 2$. ◀

We now show that the algorithm takes $O(\sqrt{f})$ rounds. We first observe that if a process is in the bad set of each correct process, then its gradecast will be graded with score 0 by each correct process. We introduce the notion of terrible processes. A process is **terrible** at round r if it is graded with score 2 by at least one correct process in each round before r and no correct process grades it with score 2 at round r. From the above definition, we observe that the terrible processes at each round are included into B_i for each correct i at line 5 of the algorithm. So, a Byzantine process can be terrible at most once.

▶ **Lemma 6.** *Suppose there are f_r terrible processes at round r, then each process decides within $r + f_r + 2$ rounds.*

▶ **Lemma 7.** *$F \leq 2\sqrt{f} + 2$, where f is the maximum number of Byzantine failures in the system such that $n \geq 3f + 1$.*

Proof. Consider the first $\sqrt{f}$ rounds. At least one of these rounds has less than $\sqrt{f}$ terrible processes. By Lemma 6, starting from that round, each undecided process needs at most $\sqrt{f} + 2$ more rounds to decide. Thus, the total number of rounds is at most $2\sqrt{f} + 2$. ◀

The obtain the $O(\sqrt{f_a})$ rounds guarantee, we let each correct process dynamically update its termination round, which denotes the round number such that a correct process terminates the algorithm in any case. Specifically, if a process includes t Byzantine processes into its bad set, then this process runs $\sqrt{t} + 2$ more rounds and terminate.

▶ **Theorem 8.** *There is a $\min\{3h(X) + 6, 6\sqrt{f_a} + 6\})$ rounds algorithm for the Byzantine lattice agreement problem in synchronous systems, which can tolerate $f < \frac{n}{3}$ Byzantine failures. f_a is the actual number of Byzantine failures. The term $h(X)$ is the height of the input lattice X. The algorithm takes $O(n^2 \min\{h(X), \sqrt{f_a}\})$ messages.*

▶ **Corollary 9.** *There is a $\min\{4h(X) + 8, 8\sqrt{f_a} + 8\}$ rounds algorithm for the authenticated (allow digital signatures) BLA problem in synchronous systems, which can tolerate $f < \frac{n}{2}$ Byzantine failures. The algorithm takes $O(n^2 \min\{h(X), \sqrt{f_a}\})$ messages.*

Proof. Using the 4-round Gradecast algorithm [12] in the authenticated setting that tolerates $f < \frac{n}{2}$ Byzantine failures immediately gives the result. ◀

4 $O(\log n)$ Rounds Algorithm for the BLA problem

The $3 \log n + 3$ round algorithm shown in this section is inspired by algorithms proposed for the crash failure model in [2, 21]. The basic idea is to divide a group of processes into the slave subgroup and the master subgroup based on process ids, and ensure the property that the value of any correct process in the slave group is at most the value of any correct process in the master group. With the above property, if we recurse within each subgroup, then all correct processes can obtain comparable values in $O(\log n)$ rounds.

In the Byzantine failure model, however, simply ensuring the above property is not enough. For example, suppose we divided a group of processes G into the slave group $S(G)$ and the master group $M(G)$ such that the above property is satisfied. Suppose there is a Byzantine process in $S(G)$, it might send a value to some correct process in $S(G)$ in a later round such that the value is not known by correct processes in $M(G)$. Then, a correct slave process might have a value which is greater than some master process.

In order to prevent such cases, our algorithm introduces two novel ideas. First, when we divide a group into the slave subgroup and the master subgroup, we apply a modified Gradecast algorithm to guarantee that the value of a slave process is at most the value

of a master process. The Gradecast algorithm serves the same purpose as the *Classifier* procedure as given in [21, 20]. A nice property of the modified Gradecast algorithm is that if some correct process assigns score 2 for the value gradecast by the leader, then each other correct process assigns score at least 1 for this value. Suppose we let each process in a group gradecast its value. Let U^2 denote the set of values assigned score of 2 by some correct process. Let U^1 denote the set of values assigned score of at least 1 by some correct process. Then, we must have $U^2 \subseteq U^1$. If each process in the master group updates their value to U^1 and each process in the slave group updates their value to U^2, then it is guaranteed that the set of all values of the slave group is a subset of the values of each master process.

However, the above property is only guaranteed at the current recursion level. Suppose there is a Byzantine process in the slave group, then it can gradecast a new value, which is not contained in the value set of some master process, to correct processes in the slave group. Then, the above property that the value of any correct slave process is at most the value of any correct master process does not hold any more. We need to ensure that the values of all slave processes are always a subset of the values of each master process when the recursion within each subgroup continues. To achieve that, we introduce a second novel idea. We let each process keep track of a safe value set for each other process and regard any value received from that process but not in the safe value set as invalid. This is also different from the algorithm in previous section, where each process just keeps track of one single safe lattice for all. These safe values sets are used to restrict what values a process in a slave group can send. If we can guarantee that the union of all safe value sets for processes in the slave group is a subset of the value set of each master process, then we can ensure that the above property continues to hold.

4.1 The SetGradecast Algorithm

In the $3\log n + 3$ rounds algorithm, a process needs to gradecast a set of values instead of just one single value. Thus, we propose the **SetGradecast** algorithm (presented in the full paper [18]) which is similar to the **Gradecast** algorithm, except that it is used to gradecast a set of values. In the $3\log n + 3$ rounds algorithm, each process i keeps track of a safe array S_i of size n with $S_i[j]$ being a safe value set for process j. Process i considers a value v received from process j as valid if $v \in S_i[j]$, otherwise invalid. Process i uses S_i to filter out invalid values received from any process in the **SetGradecast** algorithm. We show how to construct and update the safe value array for each process in the main algorithm.

In the **SetGradecast** algorithm, we assume that the leader needs to gradecast a set of distinct values, which can be guaranteed by introducing some unique tags for each value. If a process receives a message which contains duplicate values from some leader, the leader must be Byzantine. It just ignores the message. Each process i returns a triple $< j, R_j, C_j >$ when process j invokes the **SetGradecast** as the leader. The set R_j is the set of values gradecast by process j with score at least 1 and the map C_j stores the score assigned by process i for each value in R_j. Let v be an arbitrary value gradecast by the leader. Let c_v^i denote the score of v assigned by process i. Then, we have the following lemma.

▶ **Lemma 10.** *Algorithm **SetGradecast** has the following properties.*

1. *If a value v gradecast by a correct leader i is in the safe value set of each correct process j for i, i.e., $v \in S_j[i]$ for each correct $j \in [n]$, then the score of v assigned by each correct j must be 2, i.e., $c_v^j = 2$.*
2. *Let v be an arbitrary value gradecast by the leader. Then $|c_v^i - c_v^j| \leq 1$ for any two correct process i and j.*
3. *If a value v gradecast by a leader i is not in the safe value set of any correct process for i, i.e., $v \notin S_j[i]$ for any $j \in C$, then $c_v^j = 0$ for each $j \in C$.*

Algorithm 2 The $3 \log n + 3$ Rounds Algorithm for the BLA Problem.

Let $\mathcal{G}_r$ denote the collection of groups at round r, initially $\mathcal{G}_1 = \{G_1\}$
x_i: input value for process i V_i: value set of process i with $V_i := \{x_i\}$ initially.
S_i: an array of size n with $S_i[j]$ being the safe value set for process j.

1: **for** each process i, **in parallel do** /* Build the Initial Safe Array */
2: Process i invokes **Gradecast**(i, x_i)
3: Let $< j, v_j, c_j >$ denote the tuple obtained from the gradecast of p_j
4: Set $S_i[k] := \{v_j \mid c_j \geq 1, j \in [n]\}$ for each process $k \in [n]$
5: **endfor**
6: **for** $r := 1$ to $\log n$
7: Divide each group $G \in \mathcal{G}_r$ into the slave subgroup $S(G)$ and the master subgroup $M(G)$ based on process ids
8: Let $\mathcal{G}_{r+1}$ denote the collection of new groups
9: Each slave process p executes **SetGradecast**(p, V_p)
10: **for** each process i, **in parallel do**
11: Let $< j, Val^i_j, C^i_j >$ denote the leader-value-score triple that p_i obtained for p_j
12: Let R^i_j denote the set of values assigned score 2 by p_i in the gradecast of p_j, i.e, $R^i_j := \{v \in Val^i_j \mid C^i_j[v] = 2\}$
13: **for** each group $G \in \mathcal{G}_r$
14: Let U_1 denote the set of values sent by processes in $S(G)$ with score ≥ 1, i.e., $U_1 := \bigcup_{j \in S(G)} Val^i_j$
15: Let U_2 denote the set of values sent by processes in $S(G)$ with score 2, i.e., $U_2 := \bigcup_{j \in S(G)} R^i_j$
16: $S_i[j] := U_2$ for each process $j \in S(G)$
17: $S_i[j] := S_i[j] \cup U_1$ for each process $j \in M(G)$,
18: **if** $i \in S(G)$ **then** $V_i := U_2$
19: **elif** $i \in M(G)$ **then** $V_i := U_1$
20: **endfor**
21: **endfor**
22: **endfor**
23: Output $y_i := \sqcup\{v \in V_i\}$

4.2 The Main Algorithm

The $3 \log n + 3$ rounds algorithm is shown in Alg. 2. In the algorithm, each process i stores a value set V_i which is updated at each round. Initially, $V_i = \{x_i\}$. Each process i keeps track of a safe array S_i of size n with $S_i[j]$ being the safe value set for j. Process i regards any value received from j which is not in $S_i[j]$ as invalid and thus ignores it. Different processes may have different safe value sets for a process j. Initially, all processes are in the same group, denoted as G_1. During the algorithm, processes might be divided into different groups. The algorithm proceeds as follows.

The initial round at lines 1-5 is used to build the initial safe array of each process i. At this round, each process i invokes the **Gradecast** algorithm to send its input value x_i to all. Each process i constructs the same initial safe value set for each process j, which includes all values gradecast by some process and assigned score of at least 1 by process i. We will show

later that this initial round of gradecast guarantees **Upward-Validity** of the BLA problem. Intuitively, this is because each Byzantine process can only introduce one value into the safe value sets by properties of Gradecast.

In lines 6-22, at each round r from 1 to $\log n$, each group G is divided into two subgroups: the slave group $S(G)$ and the master group $M(G)$, where $S(G)$ contains all processes in G with ids in the lower half and $M(G)$ contains all processes in G with ids in the upper half. Each process in $S(G)$ invokes **SetGradecast** to gradecast its current value set to all processes. Processes in $M(G)$ do not gradecast their value sets. After this step, for each group G, process i obtains a set of values gradecast by processes in $S(G)$. From line 13 to line 20, process i updates its safe value set S_i and its value set V_i based the values obtained in all **SetGradecast** instances. At line 16, process i updates its safe value set for each process $j \in S(G)$ to be the values gradecast with score 2 by some process in $S(G)$. At line 17, it updates the safe value set for each process in $M(G)$ to be the values gradecast with score at least 1 by some process in $S(G)$. If process i is a slave process in $S(G)$, it updates its value set to be the set of values gradecast by processes in $S(G)$ and assigned score of 2. If it is a master process in $M(G)$, it updates its value set to be the set of values gradecast by processes in $S(G)$ with score at least 1.

Table 3 Notations for the Proof of Correctness of the $O(\log n)$ Algorithm.

Variable	Definition
V_i^r	The value set held by process i at the end of round r.
S_i^r	The safe value array of i at the end of round r.
SF_j^r	Auxiliary variable. The union of the safe value sets of all correct process for j at the end of round r, i.e., $SF_j^r := \{S_i^r[j] \mid i \in C\}$
SF_G^r	Auxiliary variable. The union of the safe value sets of all correct process for processes in group G at the end of round r, i.e., $SF_G^r := \bigcup_{j \in G} SF_j^r$

4.3 Correctness and Complexity

Now we analyze the correctness and complexity of our algorithm. For any group G, let $S(G)$ and $M(G)$ denote the slave group and the master group obtained when dividing G. The variables we use for analysis are given in Table 3. Detailed proof of lemmas can be found in our full paper [18].

▶ **Lemma 11.** *Let G be a group that is divided into $S(G)$ and $M(G)$ at round r. Then (p1) $SF_{S(G)}^r \subseteq SF_G^{r-1}$ (p2) $SF_{M(G)}^r \subseteq SF_G^{r-1}$ (p3) For each $i \in G \cap C$, $V_i^r \subseteq SF_G^{r-1}$*

The following lemma shows that if a value of correct process i is contained in the safe value set of each correct process for process i, then this value remains in the value set of process i. It also implies **Downward-Validity**.

▶ **Lemma 12.** *Consider an arbitrary value $v \in V_j^r$ of correct process j. If it is contained in $S_i^r[j]$ for each correct process i, then we have (1) $v \in S_i^t[j]$ for each correct process i and $t \geq r$. (2) $v \in V_j^t$ for any $t \geq r$.*

The following lemma shows that the union of all safe value sets of correct processes for slave processes will always be a subset of the values of each master process.

▶ **Lemma 13.** *Let G be a group which is divided into $S(G)$ and $M(G)$ at round r. Then $SF_{S(G)}^r \subseteq V_j^t$ for each correct $j \in M(G)$ and any round number $t \geq r$.*

▶ **Theorem 14.** *There is a $3\log n + 3$ rounds algorithm for the Byzantine lattice agreement problem in synchronous systems, which can tolerate $f < \frac{n}{3}$ Byzantine failures. The algorithm takes $O(n^2 \log n)$ messages.*

Proof. (Sketch) **Comparability**. Let G denote the last group both i and j belong to at round r. W.l.o.g, suppose $i \in S(G)$ and $j \in M(G)$. By ($p1$), ($p2$) and ($p3$) of Lemma 11, we have $V_i^{\log n} \subseteq SF^r_{S(G)}$. Lemma 13 implies that $SF^r_{S(G)} \subseteq V_j^{\log n}$. Thus, $V_i^{\log n} \subseteq V_j^{\log n}$.

Upward-Validity. The initial round ensures that each Byzantine process can introduce at most one value into the initial safe value sets of correct processes. ◀

▶ **Corollary 15.** *There is a $4\log n + 4$ rounds algorithm for the authenticated BLA problem in synchronous systems which can tolerate $f < \frac{n}{2}$ Byzantine failures, where n is the number of processes. The algorithm takes $O(n^2 \log n)$ messages.*

5 $O(\log f)$ Algorithm for the BLA Problem

In this section, we present an algorithm for the BLA problem which takes $4\log f + 3$ synchronous rounds. The $O(\log f)$ algorithm by Xiong et al. in [21] for the crash failure model uses a crash-tolerant classifier procedure. The idea of using a classifier procedure to obtain comparable views is initially applied to implement atomic snapshot objects in shared memory systems by Attiya et al. [3]. Their crash-tolerant classifier procedure divides a group of processes into two subgroups: the master group and the slave group based on a threshold parameter k and guarantees the following properties. (C1) The value of any slave process $\leq$ the value of any master process. (C2) The value of any master process has height in the input lattice $> k$. (C3) The join of all slave values has height $\leq k$.

With the above properties, the classifier procedure can be recursively applied within each subgroup and all processes have comparable values after $O(\log f)$ rounds by setting the knowledge threshold k in a binary way as follows. Initially, all processes are in the same group with initial knowledge threshold $n - \frac{f}{2}$. Consider a group G at level r with knowledge threshold k, then the slave group of G has knowledge threshold $k - \frac{f}{2^{r+1}}$ and the master group of G has knowledge threshold $k + \frac{f}{2^{r+1}}$. If all processes exchange their values before recursively invoking the classifier procedure, each correct process must have at least $n - f$ values and have at most n values. Then, after $\log f$ levels of recursion, by applying property (C1) and (C2) recursively, all processes in different groups must have comparable values and all processes in the same group must have the same value.

The crash-tolerant classifier procedure in [21] is quite simple. All processes within the same group exchange their current values. If a process obtains a value set with size greater than k, it is classified as a master. Otherwise, it is classified as a slave. A slave process keeps its value unchanged. A master process updates its value to be the join of all values received.

In presence of Byzantine processes, however, properties (C1)-(C3) are not sufficient for subgroups to invoke the classifier procedure recursively due to the following reason. Even if we can guarantee (C1) and (C3) at the current classifier, Byzantine processes can introduce additional values into the slave group when processes in the slave subgroup invokes the classifier within themselves. Then, properties (C1) and (C3) can be violated.

In our algorithm, we apply a *Byzantine-tolerant classifier procedure* to divide a group of processes into two subgroups: the slave group and the master group. A group of processes apply the Byzantine-tolerant classifier procedure to update their value sets and decide which subgroup they are classified into. In our algorithm, each process i keeps track of a value set V_i: a set of values, which is updated in the classifier procedure. Similar to the $O(\log n)$

algorithm, we let each process store a safe value set for each group which restricts the values that processes in this group can send. If SF is the union of the safe value sets of all correct processes for the slave group, then, our algorithm guarantees that the value set of any slave process must be always a subset of SF. The following properties are guaranteed after a group of processes invoke the Byzantine-tolerant classifier procedure within themselves. (B1) *The union of the safe value sets of all correct processes* for the slave group is a subset of the value set of a master process. (B2) The value set of any master process has size $> k$. (B3) *The union of the safe value sets of all correct processes* for the slave group has size $\leq k$.

Property (B1) guarantees that the value set of a slave process is always a subset of the value set of a master process starting from the point when they are classified into different subgroups. For property (B3), our algorithm maintains the following variant: a value is considered as valid by a correct process if it is in the safe value set of some correct process. Thus, property (B3) restricts what values a group of processes can ever send, which are also the values processes in the group can ever have (will be more clear in our algorithm).

To guarantee (B1) and (B2), similar to the $O(\log n)$ algorithm, we use the SetGradecast algorithm as a communication primitive and update the safe value sets of correct processes carefully. To guarantee (B3), dividing into the slave subgroup and the master subgroup in our algorithm is based on the safe value sets of processes.

From property (B1) and (B3), we can observe that the Byzantine-tolerant classifier procedure for a group of processes depends on not only processes in the group but also other processes in the system (via the safe value sets). Thus, in our presentation, we do not present the Byzantine-tolerant classifier procedure as a separate procedure, instead we present it inside the main algorithm and call it as a *classification step*.

In the $O(\log n)$ algorithm, we divide a group into slave subgroup and master subgroup based on process ids. A Byzantine process cannot lie about its group identity, i.e., whether it is in the slave group or the master group. In the $O(\log f)$ algorithm, the classification is based on the values received at each round. So, process i does not know whether process j is classified as a slave or a master at any given round. Thus, a Byzantine process can lie about its group identity and the $O(\log f)$ algorithm needs a mechanism to prevent such lies. In the $O(\log f)$ algorithm, each process $i \in [n]$ has a label l_i, which serves as the threshold when it performs the classification step and also indicates its group identity. We require that when a process sends a value, it needs to attach its label.

We formally define a *group* as a set of processes which have the same label. The *label of a group* is the label of the processes in this group. The label of a group is also the threshold value processes in this group use to do classification. We also use label to indicate a group. We say a process is in group k if its value is associated with label k. Initially all processes are within the same group with label $k_0 = n - \frac{f}{2}$.

Consider the classification step for group G with label k. There are two main differences between the $O(\log n)$ algorithm and the $O(\log f)$ algorithm. First, in the $O(\log n)$ algorithm, only processes in the slave group of G invoke the gradecast primitive to send its current value set. In the $O(\log f)$ algorithm, all processes in group G invoke the gradecast primitive. Second, in the $O(\log n)$ algorithm, the classification is based on process ids. In the $O(\log f)$ algorithm, we let each process send its safe value set for group G to all processes in G and each process in G performs classification based on the safe sets received. If the union of the received safe sets has size $> k$, the process is classified as a master, otherwise as a slave. Each slave process updates its value to be the set of values with score 2. Each master process updates its value to be the union of its current value and the set of values with score at least 1. The safe value set is updated in a similar manner to the $O(\log n)$ algorithm.

Algorithm 3 The $4 \log f + 3$ Rounds Algorithm for the BLA Problem.

x_i: input value for process i V_i: value set of process i. $V_i := \{x_i\}$ initially.
Initially all process are within the same group with label $k_0 = n - \frac{f}{2}$
l_i: label of p_i. $l_i := k_0 = n - \frac{f}{2}$ initially and is updated at each round
Map F_i, with $F_i[k]$ being the safe value set for processes with label k.

1: **for** each process i, **in parallel do** /* Build the Initial Safe Map and Value Set */
2: Execute **Gradecast**(i, x_i)
3: Let $< j, v_j, c_j >$ denote that process j gradecasts v_j with score c_j
4: Set $F_i[k_0] := \{v_j \mid c_j \geq 1 \wedge j \in [n]\}$ // safe set for the initial group with label k_0
5: Set $V_i := \{v_j \mid c_j = 2 \wedge j \in [n]\}$
6: **endfor**
7: **for** $r := 1$ to $\log f$
8: **for** each process i, **in parallel do**
9: Execute **SetGradecast**(i, V_i, l_i)
10: Let L denote the set of labels received in all Gradecasts
11: **for** each label $k \in L$ /* Each label represents a group */
/* Lines 12-21 is the classification step for group k */
12: Let $U^k_{i,1}$ denote the set of values with label k and assigned score ≥ 1 by p_i.
13: Let $U^k_{i,2}$ denote the set of values with label k and assigned score 2 by p_i
14: Set $F_i[m(k,r)] := F_i[k] \cup U^k_{i,1}$ and $F_i[s(k,r)] := U^k_{i,2}$
15: Send $U^k_{i,2}$ to processes who gradecast with label k.
16: **if** $l_i = k$ //if my label is k
17: Let R_j denote the set of values received from p_j at line 15
18: Set $T := \cup\{R_j \mid R_j \subseteq U^k_{i,1}, j \in [n]\}$
19: /* Classification */
20: **if** $|T| > k$ **then** $V_i := U^k_{i,1}$, $l_i := l_i + \frac{f}{2^{r+1}}$ //master process
21: **else** $V_i := U^k_{i,2}$, $l_i := l_i - \frac{f}{2^{r+1}}$ //slave process
22: **endfor**
23: **endfor**
24: **endfor**
25: $y_i := \sqcup\{v \in V_i^{\log f+1}\}$

In the $O(\log f)$ algorithm, each process $i \in [n]$ keeps track of a safe value map F_i with $F_i[k]$ being the safe value set of process i for group k, i.e., $F_i[k]$ is an upper bound on the values with label k that process i considers valid. Define $s(k,r) = k - \frac{f}{2^{r+1}}$ and $m(k,r) = k + \frac{f}{2^{r+1}}$. The algorithm is shown in Alg. 3.

In lines 1-6 of the algorithm, each process first invokes **Gradecast** to send its input value to all. Then, it constructs its value set as the set of values gradecast with score 2 and its initial safe value set for the initial group as the set of values gradecast with score at least 1. Lines 1-6 serve two purposes: 1) To construct the safe value set for the initial group with label $k_0 = n - \frac{f}{2}$ and ensure that each Byzantine process can introduce at most one value in the safe value sets 2) Ensure that there are at most f values unknown to each correct processes. Then, $\log f$ recursion levels suffice for all processes to obtain comparable values.

In lines 7-25, at each round, each process invokes **SetGradecast** to send its current value to all and performs classification. When a process invokes the **SetGradecast** to send its current value set, its current label is attached. After the gradecast step, for each process

$j \in [n]$, process i obtains a value-score-label triple from the gradecast of process j. Then, process i executes line 11-15 for each group at the current round. Specifically, for the group with label k, process i obtains the set of values with score at least 1, $U_{i,1}^k$, and the set of values with score 2, $U_{i,2}^k$, from the gradecast of processes with label k. Then, process i updates its safe value set for group $m(k,r)$, i.e., the master group of group k, to be the union of its safe value set for group k and $U_{i,1}^k$. It also updates its safe value set for group $s(k,r)$, i.e., the slave group, to be $U_{i,2}^k$. Due to property 2 of SetGradecast, we must have $U_{i,2}^k \subseteq U_{j,1}^k$ for any process i and j. This step is to ensure that if a master process j obtains a value in $U_{i,2}^k$, this value will be in the safe value set of each correct process for j. Then, when the master process tries to send this value using SetGradecast, it must be assigned score of 2 by each correct process, due to property 3 of SetGradecast. At line 15, process i sends its safe value set for the slave group to the processes in group k.

Lines 16-21 are only executed by processes in group k. They obtain the set of values sent by all processes at line 15 and do classification based the size of this set. To prevent a Byzantine process from sending arbitrary values at line 15, each process in group k only accepts the set R_j from process j if R_j is a subset of $U_{i,1}^k$. If process j is correct, this condition must hold by property 2 of SetGradecast. So the sets sent from correct processes will always be accepted. Then, the set T at line 18 must contain all the sets sent from correct processes. Since each such set is the safe value set of a process for the slave group, each process in group k is actually doing classification based on the safe sets of processes for the slave group. Lines 20-21 is the classification step. If the size of T is greater than k, then the process is classified as a master and updates its value to be the set of values gradecast by processes in group k with score at least 1. Otherwise, its value is updated to be the set of values gradecast by processes in group k with score 2. Its label is updated based on whether the process is a master or a slave.

Due to space limitation, we put the correctness proof and complexity analysis in our full paper [18]. Our main result is summarized in the following Theorem.

▶ **Theorem 16.** *There is a $4\log f + 3$ rounds algorithm for the BLA problem in synchronous systems which can tolerate $f < \frac{n}{3}$ Byzantine failures. It takes $O(n^2 \log f)$ messages.*

▶ **Corollary 17.** *There is a $5\log f + 4$ rounds algorithm for the authenticated BLA problem in synchronous systems which can tolerate $f < \frac{n}{2}$ Byzantine failures. It takes $O(n^2 \log f)$ messages.*

6 Conclusion

We have presented three algorithms for the Byzantine lattice agreement problem in synchronous systems. The first algorithm has early stopping property. The second algorithms take logarithmic rounds. The $O(\log f)$ rounds upper bound matches the bound for the crash failure setting. For future work, the following questions are interesting: 1) Can we improve the upper bound or prove some lower bound on the round complexity? 2) Can we solve the BLA problem in asynchronous systems in logarithmic rounds? We have some partial results in [19], but the algorithm proposed can only tolerate $f < \frac{n}{5}$ Byzantine failures.

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. Hagit Attiya, Maurice Herlihy, and Ophir Rachman. Atomic snapshots using lattice agreement. *Distributed Computing*, 8(3):121–132, 1995.
3. Hagit Attiya and Ophir Rachman. Atomic snapshots in o (n log n) operations. *SIAM Journal on Computing*, 27(2):319–340, 1998.
4. Michael Ben-Or, Danny Dolev, and Ezra N Hoch. Simple gradecast based algorithms. *arXiv preprint arXiv:1007.1049*, 2010.
5. Gabriel Bracha. Asynchronous Byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.
6. B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, Cambridge, UK, 1990.
7. Giuseppe Antonio Di Luna, Emmanuelle Anceaume, Silvia Bonomi, and Leonardo Querzoni. Synchronous byzantine lattice agreement in $\mathcal{O}(\log f)$ rounds. *arXiv preprint arXiv:2001.02670*, 2020.
8. Giuseppe Antonio Di Luna, Emmanuelle Anceaume, and Leonardo Querzoni. Byzantine generalized lattice agreement. *arXiv preprint arXiv:1910.05768*, 2019.
9. Jose M Faleiro, Sriram Rajamani, Kaushik Rajan, G Ramalingam, and Kapil Vaswani. Generalized lattice agreement. In *Proceedings of the 2012 ACM symposium on Principles of distributed computing*, pages 125–134. ACM, 2012.
10. Paul Feldman and Silvio Micali. Optimal algorithms for Byzantine agreement. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 148–161. ACM, 1988.
11. Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.
12. Jonathan Katz and Chiu-Yuen Koo. On expected constant-round protocols for byzantine agreement. In *Annual International Cryptology Conference*, pages 445–462. Springer, 2006.
13. Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.
14. Marios Mavronicolasa. A bound on the rounds to reach lattice agreement. *http://www.cs.ucy.ac.cy/ mavronic/pdf/lattice.pdf*, 2018.
15. Thomas Nowak and Joel Rybicki. Byzantine approximate agreement on graphs. In *33rd International Symposium on Distributed Computing (DISC 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.
16. Jan Skrzypczak, Florian Schintke, and Thorsten Schütt. Linearizable state machine replication of state-based crdts without logs. *arXiv preprint arXiv:1905.08733*, 2019.
17. TK Srikanth and Sam Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Computing*, 2(2):80–94, 1987.
18. Xiong Zheng and Vijay Garg. Byzantine lattice agreement in synchronous systems. *arXiv preprint arXiv:1910.14141*, 2019.
19. Xiong Zheng and Vijay Garg. Byzantine lattice agreement in asynchronous systems. *arXiv preprint arXiv:2002.06779*, 2020.
20. Xiong Zheng, Vijay K. Garg, and John Kaippallimalil. Linearizable Replicated State Machines With Lattice Agreement. In Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller, editors, *23rd International Conference on Principles of Distributed Systems (OPODIS 2019)*, volume 153 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 29:1–29:16, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
21. Xiong Zheng, Changyong Hu, and Vijay K Garg. Lattice agreement in message passing systems. In *32nd International Symposium on Distributed Computing (DISC 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

Fast Distributed Algorithms for Girth, Cycles and Small Subgraphs

Keren Censor-Hillel
Technion – Israel Institute of Technology, Haifa, Israel
ckeren@cs.technion.ac.il

Orr Fischer
Tel-Aviv University, Israel
orrfischer@mail.tau.ac.il

Tzlil Gonen
Tel-Aviv University, Israel
tzlilgon@gmail.com

François Le Gall
Nagoya University, Japan
legall@math.nagoya-u.ac.jp

Dean Leitersdorf
Technion – Israel Institute of Technology, Haifa, Israel
dean.leitersdorf@gmail.com

Rotem Oshman
Tel-Aviv University, Israel
roshman@tau.ac.il

Abstract

In this paper we give fast distributed graph algorithms for detecting and listing small subgraphs, and for computing or approximating the girth. Our algorithms improve upon the state of the art by polynomial factors, and for girth, we obtain a constant-time algorithm for additive +1 approximation in Congested Clique, and the first parametrized algorithm for exact computation in Congest.

In the Congested Clique model, we first develop a technique for learning small neighborhoods, and apply it to obtain an $O(1)$-round algorithm that computes the girth with only an additive +1 error. Next, we introduce a new technique (the partition tree technique) allowing for efficiently listing all copies of any subgraph, which is deterministic and improves upon the state-of-the-art for non-dense graphs. We give two concrete applications of the partition tree technique: First we show that for constant k, it is possible to solve C_{2k}-detection in $O(1)$ rounds in the Congested Clique, improving on prior work, which used fast matrix multiplication and thus had polynomial round complexity. Second, we show that in triangle-free graphs, the girth can be exactly computed in time polynomially faster than the best known bounds for general graphs. We remark that no analogous result is currently known for sequential algorithms.

In the Congest model, we describe a new approach for finding cycles, and instantiate it in two ways: first, we show a fast parametrized algorithm for girth with round complexity $\tilde{O}(\min\{g \cdot n^{1-1/\Theta(g)}, n\})$ for any girth g; and second, we show how to find small even-length cycles C_{2k} for $k = 3, 4, 5$ in $O(n^{1-1/k})$ rounds. This is a polynomial improvement upon the previous running times; for example, our C_6-detection algorithm runs in $O(n^{2/3})$ rounds, compared to $O(n^{3/4})$ in prior work. Finally, using our improved C_6-freeness algorithm, and the barrier on proving lower bounds on triangle-freeness of Eden et al., we show that improving the current $\tilde{\Omega}(\sqrt{n})$ lower bound for C_6-freeness of Korhonen et al. by *any* polynomial factor would imply strong circuit complexity lower bounds.

2012 ACM Subject Classification Networks → Network algorithms; Theory of computation → Distributed algorithms

Keywords and phrases distributed graph algorithms, cycles, girth, Congested Clique, CONGEST

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.33

Funding This project was partially supported by the European Union's Horizon 2020 Research and Innovation Programme under grant agreement no. 755839, by the JSPS KAKENHI grants JP16H01705, JP19H04066 JP20H04139 and JP20H00579 and by the MEXT Q-LEAP grant JP-MXS0120319794.

© Keren Censor-Hillel, Orr Fischer, Tzlil Gonen, François Le Gall, Dean Leitersdorf, and Rotem Oshman;
licensed under Creative Commons License CC-BY
34th International Symposium on Distributed Computing (DISC 2020).
Editor: Hagit Attiya; Article No. 33; pp. 33:1–33:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

A fundamental problem in many computational settings is that of finding cycles and other small subgraphs within a given graph. This paper focuses on finding subgraphs in distributed networks that communicate through limited bandwidth. The motivation for this is two-fold: first, for some subgraphs H there exist distributed algorithms that perform better on H-free graphs, such as distributed cut and coloring algorithms in triangle-free graphs [15, 25]. The second reason for which we are interested in these problems is that while solving them only requires obtaining *local* knowledge, about small non-distant neighborhoods, the bandwidth restrictions impose a major hurdle for collecting this information. This induces a rich landscape of complexities for subgraph-related problems. We contribute to the effort of characterizing the complexities of subgraph-related problems by providing new techniques, from which we derive fast algorithms for such problems in the two key distributed bandwidth restricted models, namely, Congest and Congested Clique.

In the Congested Clique model, n synchronous nodes can send messages of $O(\log n)$ bits in an all-to-all fashion. The input graph is an arbitrary n vertex graph, partitioned such that every node receives the edges of a single vertex as input. Our main contribution in this model is an algorithm for obtaining a $+1$ approximation for the girth in a *constant* number of rounds, where the girth of a graph is the length of its shortest cycle.

▶ **Theorem 1.** *Given a graph G with an unknown girth g, there exists a deterministic $O(1)$ round algorithm in the Congested Clique model which outputs an integer a, such that $g \in \{a, a+1\}$.*

For comparison, note that the current state-of-the-art algorithm computes the exact girth in $O(n^{0.158})$ rounds [4]. To obtain our $+1$ approximation algorithm, we devise two main new methods, which we describe here in a nutshell. The first is an algorithm in which each node learns its entire neighborhood up to a radius which is a constant approximation of the girth. To this end, we prove that we can quickly list all paths of sufficient length, as well as efficiently distribute them to the nodes that need to learn them. The second method that we introduce is a way to double the radius of the neighborhoods that all the nodes know, by having each node acquire the knowledge held by the farthest nodes in its currently-known neighborhood. Crucially, both of these procedures can be done in $O(1)$ rounds, and could be useful for additional applications.

Our second contribution in the Congested Clique model is a *partition tree* technique which allows for efficiently detecting or listing all copies of any subgraph with at most $\log n$ nodes, in a deterministic manner. In particular, our main application of the partition tree technique is to obtain the following subgraph listing algorithm, which improves upon the state-of-the-art for non-dense graphs.

▶ **Theorem 2.** *Given a graph G with n nodes and m edges and a graph H with $p \leq \log n$ nodes and k edges, let $\tilde{m} = \max\{m, n^{1+1/p}\}$. There exists a deterministic Congested Clique algorithm that terminates in $O(\frac{k\tilde{m}}{n^{1+2/p}} + p)$ rounds and lists all instances of H in G.*

We give two concrete applications of this result. The first is fast detection of even cycles.

▶ **Corollary 3.** *Given a graph G and an integer $k \leq (\log n)/2$, there exists a deterministic $O(k^2)$-round algorithm in the Congested Clique model for detecting cycles of length $2k$.*

Note that for constant k the above algorithm completes within $O(1)$ rounds. Prior work for cycle detection in the Congested Clique model used fast matrix multiplication (FMM) and thus had polynomial round complexity, apart from detecting 4-cycles which was shown to

have a constant-round algorithm [4]. The second implication of the partition-tree technique is a fast algorithm for computing the *exact* girth in triangle-free graphs. Prior algorithms for girth in the CONGESTED CLIQUE model are based on fast matrix multiplication (FMM), a technique that can be no faster than checking for triangle-freeness.

▶ **Corollary 4.** *Given a triangle-free graph G with an unknown girth g, there exists a deterministic $\tilde{O}(n^{1/10})$-round algorithm in the CONGESTED CLIQUE model which outputs g.*

This result leverages the fact that graphs without small cycles become increasingly sparse, and the algorithm of Theorem 2 is efficient on sparse graphs. We remark that, interestingly, no analogous result going below the complexity of FMM for girth in triangle-free graphs is known for sequential algorithms, since the best known sequential algorithms for cycle detection in sparse graphs (see [2]) are not fast enough. We also note that given further lower bounds on the girth beyond triangle-freeness, the runtime of our algorithm improves even further; for instance, if the graph does not contain any k-cycle for $k \in \{3, 4, 5\}$, then our algorithm computes the exact girth in $\tilde{O}(n^{1/21})$ rounds. We refer to Proposition 13 in Section 4 for a more precise statement.

In the CONGEST model, n synchronous nodes can send messages of $O(\log n)$ bits to their neighbors only, and the input graph is the communication graph. In this model, we develop a new approach for finding cycles of a given size. A key step that is present in all known sublinear-round algorithms for finding cycles in CONGEST is the elimination of *high-degree vertices*: we check whether there is a cycle that includes a high-degree node, and if we conclude that there is no such cycle, we can remove the high-degree nodes from the graph. The remaining graph is much easier to handle, since it has low degree. In prior work, the high-degree vertices were eliminated by sequentially enumerating over them and starting a short BFS from each one. Here we introduce a different method for finding cycles that include a high-degree node: intuitively, we show that if we start from a *neighbor* of a small even cycle, we can quickly find the cycle itself. Since high-degree nodes have many neighbors, if we sample a uniformly random node in the graph, we are somewhat likely to hit a neighbor of the high-degree node, and from there we can find the cycle in constant rounds.

We apply this technique to give a fast algorithm that detects small even cycles, and a fast parameterized algorithm for computing the *exact* girth. Specifically, we obtain the following:

▶ **Theorem 5.** *Given a graph G, there exists a randomized algorithm in the CONGEST model for detection of $2k$-cycles in $O(n^{1-1/k})$ rounds, for $k = 3, 4, 5$.*

This significantly improves upon the running time of $O(n^{1-1/\Theta(k^2)})$ of the previous state-of-the-art [10]: for cycles of length 6,8 or 10, the previous algorithm had running time $O(n^{3/4})$, $O(n^{5/6})$ or $O(n^{10/11})$, respectively. We believe that going below round complexity of $O(n^{1-1/k})$ for C_{2k}-detection in the CONGEST model would require a breakthrough beyond currently known techniques, with potential ramifications also for the CONGESTED CLIQUE model.

For exact girth, previously, an $O(n)$-round algorithm for exact girth was known, based on computing all-pairs shortest paths [16]. Our result is as follows:

▶ **Theorem 6.** *Given a graph G with an unknown girth g, there exists a randomized $O(\min\{g \cdot n^{1-1/\Theta(g)}, n\})$-round algorithm in the CONGEST model which outputs g.*

"Outputs" here means that the first node that halts outputs the girth. Other nodes of the graph may halt later, and output larger values. This is unavoidable, unless we introduce a term in the running time that depends on the diameter of the graph.

Our final result is an *obstacle* on proving lower bounds for C_6-freeness in CONGEST. In [19] it was shown that the C_{2k}-freeness problem is subject to a lower bound of $\widetilde{\Omega}(\sqrt{n})$, for any k. For C_6-freeness, the best known algorithm is our new algorithm here, which runs in $\widetilde{O}(n^{2/3})$ rounds, and there are reasons to believe that this may be optimal. Unfortunately, we show that proving a lower bound of the form $\Omega(n^{1/2+\alpha})$, for any constant $\alpha > 0$, would imply breakthrough results in circuit complexity. This result uses ideas from our improved C_6-freeness algorithm, and the barrier on proving lower bounds on triangle-freeness from [10, 11].

Related work. The problem of subgraph-freeness, and in particular cycle detection, has been extensively studied in the CONGESTED CLIQUE and CONGEST models. While there are only a few papers which study girth computation, related problems such as diameter computation or shortest paths were also extensively studied in these models.

In the first work to consider girth computation in the sequential setting, Itai and Rodeh [17] gave algorithms with running time $O(mn)$ and $O(n^2)$ for computing exact girth and $+1$ approximation of the girth, respectively, using a BFS approach, and an $O(n^\omega)$ algorithm for exact girth using an algebric method, where ω is the exponent of matrix multiplication. Later, various trade-offs between running time and additive or multiplicative approximations for girth were obtained (e.g [22, 26–28]).

In the CONGESTED CLIQUE model, an $O(n^{0.158})$ round algorithm for exact girth and a $2^{O(k)}n^{0.158}$ round algorithm for C_k-detection for any k was shown by [4] based on matrix multiplication techniques. These algebraic techniques were later extended by [3, 5, 20].

For general subgraphs, a CONGESTED CLIQUE algorithm for listing all instances of a subgraph H of size k in $\widetilde{O}(n^{1-2/k})$ rounds was shown in [8], which was shown to be tight for triangles [18, 23] and later for cliques of size $k > 3$ as well [12]. In this work we give a "sparsity-aware" version of this result, which has improved performance as the graph becomes sparser. Previously, distributed "sparsity-aware" algorithm were studied in the context of distributed sparse matrix multiplication [3, 5] and in the context of the k-machine model [23].

Frischknecht et al. [13] was the first work to consider girth computation in CONGEST, and showed that at least $\widetilde{\Omega}(\sqrt{n})$ rounds are required in order to obtain a $(2-\epsilon)$-approximation of the girth. Peleg et al. [24] showed an algorithm computing a $(2 - 1/g)$-approximation of the girth with round complexity $O(D + \sqrt{gn}\log n)$, where g is the girth of the graph. Holzer et al. [16] showed an algorithm for exact girth computation in $O(n)$ rounds, based on an exact all-pairs shortest path algorithm, and an algorithm for computing an $(1 + \epsilon)$-approximation of the girth in $O(\min\{n/g + D\log(D/g), n\})$ rounds.

In the cycle-freeness problem in the CONGEST setting, Drucker et al. [9] showed a near tight lower bound of $\widetilde{\Omega}(n)$ for constant sized odd-length cycles, as well as a lower bound of $\widetilde{\Omega}(n^{1/k})$ for C_{2k}-freeness, which was later improved to $\widetilde{\Omega}(\sqrt{n})$ by Korhonen et al. [19]. A CONGEST randomized algorithm for listing all triangles with round complexity $\widetilde{O}(n^{1/3})$ was shown in [7], which improved the previous $\widetilde{O}(n^{1/2})$-round algorithm of [6] and the $\widetilde{O}(n^{3/4})$-round algorithm of [18]. The first sublinear-time algorithm for C_{2k}-freeness for $k \geq 3$ was given in [12] running in $\widetilde{O}(n^{1-1/(k^2-k)})$ rounds, and was later improved in [10] to round complexity $\widetilde{O}(n^{1-2/(k^2-k+2)})$ for odd k and $\widetilde{O}(n^{1-2/(k^2-2k+4)})$ for even k.

2 Preliminaries

Definitions. Given a graph H, the H-*listing* problem is a problem in which each node may output a set of H-copies, and the goal of the network is that w.h.p. the union over the sets of outputted H-copies by the nodes is exactly the set of H-copies in G.

The *Túran number* of a graph H, denoted $\text{ex}(n, H)$, is the maximum number of edges m such that there exists a graph G on n vertices and m edges which contains no copy of H.

▶ **Lemma 7** (Túran number of C_{2k} [14]). *For $k \in \mathbb{N}$, if G is C_{2k}-free, then G contains at most $17kn^{1+1/k}$ edges.*

▶ **Lemma 8** (Túran number for girth [14]). *If G is C_i-free for all $3 \leq i \leq 2k$, then G contains at most $n^{1+1/k} + n$ edges.*

Let $N_i(v)$ denote the *graph* defined by the nodes of hop-distance at most i from v, that is, it includes all such nodes and all the *edges* incident to nodes with hop-distance at most $i-1$ from v. For sets $A, B \subseteq V$, denote by $E(A, B) = \{(a, b) \in E \mid a \in A \wedge b \in B\}$ the set of edges between A and B.

Load-Balanced Routing in the Congested Clique Model. We introduce a useful routing procedure which extends that of [21], and it is used throughout our results. The routing procedure of [21] routes a set of messages where each node needs to send and receive at most $O(n)$ messages, in $O(1)$ rounds. The following shows that it possible to replace the constraint where each node needs to *send* at most $O(n)$ messages with one stating that the messages each node desires to send are based on at most $O(n \log n)$ bits. This allows us to route messages even when the some nodes are each a source of $\omega(n)$ messages.

▶ **Lemma 9** (Load Balanced Routing). *Any routing instance $\mathcal{M}$, in which every node v is the target of up to $O(n)$ messages, and v locally computes the messages it desires to send from at most $|R(v)| = O(n \log n)$ bits, can performed in $O(1)$ rounds.*

The proof of Lemma 9 is deferred to the full paper. Notice that this lemma implies, in a straightforward manner, the following Corollary 10 which we refer to extensively.

▶ **Corollary 10.** *In the deterministic* Congested Clique *model, given that each node originally begins with $O(n \log n)$ bits of input, and at most $O(1)$ rounds have passed since the initiation of the algorithm, then any routing instance $\mathcal{M}$, in which every node v is the target of up to $O(n \cdot x)$ messages, can performed in $O(x)$ rounds.*

While this is a weaker statement than that of Lemma 9, it is convenient to use when showing constant-time algorithms in the Congested Clique model, as it completely circumvents the need for a bound on the number of messages each node desires to send.

3 Deterministic $O(1)$ Round Algorithm for $+1$ Girth Approximation in the Congested Clique

In this section we prove Theorem 1: we construct a deterministic $O(1)$ round algorithm for $+1$ girth approximation in the Congested Clique model. The algorithm is composed of two phases, each a novel technique on its own, and through their combination, we achieve the desired result. The first procedure is based on a *subgraph enumeration* approach and allows each node to learn its $\lfloor \frac{g}{20} \rfloor$ hop-neighborhood in $O(1)$ rounds. Formally, it is shown in the following Theorem 11.

▶ **Theorem 11.** *Given a graph G, with n nodes, and an unknown girth $g < \log n$, there exists an $O(1)$ round algorithm in the* Congested Clique *model, which either outputs g, or, ensures that every node knows its $\lfloor g/20 \rfloor$ hop-neighborhood.*

The latter procedure is based on a *BFS-like* approach and allows each node to double the hop-distance of the neighborhood which it knows in $O(1)$ rounds, *at least* until the first cycle is encountered. This is stated formally in Theorem 12

▶ **Theorem 12.** *Let G be a graph with n nodes, an unknown girth $g < \log n$, and with minimum degree $\delta \geq 2$. Assume that for a given integer parameter $a > 0$, every $v \in V$ knows the edges of $N_a(v)$, and that $N_a(v)$ is a tree. There exists an algorithm which completes in $O(1)$ rounds of the* Congested Clique *model, and either reports g exactly, or, reaches one of the following two states: (1) Every $v \in V$ knows $N_{2a}(v)$,* or *(2) For some value $b \geq \lceil \frac{g}{2} \rceil - 1$ which is agreed upon by all nodes of the network, every $v \in V$ knows all of $N_b(v)$. All nodes know whether g was reported exactly, and if not, which state was reached.*

Thus, by invoking the first algorithm once, and then the latter for a constant number of times, we achieve an $O(1)$ round algorithm for the approximation problem. The reason we achieve a $+1$ approximation, and not an exact result, is due to the fact that the second algorithm stops right before detecting the shortest cycle in the graph and cannot differentiate whether it is of odd or even length. In Section 3.1, we formally prove Theorem 1, when $g < \log n$ and the minimum degree is $\delta \geq 2$, using Theorems 11, 12. The constraints $g < \log n$ and $\delta \geq 2$ can be quickly overcome by eliminating some trivial, degenerate cases, and these details are deferred to the full version. Finally, in Sections 3.2, 3.3, we proceed to our fundamental technical contributions by showing the proofs of Theorems 11, 12.

3.1 Proving Theorem 1

Here, we prove Theorem 1, in case that $g < \log n$ and the minimum degree is $\delta \geq 2$.

Proof of Theorem 1. We first invoke the algorithm from Theorem 11, in $O(1)$ rounds. Either g was outputted, or, every node learned its $\lfloor g/20 \rfloor$ hop-neighborhood.

Next, we invoke the algorithm from Theorem 12. The nodes now learned new, larger neighborhoods - regardless of whether the algorithm halted in State 1 (every $v \in V$ knows $N_{2a}(v)$) or State 2 (for some $b \geq \lceil \frac{g}{2} \rceil - 1$, every $v \in V$ knows all of $N_b(v)$). If any node sees a cycle, then it broadcasts the length of the shortest cycle which it sees and all the nodes terminate and output the minimum of the values which were broadcast in the network. It is clear that, in this case, the exact value of g is outputted, since all nodes know the neighborhoods surrounding them of same radius, and thus if any node saw a cycle, one node must have seen the shortest cycle in the graph.

Finally, in the case that no cycle was seen so far, we differentiate between the states at which the algorithm from Theorem 12 can halt at. If it halts at State 1, then every node learned twice the radius of the neighborhood it already knew. In such a case, we invoke Theorem 12 again and repeat. Notice that we can do this at most $O(1)$ times, before either seeing a cycle or halting at State 2, due to the fact that the nodes originally know their $\lfloor g/20 \rfloor$ hop-neighborhoods. In the case that we eventually halt at State 2, and no cycles were seen by any node so far, all the nodes output that the girth is either $2b + 1$ or $2b + 2$, where b is the radius of the neighborhoods which they learned. It is clear that if all nodes learned their $b \geq \lceil \frac{g}{2} \rceil - 1$ hop-neighborhoods, and none saw a cycle, then it must be that $b = \lceil \frac{g}{2} \rceil - 1$ and thus either $g = 2b + 1$ or $g = 2b + 2$. ◀

3.2 Phase I: Initial Neighborhood Learning

The key procedure of this phase (formally stated above as Theorem 11) consists of two major steps. In the first step, either each path of length $\lfloor \frac{g}{10} \rfloor$ in G is detected by at least one node,

or g is outputted. This step can be seen as an edge-partition variant of the listing algorithm in [8]. The second step uses a load-balancing routine in order to redistribute the information computed in the first step so that each node v learn its $\lfloor \frac{g}{20} \rfloor$ hop-neighborhood.

Step 1: Path Listing. We next list all paths of length $\lfloor \frac{g}{10} \rfloor$, or output g, in $O(1)$ rounds.

First, each node sends its degree to the rest of the network, and each then locally calculates the number of edges in the graph $m = \sum_v deg(v)/2$. Let $k \in \mathbb{N}$ be the largest integer such that $m \leq n^{1+1/k} + n$. Then, each node v is assigned a hard-coded range of $deg(v)$ indices in $\{1, \ldots, m\}$, and locally numbers its edges using these indices.

If $k \leq 4$, then by Lemma 8 the girth is of size at most 10, and thus, trivially, paths of length $\lfloor \frac{g}{10} \rfloor \leq 1$ are known and we can halt. Thus, from here on, we may assume that $k \geq 5$.

Let P be a partition of the set $\{1, \ldots, m\}$ into $\lceil kn^{2/k}/(20e) \rceil$ consecutive segments of size $O\left(\frac{m}{\lceil kn^{2/k}/(20e) \rceil} + 1\right)$, and let K be a family containing all the possible choices of $\lfloor k/4 \rfloor$ segments from P (in this context, e denotes the mathematical constant). It holds that

$$|K| = \binom{\lceil kn^{2/k}/(20e) \rceil}{\lfloor k/4 \rfloor} \leq \left(\frac{e\lceil kn^{2/k}/(20e) \rceil}{\lfloor k/4 \rfloor}\right)^{\lfloor k/4 \rfloor} \leq \left(\frac{n^{2/k}}{2} + 1\right)^{\lfloor k/4 \rfloor} \leq n^{\frac{2}{k}\lfloor k/4 \rfloor} \leq n,$$

where the first inequality holds due to the well-known combinatorial statement that $\binom{n}{k} \leq (\frac{ne}{k})^k$, for all $n \in \mathbb{N}, 1 \leq k \leq n$, and in the other inequalities, the fact that $5 \leq k < \log n$ is used. Thus, it is possible to associate each $k_i \in K$ with a unique node v_i. Each k_i is a set of $\lfloor k/4 \rfloor$ sets of $O\left(\frac{m}{\lceil kn^{2/k}/(20e) \rceil} + 1\right)$ edges, and so let E_i denote the edges in the sets contained in k_i. Notice that

$$|E_i| \leq \lfloor k/4 \rfloor \left(\frac{m}{\lceil kn^{2/k}/(20e) \rceil} + 1\right) \leq (k/4)\frac{20en^{1+1/k}}{kn^{2/k}} + k = 5en^{1-1/k} + k \leq n,$$

and therefore, by Corollary 10, it is possible for each v_i to learn all of the edges in E_i in $O(1)$ rounds of communication.

Finally, every node v_i broadcasts the shortest cycle which it witnesses in E_i. Notice that every path, p, of length at most $\lfloor k/4 \rfloor$, is fully contained inside some E_j, due to the construction of P, and therefore the corresponding node, v_j, which now knows all of E_j, will witness p. Thus, if $g \leq \lfloor k/4 \rfloor$, some node will witness the shortest cycle in the graph and be able to broadcast its length, g. Otherwise, notice that since $m \not\leq n^{1+1/(k+1)} + n$, Lemma 8 implies that the graph is not C_i-free for all $i \leq 2(k+1)$, and thus $g \leq 2(k+1)$. Thus all paths of length at most $\lfloor k/4 \rfloor \geq \lfloor g/8 - 1/4 \rfloor \geq \lfloor g/10 \rfloor$ have been listed. Notice that $g/8 - 1/4 \geq g/10$ whenever $g \geq 10$, and this can be assumed, since otherwise, trivially, paths of length $\lfloor \frac{g}{10} \rfloor < 1$ are known.

If at least one node v_i informs about a cycle in E_i, the minimum number sent by a node is outputted as g, and the algorithm terminates. Otherwise, it proceeds to the second step.

Step 2: Neighborhood Learning. We desire to redistribute some of the information learned in the previous step so that each node will know its $\lfloor \frac{g}{20} \rfloor$ hop-neighborhood.

Notice that all paths of length at most $\lfloor g/10 \rfloor$ have been listed. Therefore, also all paths of length at most $\lfloor \frac{g}{20} \rfloor$ have been listed. We strive to redistribute this information so that each node v knows all paths of length at most $\lfloor \frac{g}{20} \rfloor$ which start at v, and thus v knows its entire $\lfloor \frac{g}{20} \rfloor$ hop-neighborhood. Notice that we would like for each v to know both the nodes in its $\lfloor \frac{g}{20} \rfloor$ hop-neighborhood, as well as the edges between them.

We begin by ensuring that each v knows every node u in its $\lfloor \frac{g}{20} \rfloor$ hop-neighborhood. Let $v \in V$ and u be some node in its $\lfloor \frac{g}{20} \rfloor$ hop-neighborhood. Since $\lfloor \frac{g}{20} \rfloor < g/2$, then the $\lfloor \frac{g}{20} \rfloor$ hop-neighborhood of v is a tree. Therefore, there exists exactly one path, $p_{v,u}$, of length

at most $\lfloor \frac{g}{20} \rfloor$ between v and u. In the previous step, we ensured that at least one node w is aware of $p_{v,u}$. Specifically, notice that it might be the case that many nodes know about $p_{v,u}$, due to the last step, yet, every node w which knows of this path also knows all the other nodes w' which learned this path through their $E_{w'}$. Thus, it is possible to choose, in a hard-coded manner, a single node w which will be responsible for informing v that $p_{v,u}$ exists. Having done that, node w desires to convey to v the message that u is in its $\lfloor \frac{g}{20} \rfloor$ hop-neighborhood, in addition to the hop-distance between v and u – that is, the length of $p_{v,u}$. Notice that for each such u, node v is destined to receive exactly one message, and therefore every node in the graph is the target of $O(n)$ messages. This shows that Corollary 10 may be invoked in order to deliver all these messages in $O(1)$ rounds.

Now, we desire to inform every v of the edges in its $\lfloor \frac{g}{20} \rfloor$ hop-neighborhood. Node v now knows all the nodes u in this neighborhood, as well as the hop-distance to each of them. Node v sends a message to each such u which is at most $\lfloor \frac{g}{20} \rfloor - 1$ hops away from it, and requests that u send to v *all* its incident edges in the graph. Notice that all these edges are exactly all the edges contained in the $\lfloor \frac{g}{20} \rfloor$ hop-neighborhood of v, and since this neighborhood is a tree, v is the target of at most $O(n)$ messages. As before, this shows that Corollary 10 may be invoked in order to deliver all these messages in $O(1)$ rounds.

3.3 Phase II: Neighborhood Doubling

The key procedure in this phase (formally stated above as Theorem 12) is an $O(1)$ round algorithm which doubles the radius of the hop-neighborhood known to each node, until the nodes know a neighborhood large enough in order to approximate the girth up to an additive value of 1. The algorithm works along the following lines. Denote by $F_a(v)$, the nodes which are exactly at distance a from v – we refer to these as the *front-line* nodes. Each nodes v initially knows $N_a(v)$, and at once attempt to learn all of $\bigcup_{u \in F_a(v)}(N_a(u) \setminus N_a(v))$, in an efficient manner. If this step succeeds, then all the nodes reach State 1, and halt. Otherwise, they coordinate to increase the radii of the neighborhoods which they know by as much as possible in $O(1)$ rounds, and ultimately arrive at State 2, and halt.

Halting at State 1. Let $v \in V$ and $u \in F_a(v)$. Node u aims to send to node v the edges in $N_a(u) \setminus N_a(v)$. Notice that node u can locally compute these edges as follows. It observes the first node w on the path between v, u. Since $N_a(u)$ is a tree, for every node $w' \in N_a(u)$, there is exactly one simple path, $p_{u,w'}$, which u sees to w'. Notice that $w' \in N_a(v)$ if and only if $p_{u,w'}$ passes through w. Thus, node u knows exactly which edges it desires to send to node v. However, before sending them, it first sends to node v the value $|N_a(u) \setminus N_a(v)|$.

We now shift back to the perspective of node v. It computes and broadcasts an upper bound on $|\bigcup_{u \in F_a(v)}(N_a(u) \setminus N_a(v))|$, by calculating $\sum_{u \in F_a(v)} |N_a(u) \setminus N_a(v)|$. If all nodes broadcast values which are at most $n-1$, then by Corollary 10, it is possible in $O(1)$ rounds to perform all the routing requests and have each node double the radius of the neighborhood which it knows. At this point, the nodes collectively reach State 1 and halt.

Otherwise, at least one node reported a value greater than or equal to n. This implies that for some node v, there is a cycle in $N_{2a}(v)$, since at least two nodes $u, u' \in F_a(v)$ have simple paths in their a hop-neighborhoods to the same node w. In this case, the nodes proceed to a second part of the algorithm, which eventually leads to halting at State 2.

Halting at State 2. Our goal, at this stage, is to determine the largest possible value $i' \in \{1, \ldots, a-1\}$, such that for every node v, $\sum_{u \in F_a(v)} |N_{i'}(u) \setminus N_a(v)| < n$. Once this is achieved, then the algorithm can complete in a similar manner to that above. To see this,

assume that we have this maximal value i'. Therefore, all nodes v can learn $N_{a+i'}(v)$ in $O(1)$ rounds, similarly to above. If any cycle is seen, then g is outputted and the algorithm halts. Otherwise, due to the definition of i', there must exist some node v' such that $\sum_{u \in F_a(v')} |N_{i'+1}(u) \setminus N_a(v')| \geq n$. This implies that there is a cycle in $N_{a+i'+1}(v')$, and therefore $2a + 2i' < g \leq 2a + 2i' + 2$. As such, $a + i' = (2a' + 2i' + 2)/2 - 1 \geq \lceil g/2 \rceil - 1$, and we may halt at State 2.

We now show how to find i'. This is trivially possible to accomplish in $O(a)$ rounds – each node u simply sends to v the values $\{|N_1(u) \setminus N_a(v)|, \ldots, |N_{a-1}(u) \setminus N_a(v)|\}$, node v locally computes the $a - 1$ different sums, and broadcasts them. However, this does not suffice for our goal of an $O(1)$ round algorithm, as a can be logarithmic in n. Instead, let every node v broadcast $|F_a(v)|$, and denote by v' the node with maximal $|F_a(v')|$, and write $d = \lfloor n/|F_a(v')| \rfloor$. For every node v and $u \in F_a(v)$, node u sends to v the values $\{|N_1(u) \setminus N_a(v)|, \ldots, |N_d(u) \setminus N_a(v)|\}$, node v computes the d different sums of these values from all $u \in F_a(v)$, and broadcast them. Notice that this takes $O(1)$ rounds, using Corollary 10 as each node wants to receive at most $O(n)$ messages. Notice that it is now possible in $O(1)$ rounds to compute $\min\{i', d\}$ – either $i' \leq d$, and thus $\min\{i', d\} = i'$ and we can compute it, or, $\min\{i', d\} = d$. If we show that $g \leq 2a + 2d$, then if all v learn $N_{a+\min\{i',d\}}(v)$, this would suffice in order to either find the exact girth or halt at State 2, as required.

We claim that $g \leq 2a + 2d$. To see this, assume that $g > 2a + 2d$. Since $g > 2a + 2d$, there are no cycles in $N_{a+d}(v')$. Combining this with the fact that we assume the minimal degree in G to be at least 2, we can see that for all $j \neq j' \in \{1, \ldots, d\}$, it holds that $|F_{a+j}(v')| \geq |F_{a+j-1}(v')|$, and $F_{a+j}(v') \cap F_{a+j'}(v') = \emptyset$. Thus, in $N_{a+d}(v')$ there are at least $(d + 1) \cdot |F_a(v')| = (\lfloor n/|F_a(v')| \rfloor + 1) \cdot |F_a(v')| > n$ nodes, a clear impossibility. As we have arrived at a contradiction, we get that $g \leq 2a + 2d$, as required.

4 Subgraph Listing in the Congested Clique Model

We show an efficient "sparsity-aware" algorithm to *list* subgraphs in the Congested Clique model. Our main result is the following theorem, which is proven in the following sections.

▶ **Theorem 2.** *Given a graph G with n nodes and m edges and a graph H with $p \leq \log n$ nodes and k edges, let $\tilde{m} = \max\{m, n^{1+1/p}\}$. There exists a deterministic Congested Clique algorithm that terminates in $O(\frac{k\tilde{m}}{n^{1+2/p}} + p)$ rounds and lists all instances of H in G.*

As mentioned in the introduction, we can combine this result with known bounds on the number of edges in graphs without specific subgraphs, to achieve fast subgraph *detection* results. First, by combining Theorem 2 with Lemma 7, we immediately get Corollary 3: If the graph contains more than $17kn^{1+1/k}$ edges (which can be checked in a single round), then by Lemma 7 we can safely output that there must exist a cycle of length $2k$. Otherwise, plugging $p = 2k$ and $m = 17kn^{1+1/k}$ in Theorem 2 gives that we can detect (and even list, in this case) the existence of a cycle of length $2k$ within $O(k^2)$ rounds. Next, by combining Theorem 2 with Lemma 8, we can get the following result:

▶ **Proposition 13.** *Given a graph G with n nodes, m edges and an unknown girth g such that $g > \ell$ for some known ℓ, and defining $f(x) = n^{1/\lfloor (x-1)/2 \rfloor - 2/(2 \cdot \lfloor (x-1)/2 \rfloor + 1)}$, there is a deterministic $\tilde{O}(\min\{f(g), f(2 \cdot \lfloor (\ell+1)/2 \rfloor + 1)\})$ round algorithm in the Congested Clique model which outputs g.*

Proposition 13 first shows that the exact girth can be computed in $\tilde{O}(f(g))$ rounds – a polynomial improvement over the state-of-the-art for all graphs with $g \geq 5$. Moreover, if it is

known that the graph has girth greater than ℓ,[1] then the round complexity is additionally guaranteed to be $\tilde{O}(f(2 \cdot \lfloor(\ell+1)/2\rfloor + 1))$. For instance, for any odd value $\ell = 2r-1$ we get the upper bound $\tilde{O}(n^{1/r-2/(2r+1)})$. Taking $r = 2$ gives Corollary 4 stated in the introduction, which improves upon the state-of-the-art for triangle free graphs.

We note that more claims can be shown using bounds for the Túran numbers of various other graphs – for example, for detection of $K_{s,t}$ (complete bipartite graph with s nodes on one side and t on the other) for certain values of s, t.

4.1 Partition trees

We introduce the notion of *partition trees*, as a fundamental tool for subgraph listing in the *deterministic* CONGESTED CLIQUE model. Partition trees are a deterministic load-balancing mechanism that evenly divides the work of checking whether any copies of a subgraph are present. In prior work, randomized load-balancing was used for this purpose, but this incurs logarithmic factors which we cannot tolerate here. Throughout this section, given a subgraph with p nodes, we frequently refer to the value $x = n^{1/p}$. We assume that x is an integer, because $p \le \log n$ implies $x \ge 2$, and so it is possible to round x to an integer without affecting the round complexity or correctness.

We start with Definition 14, which defines a p-partition tree, which is a tree structure in which every node represents a partition of the graph G. Then, in Definition 15, we define an H-partition tree, in which we require certain conditions on the number of edges between parts of a p-partition, based on the subgraph H of p nodes which we will want to list.

▶ **Definition 14** (p-partition tree, Figure 1)**.** *Let $G = (V, E)$ be a graph with n nodes and m edges, and let $p \le \log n$. A p-*partition tree *$T = T_{G,p}$ is a tree of p layers (depth $p-1$), where each non-leaf node has at most $x = n^{1/p}$ children. Each node in the tree is associated with a partition of V consisting of at most x parts.*

We inductively denote all partitions associated with nodes in T as follows. The partition associated with the root r of T is called the root partition*, and is denoted by $P_\emptyset$. Given a node with a partition denoted by $P_{(\ell_1,\ldots,\ell_{i-1})}$, the partition associated with its jth child, for $0 \le j \le x-1$, is denoted $P_{(\ell_1,\ldots,\ell_{i-1},j)}$.*

The at most x parts of each partition $P_{(\ell_1,\ldots,\ell_i)}$ are denoted by $U_{(\ell_1,\ldots,\ell_i),j}$, for $0 \le j \le x-1$. For each $0 \le j \le x-1$, the part $U_{(\ell_1,\ldots,\ell_{i-1}),\ell_i}$ is called the parent *of the part $U_{(\ell_1,\ldots,\ell_{i-1},\ell_i),j}$, also denoted as $U_{(\ell_1,\ldots,\ell_{i-1}),\ell_i} = \texttt{parent}(U_{(\ell_1,\ldots,\ell_{i-1},\ell_i),j})$.*

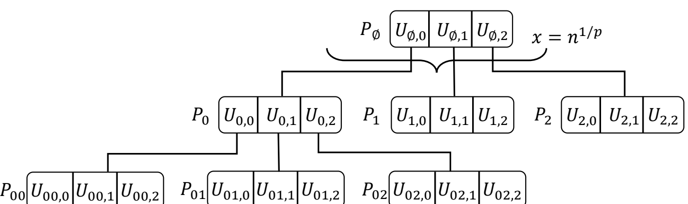


Figure 1 A partial illustration of a partition tree with $p, x = 3$.

▶ **Definition 15** (H-partition tree)**.** *Let $G = (V, E)$ be a graph with n nodes and m edges, and let H be a graph with $p \le \log n$ nodes, $\{z_0, \ldots, z_{p-1}\}$, and denote $d_i = |\{\{z_i, z_t\} \in E_H \mid t < i\}|$*

[1] It is possible to phrase a slightly stronger result which does not require a lower bound on the girth, but rather that for a specific k, which depends on the sparsity of the graph, there will not be any cycles of length $2k+1$.

for each $0 \leq i \leq p-1$*,* $x = n^{1/p}$ *and* $\tilde{m} = \max\{m, nx\}$*. A* H-partition tree $T = T_{G,H}$ *is a* p*-partition tree with the following additional constraints, for some constants* c_1, c_2*. .*

1. *for every part* $U = U_{(\ell_1,\ldots,\ell_{i-1},\ell_i),j}$*, it holds that* $|E(U,V)| \leq c_1 m/x + n$*, and*
2. *for every part* $U_i = U_{(\ell_1,\ldots,\ell_{i-1},\ell_i),j}$*, and all of its ancestor parts* $U_t = \texttt{parent}(U_{t+1})$ *for* $t = i-1, \ldots 0$*, it holds that* $\sum_{t<i,\{z_i,z_t\}\in E_H} |E(U,U_t)| \leq c_2 d_i \tilde{m}/x^2 + n$*,*

Notice that in Definition 15, we define $\tilde{m}$ as an upper bound on m, the number of edges in the input graph. This is done as if the graph is *too* sparse, we use a slightly higher bound on the number of edges in order to make decisions regarding the constraints on the partitions. We note that $\tilde{m}$ is purely a technicality – we do not *require* that there be at least this many edges in the graph. In the following two theorems we show that we can construct an H-partition tree and use it to efficiently perform H-listing.

▶ **Theorem 16.** *Let* $G = (V, E)$ *be a graph with* n *nodes, and let* H *be a graph with* $p \leq \log n$ *nodes. There exists a deterministic* Congested Clique *algorithm that completes in* $O(1)$ *rounds and constructs an* H*-partition tree* T*, such that* T *is known to all nodes of* G *– that is, all nodes know all the partitions making up* T*.*

Given an H-partition tree, we can list all instances of H in G.

▶ **Theorem 17.** *Let* $G = (V, E)$ *be a graph with* n *nodes, let* H *be a graph with* $p \leq \log n$ *nodes and* k *edges, and denote* $x = n^{1/p}$ *and* $\tilde{m} = \max\{m, nx\}$*. There exists a deterministic* Congested Clique *algorithm that completes in* $O(\frac{k\tilde{m}}{n^{1+2/p}} + p)$ *rounds and lists all instances of* H *in* G*, given an* H*-partition tree* T *that is known to all nodes.*

Thus, Theorems 16 and 17 directly imply Theorem 2.

Proof sketch of Theorem 16. We construct a set of preliminary partitions in $O(1)$ rounds, and show that it is possible to construct the entire partition tree using only this set of partitions. By ensuring that these partitions are globally known, each node computes the entire tree locally. The root partition, denoted as R, only needs to adhere to the condition on the number of edges that touch each part, which is easily obtained by grouping nodes in any arbitrary order until the total number of edges exceeds the threshold.

Then, for every set of $1 \leq \ell \leq p-1$ parts of R, denoted $\{Q_{j_0}, \ldots, Q_{j_{\ell-1}}\}$, we construct a partition $M_{\{j_0,\ldots,j_{\ell-1}\}}$ of G with at most x parts. That is, we construct $(x/2+1)^{p-1} \leq x^{p-1} = n/x$ additional different partitions. When constructing the partition $M_{\{j_0,\ldots,j_{\ell-1}\}}$, we maintain three conditions. Primarily, we maintain Condition 1; that is, for every part $N = N_{\{j_0,\ldots,j_{\ell-1}\},k}$ it holds that $|E(N,V)| \leq c_1 m/x + n$. Furthermore, similarly to Condition 2, we ensure that for each part, $N = N_{\{j_0,\ldots,j_{\ell-1}\},k}$, $\sum_{0\leq i<\ell} |E(N, Q_{j_i})| \leq c_2 \ell \tilde{m}/x^2 + n$. Lastly, we ensure that $M_{\{j_0,\ldots,j_{\ell-1}\}}$ is a refinement of R.

To build the partitions and to be able to make each partition known to all nodes, we assign each partition with a set of x nodes that are called *the builder nodes.* Since we have n/x partitions, we can do this in a mutually disjoint manner. Finally, we show that each node can internally construct the entire partition tree from the set of all partitions that we created, because for every node in the partition tree we can always find the needed partition among the partitions M that we created. To see why, note first that all partitions satisfy Condition 1. Second, every partition in the tree has to satisfy Condition 2. Intuitively, this holds since the corresponding parts in the condition are parts in the root partition R, because we made sure that each partition is a refinement of R. Thus, for every set of at most p ancestor parts, we can always find among the partitions M that we created, a partition which corresponds to that set of ancestor parts with respect to Condition 2. ◀

Proof sketch of Theorem 17. Denote by $\{z_0, \ldots, z_{p-1}\}$ the nodes of H, and denote $d_i = |\{\{z_i, z_t\} \in E_H \mid t < i\}|$ for each $0 \leq i \leq p-1$. We assign each leaf of the H-partition tree T to x different nodes. Note that there are x^{p-1} leaves, which is at most n/x due to our choice of $x = n^{1/p}$. We abuse the notation and denote a node in T with the same notation as we use for the partition that is associated with it. Each leaf $P_{(\ell_1,\ldots,\ell_{p-1})}$ is thus assigned to x different nodes, and each part $U_{(\ell_1,\ldots,\ell_{p-1}),j}$ in each leaf partition is assigned to a different node. For each node $v \in V$, we denote by $U_{v,p-1}$ the part of the leaf partition that it is assigned to. Then, inductively, for every $i = p-2, \ldots 0$, we denote $U_{v,i} = \texttt{parent}(U_{v,i+1})$. Note that for all $v \in V$ we have that $U_{v,0}$ is a part in the root partition.

We now let each node $v \in V$ learn all edges in $\bigcup_{t<i \text{ s.t. } \{z_i,z_t\}\in E_H} E(U_{v,i}, U_{v,t})$ and list all the instances of H it sees. We prove that all instances of H in G are indeed listed by this approach, and that learning the required edges by all nodes takes $O(\frac{k\tilde{m}}{n^{1+2/p}} + p)$ rounds. ◀

5 Detecting Even Cycles and Computing the Girth in Congest

In this section we sketch our Congest algorithms for finding small even cycles and for computing the girth. Both algorithms use the same high-level idea for quickly finding a cycle of a given length; we begin with a high-level overview of this technique, and then give more details for the girth algorithm and the even-cycle detection algorithm.

5.1 Finding Cycles

Suppose we want to check if the graph contains a copy of the ℓ-cycle, C_ℓ. To simplify the exposition, let us assume that ℓ is even, $\ell = 2k$ (the odd case is handled in our exact girth algorithm, below). We partition copies of C_{2k} in the graph into two types:

Light cycles. A cycle $u_0, \ldots, u_{2k-1}$ is called *light* if $\deg(u_i) \leq n^{1/k}$ for each $i \in [2k]$. To find light $2k$-cycles, we "deactivate" all nodes with degree greater than $n^{1/k}$, and consider the subgraph G' induced by the nodes with degree at most $n^{1/k}$. In G', we simultaneously start a depth-k BFS from all nodes, by having each node of G' send out a BFS token that is forwarded up to k hops by all the nodes of G' that receive it. When all BFS instances are completed, node u_k receives the BFS token of node u_0 from two of its neighbors, and announces that it has found a $2k$-cycle. All BFS instances complete in $O(k \cdot n^{(k-1)/k})$ rounds, as the i-neighborhood of any node in G' is of size at most $n^{i/k}$ for each $i \in [k]$.

We must be careful to avoid "false positives", which might occur if u_0's token travels to u_k by paths that are not vertex-disjoint; in this case, even though u_k receives u_0's token from two neighbors, the graph might not contain a $2k$-cycle – it contains some smaller cycle. In our C_{2k} algorithm, since we are interested in constant (and very small) k, we resolve this issue using color coding [1], which incurs a multiplicative factor of $k^{O(k)}$ in the running time. In our exact girth algorithm, we cannot afford to use color coding, because we must consider cycles of super-constant length; however, here we can argue that by the time we search for ℓ-cycles, we have already ruled out the existence of ℓ'-cycles for all $\ell' < \ell$, and therefore no "false positives" can occur.

Heavy cycles. A cycle $u_0, \ldots, u_{2k-1}$ is called *heavy* if it contains some node u_i with $\deg(u_i) > n^{1/k}$. Assume that u_0 is a node with the highest degree in the cycle, so that $\deg(u_0) > n^{1/k}$. To detect a heavy cycle, we exploit the fact that a randomly-sampled node

s in the graph will be a *neighbor* of u_0 with probability $\Omega(n^{1-1/k})$.[2] If we start a depth-k BFS from every neighbor of s, that would include u_0, and we could then detect the cycle by having node u_k receive u_0's token from its two neighbors u_{k-1}, u_{k+1}. There are two issues with this approach:

- As in the case of light cycles, a cycle of length $< 2k$ could cause a "false positive". We avoid this the same way we did above, using color-coding (for C_{2k}-detection) or the fact that there are no smaller cycles (for computing the girth).
- Starting a BFS from every neighbor of s could be too expensive: s could have high degree, and starting a BFS from many nodes at the same time could cause high congestion. We resolve this issue by:
  (I) First checking whether or not s itself participates in a $2k$-cycle, by starting a k-round BFS from s (using color-coding if necessary); and
  (II) Proving that if s does not participate in a $2k$-cycle, then w.h.p., it suffices for each cycle node to forward a *constant* number of BFS tokens, in order for u_k to receive the BFS token of u_0 from both u_{k-1} and u_{k+1}. For the girth this is straightforward, but for C_{2k}-detection it is more involved.

Together, we see that each time we sample a uniformly-random graph node, we can check in constant time whether it participates in or is adjacent to a $2k$-cycle. Since $\deg(u_0) > n^{1/k}$, after $O(n^{1-1/k})$ iterations we are guaranteed to find the cycle with high probability. We now give more details for each algorithm.

5.2 Computing the Girth

To compute the girth of the graph, we consider each length $\ell = 1, 2, \ldots, \log n$ sequentially, and search for ℓ-cycles in the graph using the framework described above. Each iteration takes $\ell \cdot n^{1-1/\lfloor \ell/2 \rfloor}$ rounds. By itself, this approach would have superlinear running time in graphs with high girth. In order to cap the running time at $O(n)$ rounds we run in parallel the linear-time all-pairs shortest path (APSP) algorithm of [16]. If the APSP algorithm terminates first, we use the results of APSP to compute the exact girth, and halt. Altogether, if the graph has girth g, then we terminate after $O(\min\left\{g \cdot n^{1-1/\lfloor g/2 \rfloor}, n\right\})$ rounds.

Let us instantiate the cycle-finding framework above for the case where we know the graph does not contain ℓ'-cycles for any $\ell' < \ell$, and want to check if the graph contains an ℓ-cycle. Each time we start a BFS, we let it proceed to depth $\lceil \ell/2 \rceil$, with the BFS token storing how many hops (edges) it has traveled; a node u detects an ℓ-cycle if it receives the BFS token of some node v from two distinct neighbors $w, w' \in N(u)$, with the token having travelled $\lceil \ell/2 \rceil$ hops to reach w and $\lfloor \ell/2 \rfloor$ hops to reach w'. We must address the following concerns:

"False positives". Suppose that node $u_{\lceil \ell/2 \rceil}$ receives the BFS token of node u_0 from its two neighbors $u_{\lceil \ell/2 \rceil - 1}$ and $u_{\lceil \ell/2 \rceil + 1}$, with the token having traveled $\lceil \ell/2 \rceil$ hops to reach $u_{\lceil \ell/2 \rceil - 1}$, and $\lfloor \ell/2 \rfloor$ hops to reach $u_{\lceil \ell/2 \rceil + 1}$. Let $u_0 = v_0, v_1, \ldots, v_{\lceil \ell/2 \rceil} = u_{\lceil \ell/2 \rceil}$ and $u_0 = w_0, w_1, \ldots, w_{\lfloor \ell/2 \rfloor} = u_{\lceil \ell/2 \rceil}$ be the paths along which u_0's token traveled to $u_{\lceil \ell/2 \rceil}$, and assume for the sake of contradiction that these paths contain some shared node $v_i = w_j$,

[2] We cannot actually sample one random node in the CONGEST model, but we can simulate this by assigning each node a random priority from the range $[n^3]$, so that w.h.p. the priorities are unique. Then we have the smallest-priority node "kill off" all its competitors in a neighborhood whose size matches the running time of the algorithm. This has the same effect as sampling one uniform node.

where $i, j > 0$ and either $i < \lceil \ell/2 \rceil$ or $j < \lfloor \ell/2 \rfloor$ (or both). Then, taking a lexicographically-smallest pair (i, j) such that $v_i = w_j$, we see that the graph contains a simple cycle $u_0, v_1, \ldots, v_i = w_j, w_{j-1}, \ldots, u_0$ of length at most $i + j < \ell$, contradicting our assumption that the graph is free of cycles of length less than ℓ.

Thus, the paths through which u_0's token reaches $u_{\lceil \ell/2 \rceil}$ must be node-disjoint (except their shared endpoints), which means there is a simple ℓ-cycle in the graph.

Congestion. suppose we have sampled a neighbor $s \in N(u_0)$, and verified that s does not participate in an ℓ-cycle. We now initiate a BFS from every neighbor of s (including u_0), and claim that u_0's BFS token is received by each cycle node $u_1, \ldots, u_{\ell-1}$ before any other BFS tokens are received: thus, it suffices to have each node forward one token only, and discard all tokens received afterwards. This claim is easily proven by induction on the distance of u_i from u_0: essentially, if some neighbor u_0' of s is able to "overtake" u_0's BFS token and arrive at some cycle node u_i first (or at the same time), and if u_i is the nearest such node to u_0, then we get a cycle of length at most ℓ that passes through s. But we already verified that s does not participate in an ℓ-cycle, and that the graph contains no smaller cycles, so this cannot happen.

5.3 $2k$-Cycle Detection

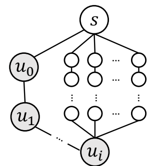
The details and especially the correctness proof of our $2k$-cycle detection algorithm are more complex, so we give here a high-level overview of the ideas, and defer the details to the full version. For the sake of symmetry, we represent the nodes of a $2k$-cycle as $u_0, u_1, \ldots, u_k = u_{-k}, u_{-(k-1)}, \ldots, u_{-1}$, so that the two paths leading from u_0 to u_k are given by $u_0, u_1, \ldots, u_k$ and $u_0, u_{-1}, \ldots, u_{-k} = u_k$. We use the same high-level framework for cycle-detection described above, but add the following ingredients.

Color coding [1]: to avoid small cycles leading to "false positives", we assign to each node $u \in V$ a random color, $c(u) \in \{-(k-1), \ldots, k\}$. Whenever we carry out a BFS, only nodes with color 0 may initiate a BFS, and a node u with color $c(u) = b \cdot i$, where $b \in \{-1, +1\}$ and $i \in \{0, \ldots, k-1\}$ is only allowed to forward BFS tokens to nodes with color $b \cdot (i+1)$. This ensures that if node u_k receives the BFS token of u_0 from its neighbors u_{k-1} and $u_{-(k-1)}$, then the token was carried along two paths that are node-disjoint, implying the presence of a simple $2k$-cycle. Since nodes choose their colors independently, the probability that a given cycle is "colored correctly" (i.e., $c(u_i) = i$ for each i) is $(1/2k)^{2k}$, a constant. We repeat each step of the algorithm sufficiently many times to ensure that if a $2k$-cycle exists, it is colored correctly at least once w.h.p, and then our color-coded BFS finds it.

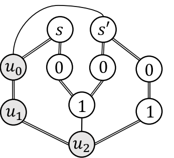
Limiting congestion: When we search for heavy cycles, we sample a uniformly random node s, check if it is part of a $2k$-cycle, and if not, we start a color-coded BFS from each 0-colored neighbor of s. There can be many such neighbors, potentially leading to congestion; however, we show that if the cycle is colored correctly, it suffices for each node with color $i \in [2k]$ to forward a constant number $T_k(i)$ of BFS tokens.

Our main concern is that the node s that we sampled is "bad", in the sense that it has many short node-disjoint paths to some cycle node u_i. If we sample such a "bad neighbor" of u_0, its 0-colored neighbors could initiate many BFS instances, which would then reach u_i and cause congestion. See Figure 2a for an illustration.

To bound the probability that we hit a "bad neighbor", we first rule out any neighbor of u_0 that itself participates in a $2k$-cycle. Next, we argue that if s has many node-disjoint paths to some cycle node u_i, such that the path nodes are colored $0, 1, \ldots, i$ (so that a



(a) A "bad neighbor" $s \in N(u_0)$.



(b) "Shared paths": the edges of the 10-cycle are indicated by double lines.

Figure 2 Illustrations for the proof sketch of the $2k$-cycle algorithm.

BFS can be initiated by the first path node and flow across the path), then we can charge these paths against the degree of u_i, as each path ends at a different neighbor of u_i. Since $\deg(u_0) \geq \deg(u_i)$, this means only a small constant fraction of u_0's neighbors have many such disjoint paths. When we sample a random node, we are unlikely to hit a "bad neighbor". (We are not worried about non-disjoint paths, as they do not contribute any "new" BFS tokens; see full version for details).

The problem with this argument is that if different neighbors of u_0 *share* paths to u_i, we might be overcounting when we charge each path against the degree of u_i. Our solution is to show that there is "not too much" sharing, otherwise a $2k$-cycle appears – and since we only consider neighbors of u_0 that are not on a $2k$-cycle, we know that this is impossible.

In Figure 2b, we show an example of one situation that must be ruled out (among others): consider $k = 5$ (i.e., 10-cycles), and suppose that two distinct neighbors $s, s' \in N(u_0)$ each have at least 10 node-disjoint paths with the "right colors", 0-1, to u_2. Suppose further that one of these paths is *shared*, as shown in the figure. In addition, node s' has at least one additional path (the rightmost path in the figure), which must exist because s' has at least 10 node-disjoint paths to u_2 (so at least one of these paths avoids all the other nodes shown in the figure). We see that there is a 10-cycle involving nodes s and s'; since we only consider neighbors of u_0 that do not themselves participate in a 10-cycle, this situation cannot arise.

6 Sketch of the $\Omega(n^{1/2+\alpha})$-Round Barrier on Lower Bounds for C_6-Detection in Congest

We prove that any lower bound on C_6-detection of the form $\Omega(n^{1/2+\alpha})$, where $\alpha > 0$, would imply new and powerful circuit lower bounds (namely, a superlinear lower bound on the number of wires in a circuit of polynomial depth). To show this, we first show that C_6-detection reduces to finding a directed triangle; this means that lower bounds on C_6-detection imply lower bounds on finding directed triangles. Next, we follow the same proof from [10] (which was given there for *undirected* triangles) to show that a lower bound of the form $\Omega(n^\alpha)$ on finding directed triangles would imply new circuit lower bounds.

The main idea of the reduction from 6-cycles to directed triangles is as follows. First, we eliminate all vertices with degree $\geq \sqrt{n}$, by calling the heavy-cycle subroutine of our new C_6-detection algorithm for $O(\sqrt{n}\log n)$ iterations. Each iteration requires constant time, and after $O(\sqrt{n}\log n)$ iterations, if there is a cycle containing a node with degree $> \sqrt{n}$, we will find one w.h.p.: w.h.p at least one iteration samples a node that is a neighbor of the cycle node that has degree $> \sqrt{n}$. Following this step, we "erase" all vertices with degree $\geq \sqrt{n}$ from the graph, together with all their edges. Let G' be the resulting graph.

Next, each vertex of G' chooses a random color $c(u) \in [6]$, and we simulate an algorithm for finding directed triangles on the directed graph $H = (V, E_H)$, where $(u, v) \in E_H$ iff $c(v) = c(u) + 2 \bmod 6$ and there is a path u, w, v in G' with $c(w) = c(v) + 1 \bmod 6$. It holds that any directed triangle in H corresponds to a well-colored simple 6-cycle in G', and vice-versa. Since G' can simulate one round in H using $\sqrt{n}$ rounds (as G' has maximum degree $\sqrt{n}$), if we can find a directed triangle in $O(n^{\alpha})$ rounds in H, then we can find a 6-cycle in $O(n^{1/2+\alpha})$ rounds in G'. Thus, intractability results for 6-cycles imply intractability for directed triangles, and following [10,11] we show this would imply new circuit lower bounds.

References

1. Noga Alon, Raphael Yuster, and Uri Zwick. Color-coding. *J. ACM*, 42(4):844–856, 1995.
2. Noga Alon, Raphael Yuster, and Uri Zwick. Finding and counting given length cycles. *Algorithmica*, 17(3):209–223, 1997. doi:10.1007/BF02523189.
3. Keren Censor-Hillel, Michal Dory, Janne H. Korhonen, and Dean Leitersdorf. Fast approximate shortest paths in the congested clique. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing (PODC 2019)*, pages 74–83, 2019.
4. Keren Censor-Hillel, Petteri Kaski, Janne H. Korhonen, Christoph Lenzen, Ami Paz, and Jukka Suomela. Algebraic methods in the congested clique. *Distributed Comput.*, 32(6):461–478, 2019. doi:10.1007/s00446-016-0270-2.
5. Keren Censor-Hillel, Dean Leitersdorf, and Elia Turner. Sparse matrix multiplication and triangle listing in the congested clique model. *Theor. Comput. Sci.*, 809:45–60, 2020. doi:10.1016/j.tcs.2019.11.006.
6. Yi-Jun Chang, Seth Pettie, and Hengjie Zhang. Distributed triangle detection via expander decomposition. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2019)*, pages 821–840, 2019.
7. Yi-Jun Chang and Thatchaphol Saranurak. Improved distributed expander decomposition and nearly optimal triangle enumeration. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing (PODC 2019)*, pages 66–73, 2019.
8. Danny Dolev, Christoph Lenzen, and Shir Peled. "tri, tri again": Finding triangles and small subgraphs in a distributed setting. In *Proceedings of the 26th International Symposium on Distributed Computing (DISC 2012)*, pages 195–209, 2012.
9. Andrew Drucker, Fabian Kuhn, and Rotem Oshman. On the power of the congested clique model. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing (PODC 2014)*, pages 367–376, 2014.
10. Talya Eden, Nimrod Fiat, Orr Fischer, Fabian Kuhn, and Rotem Oshman. Sublinear-time distributed algorithms for detecting small cliques and even cycles. In *Proceedings of the 33rd International Symposium on Distributed Computing (DISC 2019)*, volume 146 of *LIPIcs*, pages 15:1–15:16, 2019.
11. Talya Eden, Nimrod Fiat, Orr Fischer, Fabian Kuhn, and Rotem Oshman. A note on the hardness of proving distributed lower bounds for triangle-freeness. https://www.cs.tau.ac.il/~roshman/papers/EFFKO19_triangle_note.pdf, 2020.
12. Orr Fischer, Tzlil Gonen, Fabian Kuhn, and Rotem Oshman. Possibilities and impossibilities for distributed subgraph detection. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures (SPAA 2018)*, pages 153–162, 2018. doi:10.1145/3210377.3210401.
13. Silvio Frischknecht, Stephan Holzer, and Roger Wattenhofer. Networks cannot compute their diameter in sublinear time. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2012)*, pages 1150–1162, 2012.
14. Zoltán Füredi and Miklós Simonovits. *The History of Degenerate (Bipartite) Extremal Graph Problems*, pages 169–264. Springer Berlin Heidelberg, 2013.

15 Juho Hirvonen, Joel Rybicki, Stefan Schmid, and Jukka Suomela. Large cuts with local algorithms on triangle-free graphs. *Electr. J. Comb.*, 24(4):P4.21, 2017. URL: `http://www.combinatorics.org/ojs/index.php/eljc/article/view/v24i4p21`.

16 Stephan Holzer and Roger Wattenhofer. Optimal distributed all pairs shortest paths and applications. In *Proceedings of the 2012 ACM Symposium on Principles of Distributed Computing (PODC 2012)*, pages 355–364, 2012.

17 Alon Itai and Michael Rodeh. Finding a minimum circuit in a graph. *SIAM J. Comput.*, 7(4):413–423, 1978. `doi:10.1137/0207033`.

18 Taisuke Izumi and François Le Gall. Triangle finding and listing in CONGEST networks. In *Proceedings of the 2017 ACM Symposium on Principles of Distributed Computing (PODC 2017)*, pages 381–389, 2017.

19 Janne H. Korhonen and Joel Rybicki. Deterministic subgraph detection in broadcast CONGEST. In *Proceedings of the 21st International Conference on Principles of Distributed Systems (OPODIS 2017)*, pages 4:1–4:16, 2017.

20 François Le Gall. Further algebraic algorithms in the congested clique model and applications to graph-theoretic problems. In *Proceedings of the 30th International Symposium on Distributed Computing (DISC 2016)*, pages 57–70, 2016.

21 Christoph Lenzen. Optimal deterministic routing and sorting on the congested clique. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing (PODC 2013)*, pages 42–50, 2013.

22 Andrzej Lingas and Eva-Marta Lundell. Efficient approximation algorithms for shortest cycles in undirected graphs. *Inf. Process. Lett.*, 109(10):493–498, 2009.

23 Gopal Pandurangan, Peter Robinson, and Michele Scquizzato. On the distributed complexity of large-scale graph computations. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures (SPAA 2018)*, pages 405–414, 2018.

24 David Peleg, Liam Roditty, and Elad Tal. Distributed algorithms for network diameter and girth. In *Proceedings of the 39th International Colloquium on Automata, Languages, and Programming (ICALP 2012)*, pages 660–672, 2012.

25 Seth Pettie and Hsin-Hao Su. Distributed coloring algorithms for triangle-free graphs. *Information and Computation*, 243:263–280, 2015.

26 Liam Roditty and Roei Tov. Approximating the girth. *ACM Trans. Algorithms*, 9(2):15:1–15:13, 2013.

27 Liam Roditty and Virginia Vassilevska Williams. Minimum weight cycles and triangles: Equivalences and algorithms. In *Proceedings of the IEEE 52nd Annual Symposium on Foundations of Computer Science (FOCS 2011)*, pages 180–189, 2011.

28 Liam Roditty and Virginia Vassilevska Williams. Subquadratic time approximation algorithms for the girth. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2012)*, pages 833–845, 2012.

Improved MPC Algorithms for MIS, Matching, and Coloring on Trees and Beyond

Mohsen Ghaffari
ETH Zürich, Switzerland
ghaffari@inf.ethz.ch

Christoph Grunau
ETH Zürich, Switzerland
cgrunau@student.ethz.ch

Ce Jin
Tsinghua University, Beijing, China
cejin@mit.edu

Abstract

We present $O(\log \log n)$ round scalable Massively Parallel Computation algorithms for maximal independent set and maximal matching, in trees and more generally graphs of bounded arboricity, as well as for coloring trees with a constant number of colors. Following the standards, by a *scalable* MPC algorithm, we mean that these algorithms can work on machines that have capacity/memory as small as n^δ for any positive constant $\delta < 1$. Our results improve over the $O(\log^2 \log n)$ round algorithms of Behnezhad et al. [PODC'19]. Moreover, our matching algorithm is presumably optimal as its bound matches an $\Omega(\log \log n)$ conditional lower bound of Ghaffari, Kuhn, and Uitto [FOCS'19].

2012 ACM Subject Classification Theory of computation → Massively parallel algorithms

Keywords and phrases Massively Parallel Computation, MIS, Matching, Coloring

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.34

Funding This work was supported in part by funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 853109), and the Swiss National Foundation (project grant 200021-184735).

1 Introduction and Related Work

We present improved algorithms for some of the central graph problems in the study of large-scale algorithms – namely maximal matching and matching approximation, maximal independent set, and graph coloring – in the Massively Parallel Computation (MPC) setting. We first review the related model and known results, and then we present our contributions.

1.1 Massively Parallel Computation

The MPC model was introduced by Karloff et al. [25], and after some refinements [23, 8], it has by now become the standard theoretical abstraction for studying algorithmic aspects of large-scale data processing. This model captures the commonalities of popular practical frameworks such as MapReduce [18], Hadoop [33], Spark [34] and Dryad [24].

In the particular case of MPC for graph problems, we assume that our n-node m-edge input graph is partitioned arbitrarily among a number of machines, each with memory S. This *local memory* S is assumed to be considerably smaller than the entire graph size. Over all the machines, the summation of memories (called the *global memory*) should be at least $\Omega(m + n)$, and often assumed to be at most $\widetilde{O}(m + n)$. The machines communicate in synchronous message-passing rounds, subject to only one constraint: per round, each

© Mohsen Ghaffari, Christoph Grunau, and Ce Jin;
licensed under Creative Commons License CC-BY
34th International Symposium on Distributed Computing (DISC 2020).
Editor: Hagit Attiya; Article No. 34; pp. 34:1–34:18
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

machine can only send or receive a data of size at most S, which is all that it could fit in its memory. The objective is to solve various graph problems in the fewest possible number of rounds.

As the graph sizes are getting larger and larger (at a pace that exceeds that of single computers), we would like to have algorithms where each machine's local memory is much smaller than the entire graph size. However, dealing with smaller local memory is known to increase the difficulty of the problem, as we soon review in state-of-the-art. Based on this, the state-of-the-art in MPC algorithms can be divided into three regimes, depending on how the local memory S compares with the number of vertices n: (A) The strongly super-linear memory regime where $S \geq n^{1+\delta}$ for a constant $\delta > 0$, (B) The near-linear memory regime where $S = n \operatorname{poly}(\log n)$, and (C) the strongly sublinear memory regime where $S \leq n^{\delta}$ for a positive constant $\delta < 1$. We note that algorithms in the last category are the most desirable ones and they are usually referred to as *scalable MPC algorithms*.

1.2 State-of-the-Art

For the problems under consideration in this paper, simple $O(\log n)$ time algorithms follow from classic literature in the LOCAL model and PRAM [30, 1]. The primary objective in MPC is to obtain significantly faster algorithms than the PRAM counterpart, ideally just constant rounds or $O(\log\log n)$ rounds. Over the past few years, there has been progress toward this, gradually moving toward the lower memory regimes.

For the strongly superlinear memory regime, Lattanzi et al. [27] presented $O(1)$-round algorithms for maximal matching and maximal independent set. Progress to lower memory regimes was slow afterward. But that changed with a breakthrough of Czumaj et al. [17], who gave an $O(\log^2 \log n)$ round algorithm for $(2+\varepsilon)$-approximation of maximum matching in the near-linear memory regime. Shortly after, for the same near-linear memory regime, Assadi et al. [3] gave an $O(\log\log n)$ round algorithm for $(1+\varepsilon)$-approximation of maximum matching and constant approximation of minimum vertex cover and independently Ghaffari et al. [20] gave $O(\log\log n)$ round algorithms for $(1+\varepsilon)$-approximation of maximum matching, $(2+\varepsilon)$-approximation of minimum vertex cover, and maximal independent set. Finally, Behnezhad et al. [12] gave an $O(\log\log n)$ round algorithm for maximal matching.

For the much more stringent strongly sublinear memory regime, there has also been some progress but much slower: For the case of general graphs, Ghaffari and Uitto [22] provide an $\widetilde{O}(\sqrt{\log \Delta} + \log\log n)$ algorithm for graphs with maximum degree Δ, which remains the best known. When $\Delta > n^{\Omega(1)}$, the only known algorithms that outperform this bound and reach the $\operatorname{poly}(\log\log n)$ regime are results for special graph families, namely trees [14, 6, 7] and more generally graphs of small, e.g., polylogarithmic, arboricity [13, 10, 9]. Recall that the *arboricity* α of a graph is the minimum number of forests into which its edges can be partitioned. Many natural graph classes, such as planar graphs and graphs with bounded treewidth, have small arboricity. Concretely, Behnezhad et al. [9] provide Maximal Matching and MIS algorithms that run in $O((\log\log n)^2)$ rounds and leave remaining graphs with maximum degree $\operatorname{poly}(\max(\log n, \alpha))$. By invoking the aforementioned algorithm of Ghaffari and Uitto [22], they obtain maximal matching and MIS algorithms that have round complexity $O(\sqrt{\log \alpha}\log\log \alpha + (\log\log n)^2)$. Finally, a result of Ghaffari, Kuhn, and Uitto [21] shows a conditional lower bound (for *component-stable* MPC algorithms[1]) of $\Omega(\log\log n)$ for maximal

[1] Most of the MPC algorithms in the literature are component-stable. One exception is a recent result due to Czumaj, Davies, and Parter [16], showing a deterministic MPC algorithm for MIS and maximal

matching and MIS in this strongly sublinear memory regime. In essence, they show that component-stable MPC algorithms in the strongly sublinear memory regime cannot be more than exponentially faster compared to their LOCAL-model counterpart, and then they use lower bounds in the LOCAL model. Their result is conditioned on the widely believed $\Omega(\log n)$ complexity of the connectivity problem in the strongly sublinear memory regime. We note that although an $\Omega(\log n)$ round complexity lower bound is widely believed to be true for the connectivity problem in the strongly sublinear memory regime, even for distinguishing an n-node cycle from two $n/2$-node cycles, proving any super-constant lower bound would imply $\mathsf{P} \not\subset \mathsf{NC}^1$, a major breakthrough in circuit complexity [32]. Concretely, in the case of maximal matching and MIS, the results of Ghaffari et al. [21] invoke the $\Omega(\sqrt{\log n / \log\log n})$ round LOCAL-model lower bound of Kuhn, Moscibroda, and Wattenhofer [26] to show that an $o(\log\log n)$-round MPC algorithm for maximal matching or MIS would imply an $o(\log n)$ round algorithm for connectivity, which would break the conjectured $\Omega(\log n)$ complexity. We emphasize that in the case of maximal matching, this $\Omega(\log\log n)$ round complexity lower bound for MPC algorithms holds even if the input graph is a tree.

1.3 Our Contribution

Our main contribution is an improvement over the algorithm of Behnezhad et al. [9]. We obtain MPC algorithms in the strongly sublinear memory regime that solve maximal matching and MIS in $O(\log\log n)$ rounds, in trees and constant-arboricity graphs. And more generally as a function of the arboricity α, our maximal matching and MIS algorithms run in $O(\sqrt{\log\alpha}\log\log\alpha + \log\log n)$ rounds, where the first term again comes from invoking the algorithm of Ghaffari and Uitto [22] for remaining graphs with maximum degree $\mathrm{poly}(\alpha)$.

▶ **Theorem 1.** *There is an $O(\sqrt{\log\alpha}\log\log\alpha + \log\log n)$ round MPC algorithm using n^δ local memory and $\widetilde{O}(n+m)$ global memory that with high probability computes a maximal matching and a maximal independent set of a graph with arboricity α.*

We note that on graphs with polylogarithmic arboricity, our bound provides a quadratic improvement over the previous algorithms [10, 13, 9]. Moreover, our algorithm for maximal matching is presumably optimal, as the $\Omega(\log\log n)$ conditional lower bound shown by Ghaffari, Kuhn, and Uitto [21] holds even on trees (where $\alpha = 1$).

Our second contribution is providing a similarly fast algorithm for 4-coloring trees.

▶ **Theorem 2.** *There is an $O(\log\log n)$ round MPC algorithm using n^δ local memory and $O(n+m)$ global memory that with high probability computes a 4-coloring of any tree.*

This algorithm also matches a conditional lower bound of $\Omega(\log\log n)$, which follows from the conditional impossibility results of [21], showing that strongly sublinear memory MPC algorithms cannot be more than exponentially fast compared to their LOCAL model counterpart, and the $\Omega(\log n)$ LOCAL-model lower bound for $O(1)$-coloring trees of Linial [29].

1.4 Preliminaries and Notations

Let $[n]$ denote the set $\{1, 2, \ldots, n\}$. Let $G = (V, E)$ be an undirected graph. Let $\deg_G(u)$ denote the degree of a vertex u in graph G, and let $\mathrm{dis}_G(u, v)$ denote the minimum number of edges in a path connecting u and v (when G is clear from the context, we may also

matching using strongly sublinear local memory.

write $\deg(u)$ and $\mathrm{dis}(u,v)$ for simplicity). For a vertex subset $V' \subseteq V$, define $\mathrm{dis}_G(u,V') = \min_{v\in V'} \mathrm{dis}_G(u,v)$. The *k-hop neighborhood* of a vertex v in G is the subgraph containing all vertices u satisfying $\mathrm{dis}_G(u,v) \le k$, together with every edge that lies on some path starting from v with length at most k. The k-hop neighborhood of a vertex set $V' \subseteq V$ is the union of k-hop neighborhoods of $v \in V'$. We use $G[V']$ to denote the subgraph of G induced by the vertex subset $V' \subseteq V$. For simplicity we use $\mathrm{dis}_{V'}(u,v)$ as a shorthand for $\mathrm{dis}_{G[V']}(u,v)$, when the underlying graph G is clear from the context; similarly, we can define $\deg_{V'}(u)$.

The *arboricity* α of a graph is the minimum number of forests into which its edges can be partitioned. Nash-Williams [31] showed that the arboricity is equal to the maximum value of $\lceil m_S/(n_S-1)\rceil$, where m_S, n_S are the number of edges and vertices in any subgraph S.

2 MIS and Matching

In this section we prove Theorem 1. In Section 2.1, we review the previous work by Behnezhad et al. and informally describe our new techniques for improving their algorithm. In Section 2.2, we give formal definitions and describe the structure of our pipelined MPC algorithm. Then in Section 2.3 and Section 2.4, we present the implementation and loop invariants of our algorithm in detail. In Section 2.5, we analyze the space complexity of the MPC algorithm.

2.1 Overview

Review of Behnezhad et al.'s algorithm. We first briefly review the main ideas of the previous MPC algorithm for Maximal Independent Set (MIS) and Maximal Matching (MM) by Behnezhad et al. [9, 10]. Their result is based on the LOCAL algorithm due to Barenboim et al. [5]. This LOCAL algorithm is formally stated in Lemma 3. In a graph with small arboricity α and large maximum degree $\Delta \ge \mathrm{poly}(\alpha, \log n)$, the LOCAL algorithm finds a matching (or an independent set) in $O(1)$ rounds and removes the involved vertices from the graph, so that the number of high-degree vertices (i.e., with degree $> \sqrt{\Delta}$) decreases by a factor of Δ in the remaining graph.

By repeating this LOCAL algorithm $O(\log_\Delta n)$ times, all high-degree vertices are eliminated, that is, they either get removed from the graph or become low-degree. Thus, the remaining graph has maximum degree at most $\sqrt{\Delta}$. We call this procedure a *phase*. After $O(\log\log\Delta)$ phases, the maximum degree drops below $\mathrm{poly}(\alpha, \log n)$, and we switch to the algorithm by Ghaffari and Uitto [22] to find a maximal matching (or MIS) of the remaining low-degree graph. Combining with the partial solution obtained in previous phases, we then obtain a maximal matching (or MIS) of the input graph.

To implement the algorithm efficiently in the MPC model, Behnezhad et al. use the, by-now standard, graph exponentiation technique [28, 19]: For each vertex v, we store its d-hop neighborhood in one machine, with d initially being 1. In $O(1)$ MPC rounds we can double the radius to $2d$, by requesting and collecting the d-hop neighborhoods of all vertices u in the d-hop neighborhood of v. Hence in $O(\log d)$ MPC rounds we can collect the d-hop neighborhood of each vertex, which has at most Δ^d edges and can fit into the n^δ local memory when $d = \delta \log_\Delta n$. Then we can simulate $O(\log_\Delta n)$ LOCAL rounds in this phase using only $O(1)$ MPC rounds. As each phase requires $O(\log d) \le O(\log\log n)$ MPC rounds for graph exponentiation, the total round-complexity of this MPC algorithm is $O(\log\log n \log\log\Delta) \le O(\log^2 \log n)$.

The algorithm described above would need $\widetilde{O}(n^{1+\delta}+m)$ global memory. In order to reduce it to $\widetilde{O}(n+m)$, Behnezhad et al. used an additional idea: As our objective in a phase is to make high-degree vertices disappear, we can without loss of generality assume that

the LOCAL algorithm only removes vertices that are in the $O(1)$-hop neighborhood of the high-degree vertices, and other vertices are considered as irrelevant and do not participate in the LOCAL algorithm. As the number of high-degree vertices decreases after every execution of the LOCAL algorithm, the number of relevant vertices also decreases quickly. Then, we have enough space for expanding the stored neighborhood of each relevant vertex.

New techniques. Our new idea is a *pipelining* technique: as more vertices become irrelevant in the current phase, we can start running the next phase on these vertices, *concurrently* with the current phase which has not necessarily finished completely. In our algorithm, after starting phase $(\ell - 1)$, we wait for $O(1)$ MPC rounds and then start phase ℓ. There are $O(\log\log\Delta)$ phases in total, each taking $O(\log\log n)$ MPC rounds, so the round complexity is $O(\log\log\Delta)\cdot O(1) + O(\log\log n) = O(\log\log n)$ in total. Our pipelined algorithm produces exactly the same result as the unpipelined version does (using the same random bits).

When running phase ℓ, we need to deal with *pending* vertices that have not finished their computation in previous phases $1, 2, \ldots, \ell - 1$. As it is still unknown whether they will be removed by the start of phase ℓ, the messages sent from these vertices during phase ℓ are temporarily marked as *pending*. If a vertex receives a pending message, then its state also becomes pending, and so on. When simulating the LOCAL computation in phase ℓ, we are only able to simulate the behaviour of non-pending vertices; only upon finishing the simulation of previous phases $1, 2, \ldots, \ell - 1$ for some vertex, the initial state of this vertex in phase ℓ becomes known, which allows us to resume the phase ℓ computation around its neighborhood and remove the pending marks.

Now we take a closer look at what happens when we perform graph exponentiation. In the ideal scenario, after phase $\ell - 1$ completely finishes, the maximum degree drops below $\Delta_{\ell-1} := \Delta^{1/2^{\ell-1}}$, allowing us to store the $(\delta \log_{\Delta_{\ell-1}} n)$-hop neighborhood of a vertex into the local memory. However, in the actual situation, when expanding the neighborhood we may encounter pending vertices, which may not have degree upper bounded by $\Delta_{\ell-1}$. It would be problematic if we simply excluded these vertices when we expand the neighborhood, as they may later have degree reduced to the interval $(\Delta_\ell, \Delta_{\ell-1}]$ and become relevant in phase ℓ. So we have to store them as well, and this poses challenges to bounding the local memory and the global memory of our algorithm – this is the most technical part of our analysis.

In this overview, we provide some intuition on how we analyze the global memory used for storing the neighborhoods. We will analyze the neighborhoods of non-pending vertices and of pending vertices separately. In phase ℓ, after repeating the LOCAL algorithm k times, the number of non-pending vertices v with degrees in $(\Delta_\ell, \Delta_{\ell-1}]$ can be bounded by $n/\Delta_{\ell-1}^k$. The d-hop neighborhood (we will maintain $d = ck$ for some constant $c \in (0, 1)$) of such a non-pending vertex v only contains vertices u with degree at most $\Delta_{\ell-1}$, since otherwise u would still be waiting for phase $\ell - 1$ to finish, causing v to become a pending vertex. Hence the total size of d-hop neighborhoods of these non-pending vertices v is roughly at most $(n/\Delta_{\ell-1}^k) \cdot \Delta_{\ell-1}^d < n$. Now we consider the pending vertices. For any pending vertex v, it must be within k distance from a vertex u which is currently stuck at phase $\ell - q$, for some $q \geq 1$. We pick such u with the biggest q, so v can be reached from u by a k-step path on which all vertices have degree $\leq \Delta_{\ell-q}$ (since if any of them had degree greater than $\Delta_{\ell-q}$, it would be stuck at a phase earlier than $\ell - q$, contradicting the maximality of q). Hence we can bound the number of pending vertices v by $\sum_{q\geq 1} H_{\ell-q} \cdot \Delta_{\ell-q}^k$, where $H_{\ell-q}$ denotes the number of vertices currently stuck at phase $\ell - q$, and the total size of neighborhoods of these v can be bounded by a similar summation. In order to show a near-linear upper bound, we want to make this summation dominated by a geometric series. To do this, we use a similar argument to show an upper bound on $H_{\ell-q}$, and hence after setting appropriate parameters, we can use *induction on* ℓ to establish the desired memory bound.

2.2 Definitions and algorithm structure

Suppose each machine in the MPC model has n^δ local memory, for some constant $0 < \delta < 1$. We will use the following LOCAL algorithm [5] as a black box.

▶ **Lemma 3** ([5, Theorem 7.2], restated[2]; see also [10, Section 5.2]). *Let α and Δ be parameters satisfying $\Delta \geq \mathrm{poly}(\alpha, \log n)$. Suppose the input graph G has arboricity at most α and maximum degree at most Δ. Let $H_G := \{v : \deg_G(v) > \sqrt{\Delta}\}$ denote the set of high-degree vertices in graph G.*

There is a LOCAL algorithm of round complexity $r = O(1)$ that computes a matching M (or an independent set I) of G, such that $|H_{G'}| \leq |H_G|/\Delta$ w.h.p., where G' is the induced subgraph of G where matched vertices in M (or vertices in I and their neighbors) are removed. Moreover, the algorithm has the following properties.

(1) *The state description and random bits for each vertex, as well as communication along every edge in each round, have length at most $\mathrm{polylog}(n)$ bits. The space usage for computing the new state of vertex v is $\deg_G(v) \cdot \mathrm{polylog}(n)$, and the space usage for computing the message sent along each edge is $\mathrm{polylog}(n)$.*

(2) *Let H^+ denote the r-hop neighborhood of H_G in G. Then vertices outside H^+ are not affected by the algorithm (i.e., they will stay in graph G', and their degrees in G' are unchanged).*

Let $\Delta_\ell := \Delta^{1/2^\ell}$, where Δ is the maximum degree of the input graph. The unpipelined algorithm sequentially runs $L = O(\log\log\Delta)$ *phases*, where phase ℓ ($\ell = 1, 2, \ldots, L$) repeats the LOCAL algorithm $O(\log_{\Delta_\ell} n)$ times with degree parameter $\Delta_{\ell-1}$, and reduces the maximum degree from $\Delta_{\ell-1}$ to Δ_ℓ.

Our pipelined MPC algorithm runs in $O(\log\log n)$ *iterations* (starting from iteration 1), each taking $O(1)$ MPC rounds. In each iteration, there are multiple *phases* concurrently being simulated by our MPC algorithm. There is a small *lag* $t = O(1)$ between the start of phase ℓ and $\ell+1$, i.e., the simulation of phase ℓ starts at iteration $j = (\ell-1)t+1$, by running INITIALIZE(ℓ). In each iteration j, we perform SIMULATE(ℓ, j), UPDATE, and EXPAND(ℓ, j), concurrently for all active phases ℓ (i.e, phases that have already started). See Algorithm 1 for the algorithm structure. The value of parameter t will be determined in Section 2.5.

Algorithm 1 Structure of our pipelined MPC algorithm.

```
Number of phases L = O(log log Δ).
Lag parameter t = O(1).
for iteration j ← 1, 2, ..., O(log log n) do
    L_active ← min{L, ⌈(j − 1)/t⌉} (Number of phases that have already started)
    if j = (ℓ − 1)t + 1 for some ℓ ∈ [L] then
        INITIALIZE(ℓ) (Phase ℓ starts in this iteration)
    SIMULATE(ℓ, j) for all ℓ ∈ [L_active] concurrently
    UPDATE
    EXPAND(ℓ, j) for all ℓ ∈ [L_active] concurrently
Switch to Ghaffari and Uitto's algorithm [22]
```

Our MPC algorithm maintains the subgraph induced by the remaining vertices. After iteration j finishes, the remaining induced subgraph is denoted by $G(j)$. We have $G = G(0) \supseteq G(1) \supseteq \cdots$. In iteration j, concurrently for each active phase ℓ, SIMULATE(ℓ, j) simulates

[2] The proof of [5, Theorem 7.2] presented an $O(1)$-round algorithm that reduces the number of vertices with degree greater than $t\alpha$ by a $t^{1/7}$ factor, for any parameter $t \geq \mathrm{poly}(\alpha, \log n)$. By choosing $t = \sqrt{\Delta}/\alpha$ and repeating the algorithm $O(1)$ times we obtain the claimed statement.

(part of) the LOCAL computation in phase ℓ, and removes the matched vertices (or vertices in the independent set and their neighbors) from the graph. After removing the vertices, in Update we compute the degrees of vertices in the remaining graph $G(j)$, and update some other information. Then, in Expand(ℓ, j), we perform one graph exponentiation step and double the radius of the neighborhoods stored in memory.

2.3 LOCAL simulation with incomplete information

As discussed in Section 2.1, our pipelining algorithm is complicated by the presence of pending vertices. In this section we explain how to deal with them properly when simulating LOCAL algorithms.

When we simulate a LOCAL algorithm A on an input graph G, we may not have the complete information of the initial states of vertices. We say vertex v is *initially pending* if v's initial state is unknown.[3] We have the following simple observation:

▶ **Observation 4.** *Define a LOCAL algorithm A', which is the same as A except for the following additional rule: the messages sent from a pending vertex are marked as pending, and a vertex receiving a pending message becomes pending.*

If v's R-hop neighborhood in the input graph G does not contain an initially pending vertex, then v's state after running algorithm A' for R rounds is non-pending, and is the same as its state in algorithm A after R rounds.

Now we introduce another simple lemma which will be helpful in reducing the space usage of our MPC simulation. Consider a LOCAL algorithm A in which vertices may get eliminated during execution: after a vertex is eliminated, it no longer participates in later rounds of the algorithm. It is well-known that we can simulate the behaviour of a vertex v up to R LOCAL rounds provided that we know v's R-hop neighborhood in the input graph $G = (V, E)$. However, if we somehow know a subset U of vertices that will be eliminated by A, then we only need to collect v's R-hop neighborhood in the induced subgraph $G[V \setminus U]$ (plus the messages sent from U when they are alive), which could be much smaller than its R-hop neighborhood in G.

▶ **Lemma 5.** *Suppose a LOCAL algorithm A runs on an input graph $G = (V, E)$, and $U_0 \subseteq V$ is the set of vertices that are eliminated during the execution of A.*

Let $U \subseteq U_0$, and $v \in V \setminus U$. Then, we can simulate the behaviour of v in the first R rounds of the algorithm A, provided that we have the information of:

(a) *The R-hop neighborhood of v in the induced subgraph $G[V \setminus U]$.*

(b) *The complete communication history along every edge $(u, w) \in E$ where $u \in U$ and w belongs to the neighborhood defined in (a), until u gets eliminated by A.*

The proof of this lemma is trivial.

2.4 Algorithm in detail

Recall that each machine in the MPC model has n^δ memory. Define $s := \lceil 10/\delta \rceil$. Recall that $r = O(1)$ is the round complexity of the LOCAL algorithm from Lemma 3, and $G(j)$ is the graph induced by the remaining vertices at the end of iteration j. Let $H_\ell(j)$ denote the set

[3] Suppose LOCAL algorithm A refers to the computation in phase ℓ. Here, the *state* of a vertex v includes its degree, whether it has been removed, and some other information required by Lemma 3 Property (1). If we have not finished the phase $(\ell - 1)$ simulation for vertex v, then we do not know v's state at the beginning of phase ℓ, and we will consider v to be in a special *pending state*.

of vertices v with $\deg_{G(j)}(v) > \Delta_\ell$, and let $H_\ell^+(j)$ be the r-hop neighborhood of $H_\ell(j)$ in graph $G(j)$. From Lemma 3 (Property (2)), we observe that vertices outside $H_\ell^+(j)$ are not affected by phase $1, 2, \ldots, \ell$ in the following iterations $j+1, j+2, \ldots$.

At the end of iteration j, we maintain the following invariants for all active phases ℓ.

▶ **Property 6.** *Let $j = (\ell-1)t + i$ ($i \geq 1$), i.e., j is the i-th iteration since the start of phase ℓ. Let $d_i := 2^i r$, and recall that $s := \lceil 10/\delta \rceil$. At the end of iteration j, the following hold:*

1. ***(Number of finished LOCAL executions)*** *For every $v \in H_\ell^+(j-1)$, if there is no $u \in H_{\ell-1}^+(j-1)$ satisfying $\mathrm{dis}_{H_\ell^+(j-1)}(u, v) \leq sd_i \cdot r$, then we have computed v's state after sd_i executions of the LOCAL algorithm (from Lemma 3) in phase ℓ.*
2. ***(Radius of collected neighborhoods)*** *For every $v \in H_\ell^+(j)$, we have collected the following information into one machine:*
   (a) *The d_i-hop neighborhood of v in graph $H_\ell^+(j)$.*
   (b) *The communication history during phase ℓ along every edge (u, w), where vertex $u \notin H_\ell^+(j)$ was eliminated during phase ℓ, and w belongs to the neighborhood defined in (a).*

Informally, Item 1 says that we have simulated sd_i executions of the LOCAL algorithm (each using r rounds) in phases ℓ, as long as the $sd_i \cdot r$-hop neighborhood contained no initially pending vertices when iteration j began.

Now we describe how to implement INITIALIZE(ℓ), SIMULATE(ℓ, j), UPDATE, EXPAND(ℓ, j) in iteration $j = (\ell-1)t + i$ using $O(1)$ MPC rounds and maintain Property 6 (the purpose of these procedures has been informally described at the end of Section 2.2).

▶ **Lemma 7.** *Assume Property 6 holds at the end of iteration $j-1$. We can implement SIMULATE(ℓ, j), UPDATE, EXPAND(ℓ, j) (concurrently for all phases $\ell \in [L_{\mathsf{active}}]$, see 4th line in Algorithm 1) using $O(1)$ MPC rounds so that at the end of iteration j, Property 6 holds for all phases $\ell \in [L_{\mathsf{active}}]$.*

Proof. Let $j = (\ell-1)t + 1$. Recall that $d_i = 2^i r$. In the following algorithmic description we omit the low-level implementation details of sublinear-memory MPC model. We will briefly address them in Remark 9.

- SIMULATE(ℓ, j). We apply Observation 4, where A is the phase ℓ algorithm, and vertices in $H_{\ell-1}^+(j-1)$ are initially pending. To maintain Property 6 Item 1, it suffices to simulate algorithm A' (defined in Observation 4) up to $sd_i \cdot r$ LOCAL rounds.
  Apply Lemma 5, with U being the set of vertices $u \notin H_\ell^+(j-1)$ which were eliminated during phase ℓ. By Property 6 Item 2, we already have the d_{i-1}-hop neighborhood of every vertex in $H_\ell^+(j-1)$ (as well as the communication history required by Lemma 5) stored into one machine. So we can simulate d_{i-1} LOCAL rounds for all $v \in H_\ell^+(j-1)$ in one MPC round.
  After simulating d_{i-1} LOCAL rounds, we update the current states of the vertices in all stored d_{i-1}-hop neighborhoods, using $O(1)$ MPC rounds. Then we again simulate d_{i-1} LOCAL rounds for all vertices in one MPC round, so that the total number of simulated LOCAL rounds becomes $2d_{i-1}$. And we update the current states of vertices, and so on. In this way, we can simulate the first $sd_i \cdot r$ LOCAL rounds in phase ℓ using $O(sd_i \cdot r/d_{i-1}) = O(1)$ MPC rounds. (This is the "blind coordination" technique used by Behnezhad et al. [9, Section 5.3]) When we simulate the behaviour of v in phase ℓ, we also record the communication history of v. Since the degree of v in phase ℓ is at most $\Delta_{\ell-1}$, the size of the messages is $O(\Delta_{\ell-1} \cdot \mathrm{polylog}\, n)$.
- UPDATE. After vertices are removed from $G(j-1)$ by SIMULATE(ℓ, j), let $G(j)$ be the induced subgraph of the remaining vertices, and compute the vertex degrees in graph $G(j)$. Then, for every ℓ, delete from $H_\ell(j-1)$ the vertices whose degree dropped to $\leq \Delta_\ell$

and the ones that got removed, and obtain $H_\ell(j)$. Then similarly obtain $H_\ell^+(j)$. We also delete these vertices from the stored neighborhoods.

- EXPAND(ℓ, j). To maintain Property 6 Item 2, we perform one graph exponentiation step in $O(1)$ MPC rounds. Every $u \in H_\ell^+(j)$ requests the neighborhood of every other v in the stored d_{i-1}-hop neighborhood of u. The new radius becomes $2d_{i-1} = d_i$. Then it is easy to collect the required communication history of vertices in this neighborhood.

Note that SIMULATE(ℓ, j) and EXPAND(ℓ, j) are independent across all phases ℓ and can be executed concurrently. ◀

▶ **Lemma 8.** *At iteration $j = (\ell - 1)t + 1$, we can implement* INITIALIZE(ℓ) *using $O(1)$ MPC rounds so that at the end of this iteration Property 6 holds for phase ℓ.*

Proof. We run the LOCAL algorithm in phase ℓ for sd_1 times (in the sense of Observation 4). This can be done in $O(sd_1 \cdot r) = O(1)$ MPC rounds with $\widetilde{O}(n + m)$ global memory. Then, we collect the $2r$-hop neighborhood of every vertex in $H_\ell^+(j)$ (and the required communication history) in $O(1)$ MPC rounds. ◀

▶ Remark 9. We did not spell out the low-level details, but it is not hard to verify that our algorithm only involves basic operations and can be easily implemented using the standard MPC primitives developed in previous works, e.g. [2, Section E] and the references therein. The only concern is that, when $\Delta \geq n^\delta$, we cannot gather the neighborhood of a vertex into one machine. This issue was already addressed in Behnezhad et al.'s unpipelined algorithm [10, Section 6], using a load-balancing technique. Hence we can first sequentially run $O(1)$ phases of their algorithm (in $O(\log \log n)$ MPC rounds) at the very beginning, which reduce the maximum degree of the remaining graph to below n^α for any desired constant $\alpha > 0$.

Now we use Property 6 Item 1 to show the round complexity of our algorithm.

▶ **Theorem 10.** *Our pipelined MPC algorithm runs in $O(\sqrt{\log \alpha} \log \log \alpha + \log \log n)$ rounds.*

Proof. First consider the unpipelined scenario: in phase ℓ, by Lemma 3, after $\log n$ executions of the LOCAL algorithm, w.h.p. there are no remaining vertices with degree higher than Δ_ℓ.

Now we look at our pipelined MPC simulation. Note that each iteration takes $O(1)$ MPC rounds. After $j_1 = O(t \cdot L) = O(\log \log \Delta)$ iterations, all phases have started their simulation. Then, after another $j_2 = O(\log \log n)$ iterations, every phase ℓ has at least been running for $i > \log \log n$ iterations. Let $j = j_1 + j_2$ denote the number of iterations so far. By Property 6 Item 1, if $H_{\ell-1}^+(j-1)$ is empty, then at the end of iteration j we will have finished $sd_i > \log n$ many LOCAL executions of phase ℓ, which (together with the previous paragraph) implies $H_\ell(j)$ is empty. Hence, at this point $H_1(j)$ is already empty (since $H_0^+(j-1)$ is always empty). This in turn implies that in the next iteration $j+1$, $H_2(j+1)$ will be empty. By a simple induction, after $L-1$ iterations, $H_L(j+L-1)$ becomes empty, which means that the maximum degree of the remaining graph $G(j+L-1)$ is at most $\Delta_L = \text{poly}(\alpha, \log n)$. Finally, we use Ghaffari and Uitto's algorithm [22], which takes $O(\sqrt{\log \Delta_L} \log \log \Delta_L + \sqrt{\log \log n}) \leq O(\sqrt{\log \alpha} \log \log \alpha) + \widetilde{O}(\sqrt{\log \log n})$ MPC rounds. Hence, the total round complexity is $O(\sqrt{\log \alpha} \log \log \alpha + \log \log n)$. ◀

2.5 Memory requirement

In this section we will show that our algorithm satisfies the local memory and global memory constraints. In the following, we will always assume $j = (\ell - 1)t + i$, that is, j is the i-th iteration since phase $\ell \in [L]$ starts. Recall that $\Delta_\ell := \Delta^{1/2^\ell}$. For a subgraph H, we use $|H|$ to denote the number of vertices in H. We need the following lemma.

▶ **Lemma 11.** *With high probability,* $|H^+_\ell(j)| \le 2n/\Delta_{\ell-1}^{(s-1)d_i}$.

In order to prove Lemma 11, we first define the following quantity.

▶ **Definition 12.** *For every* $v \in H^+_\ell(j)$*, let* $D_{\ell,j}(v) := \max_{\substack{u \in G(j-1): \\ \mathrm{dis}_{G(j-1)}(u,v) \le (sd_i+2)r}} \big\{ \deg_{G(j-1)}(u) \big\}$.

Note that $D_{\ell,j}(v) \ge \deg_{G(j-1)}(v^-) > \Delta_\ell$, where $v^- \in H_\ell(j)$ and $\mathrm{dis}_{G(j-1)}(v^-, v) \le r$. Then, we classify the vertices in $H^+_\ell(j)$ by the values of $D_{\ell,j}(v)$.

▶ **Definition 13.** *The vertices in* $H^+_\ell(j)$ *are partitioned as* $\dot\bigcup_{0 \le q \le \ell-1} P^{(q)}_{\ell,j}$*, where*

$$P^{(q)}_{\ell,j} := \big\{ v \in H^+_\ell(j) : D_{\ell,j}(v) \in (\Delta_{\ell-q}, \Delta_{\ell-q-1}] \big\}.$$

Informally, for $v \in P^{(q)}_{\ell,j}$, there exists a nearby vertex u which is currently stuck in phase $\ell - q$. In order to bound $|H^+_\ell(j)|$, we will bound the size of $P^{(q)}_{\ell,j}$ separately as follows.

▶ **Lemma 14.** *With high probability, the following upper bounds hold.*

1. $|P^{(0)}_{\ell,j}| \le n/\Delta_{\ell-1}^{(s-1)d_i}$.
2. *For* $1 \le q \le \ell-1$, $|P^{(q)}_{\ell,j}| \le |H_{\ell-q}(j-1)| \cdot \Delta_{\ell-q-1}^{(sd_i+3)r}$.

Proof of Item 1. For $v^+ \in P^{(0)}_{\ell,j} \subseteq H^+_\ell(j)$, there exists $v \in H_\ell(j)$ such that $\mathrm{dis}_{G(j)}(v, v^+) \le r$. Suppose for contradiction that there exists $u^+ \in H^+_{\ell-1}(j-1)$ satisfying $\mathrm{dis}_{G(j-1)}(u^+, v) \le sd_i r$. Then there exists $u \in H_{\ell-1}(j-1)$ with $\mathrm{dis}_{G(j-1)}(u, u^+) \le r$, implying that $\mathrm{dis}_{G(j-1)}(u, v^+) \le \mathrm{dis}_{G(j-1)}(u, u^+) + \mathrm{dis}_{G(j-1)}(u^+, v) + \mathrm{dis}_{G(j-1)}(v, v^+) \le r + sd_i r + r = (sd_i + 2)r$. However, since $\deg_{G(j-1)}(u) > \Delta_{\ell-1}$, this contradicts $D(v^+) \le \Delta_{\ell-1}$. Hence such u^+ does not exist.

Then, by Property 6 Item 1, this implies that the state of v after sd_i executions of the LOCAL algorithm in phase ℓ is already determined. By the property of the LOCAL algorithm, the number of such vertices $v \in H_\ell(j)$ is at most $n/\Delta_{\ell-1}^{sd_i}$.

Note that v^+ is connected to v in graph $G(j)$ by a path of length at most r, and previous discussion implies that every vertex w on this path satisfies $w \notin H^+_{\ell-1}(j-1)$ and thus $\deg_{G(j)}(w) \le \Delta_{\ell-1}$. So the number of such v^+ is at most $(n/\Delta_{\ell-1}^{sd_i}) \cdot \Delta_{\ell-1}^{r+1} \le n/\Delta_{\ell-1}^{(s-1)d_i}$. ◀

Proof of Item 2. For $v \in P^{(q)}_{\ell,j}$, let $u \in G(j-1)$ be the maximizer in the definition of $D_{\ell,j}(v)$. Then v is connected to u in graph $G(j-1)$ by a path of length at most $(sd_i+2)r$, on which every vertex w (including u, v) has degree $\deg_{G(j-1)}(w) \le \deg_{G(j-1)}(u) \le \Delta_{\ell-m-1}$. Fixing a vertex u, the number of vertices v that can be reached from u in this way is at most $\Delta_{\ell-q-1}^{(sd_i+2)r+1} \le \Delta_{\ell-q-1}^{(sd_i+3)r}$. The number of vertices $u \in G(j-1)$ with $\deg_{G(j-1)}(u) \in (\Delta_{\ell-q}, \Delta_{\ell-q-1}]$ is at most $|H_{\ell-q}(j-1)|$. Hence, we conclude that $|P^{(q)}_{\ell,j}| \le |H_{\ell-q}(j-1)| \cdot \Delta_{\ell-q-1}^{(sd_i+3)r}$. ◀

Proof of Lemma 11. To show that $|H^+_\ell(j)| \le 2n/\Delta_{\ell-1}^{(s-1)d_i}$, we prove by induction on ℓ.

For $\ell = 1$, by Lemma 14, we have $|H^+_1(j)| = |P^{(0)}_{1,j}| \le n/\Delta_0^{(s-1)d_i} \le 2n/\Delta_{\ell-1}^{(s-1)d_i}$.

For $\ell \geq 2$, by Lemma 14 and induction hypothesis,

$$\begin{aligned}|H_\ell^+(j)| = |P_{\ell,j}^{(0)}| + \sum_{1\leq q<\ell} |P_{\ell,j}^{(q)}| &\leq \frac{n}{\Delta_{\ell-1}^{(s-1)d_i}} + \sum_{1\leq q<\ell} |H_{\ell-q}^+(j-1)| \cdot \Delta_{\ell-q-1}^{(sd_i+3)r} \\ &\leq \frac{n}{\Delta_{\ell-1}^{(s-1)d_i}} + \sum_{1\leq q<\ell} \frac{2n}{\Delta_{\ell-q-1}^{(s-1)d_{i+qt-1}}} \cdot \Delta_{\ell-q-1}^{(sd_i+3)r} \\ &\leq \frac{n}{\Delta_{\ell-1}^{(s-1)d_i}} + \sum_{1\leq q<\ell} \frac{2n}{\Delta_{\ell-1}^{d_i((s-1)2^{qt-1}-(s+2)r)/2^q}} \\ &\leq \frac{n}{\Delta_{\ell-1}^{(s-1)d_i}} + \sum_{1\leq q<\ell} \frac{2n}{\Delta_{\ell-1}^{(s-1)d_i q+1}},\end{aligned}$$

where in the second inequality we used the induction hypothesis and $j-1 = (\ell-q-1)t + i + qt - 1$, and in the last inequality we chose t to be a big enough constant. Then, assuming $\Delta_{\ell-1} \geq 4$, we conclude $|H_\ell^+(j)| \leq 2n/\Delta_{\ell-1}^{(s-1)d_i}$. ◀

Now we turn to analyzing the memory used for storing the d_i-hop neighborhood of $v \in H_\ell^+(j)$ (together with their communication history required by Property 6 Item 2).

▶ **Lemma 15.** *For $v \in P_{\ell,j}^{(q)}$ $(0 \leq q < \ell)$, the memory for storing the d_i-hop neighborhood of v in graph $H_\ell^+(j)$ is at most $\Delta_{\ell-q-1}^{d_i+2} \cdot \operatorname{polylog} n$.*

Proof. Since $v \in P_{\ell,j}^{(q)}$, we have $D_{\ell,j}(v) \leq \Delta_{\ell-q-1}$. By the definition of $D_{\ell,j}(v)$, every vertex u in the d_i-hop neighborhood of v in $H_\ell^+(j)$ has $\deg_{H_\ell^+(j)}(u) \leq \deg_{G(j-1)}(u) \leq D_{\ell,j}(v) \leq \Delta_{\ell-q-1}$, and thus the number of vertices u in the d_i-hop neighborhood of v is at most $\Delta_{\ell-q-1}^{d_i+1}$. Hence the total memory for storing this neighborhood (together with their communication history in phase ℓ) is at most $\Delta_{\ell-q-1}^{d_i+1} \cdot (\Delta_{\ell-1} \operatorname{polylog} n) \leq \Delta_{\ell-q-1}^{d_i+2} \cdot \operatorname{polylog} n$. ◀

We prove that this memory does not violate the memory constraints.

▶ **Theorem 16** (Local memory constraint)**.** *With high probability, for any $v \in H_\ell^+(j)$, the memory for storing the d_i-hop neighborhood of v in graph $H_\ell^+(j)$ is at most n^δ.*

Proof. Let q such that $v \in P_{\ell,j}^{(q)}$. Such v exists only if $|P_{\ell,j}^{(q)}| \geq 1$. We consider two cases.

1. If $q = 0$, by Lemma 14 and $|P_{\ell,j}^{(0)}| \geq 1$ we have $n/\Delta_{\ell-1}^{(s-1)d_i} \geq 1$. Then by Lemma 15, the memory is $\Delta_{\ell-1}^{d_i+2} \cdot \operatorname{polylog} n \leq n^{(d_i+2)/((s-1)d_i)} \cdot \operatorname{polylog} n \ll n^\delta$, where the last inequality follows from $s \geq 10/\delta$ and $d_i \geq 2$.
2. If $1 \leq q < \ell$, by Lemma 14 and $|P_{\ell,j}^{(q)}| \geq 1$ we have $|H_{\ell-q}(j-1)| \geq 1$, which then implies $2n/\Delta_{\ell-q-1}^{(s-1)d_{i+qt-1}} \geq 1$ by Lemma 11. Then similarly by Lemma 15, the memory requirement is $\Delta_{\ell-q-1}^{d_i+2} \cdot \operatorname{polylog} n \leq (2n)^{(d_i+2)/((s-1)d_{i+qt-1})} \cdot \operatorname{polylog} n \ll n^\delta$. ◀

▶ **Theorem 17** (Global memory constraint)**.** *With high probability, the global memory for storing the neighborhoods is $\widetilde{O}(n)$.*

Proof. Since there are only $L = O(\log\log\Delta)$ phases running concurrently, it suffices to show that, for every $\ell \in [L]$, the total memory for storing the d_i-hop neighborhoods of all vertices v in graph $H_\ell^+(j)$ is at most $\widetilde{O}(n)$, where $j = (\ell-1)t + i$.

By Lemma 15, the total memory can be bounded by

$$|P_{\ell,j}^{(0)}| \cdot \Delta_{\ell-1}^{d_i+2} \cdot \operatorname{polylog} n + \sum_{1\leq q<\ell} |P_{\ell,j}^{(q)}| \cdot \Delta_{\ell-q-1}^{d_i+2} \cdot \operatorname{polylog} n$$

$$\leq \frac{n}{\Delta_{\ell-1}^{(s-1)d_i}} \cdot \Delta_{\ell-1}^{d_i+2} \cdot \text{polylog}\, n + \sum_{1 \leq q < \ell} |H_{\ell-q}^{+}(j-1)| \cdot \Delta_{\ell-q-1}^{(sd_i+3)r} \cdot \Delta_{\ell-q-1}^{d_i+2} \cdot \text{polylog}\, n$$
$$\leq \widetilde{O}(n),$$

where the last inequality follows from a similar calculation as in the proof of Lemma 11. ◀

3 4-coloring of trees in $O(\log \log n)$ rounds

In this section, we present an algorithm that colors a tree with 4 colors in $O(\log \log n)$ MPC rounds and $O(m)$ global memory, thus providing a proof for Theorem 2. We start with a high-level overview, and then fill in the details of the algorithm and its analysis.

High-Level Overview. Our 4-coloring algorithm starts by randomly partitioning the vertices of the input tree into two sets. Each set induces a forest such that each connected component has a diameter of $O(\log n)$, with high probability. Each connected component corresponds to a tree and will be rooted (i.e., orienting each edge towards the root) by using $\Theta(\log^2 n)$ parallel black-box invocations to the connected components algorithm of Behnezhad et al. [11] (which improved on the earlier work by Andoni et al. [2]). We note that this connected components algorithm runs $O(\log D + \log \log n)$ rounds, where D denotes the maximum component diameter, using strongly sublinear local memory and $O(m)$ global memory. Once the tree is rooted, computing a 2-coloring in $O(\log \log n)$ rounds is easy: we can learn the distance from the root, by pointer jumping along the outgoing edges and leveraging that the diameter is bounded by $O(\log n)$, and then 2-color nodes based on odd or even distances. Thus, each of the two forests can be colored with 2 colors. This results in a 4-coloring of the complete tree. The presented algorithm would need a slightly superlinear global memory of $\Omega(n \log^2 n)$. To alleviate this issue and work with only $O(n)$ global memory, our actual algorithm first reduces the number of vertices to $O(n/\text{poly}(\log n))$ by using the well-known peeling algorithm [4], which in $O(\log \log n)$ iterations repeatedly removes vertices of degree at most 2. These removed vertices are then colored at the very end of the algorithm, after coloring those $O(n/\text{poly}(\log n))$ remaining vertices, in $\Theta(\log \log n)$ additional MPC rounds.

3.1 Details + Analysis

In this section, we explain and analyze the algorithm in detail. Let T denote the input tree.

3.1.1 Reducing the number of vertices

First, we explain and analyze the peeling algorithm which reduces the number of vertices to $O(n/\log^2 n)$. The peeling algorithm consists of $N = \Theta(\log \log n)$ iterations. In each iteration, we remove all vertices that have at most 2 neighbors in the current graph. In each iteration of the peeling algorithm, a constant fraction of the remaining vertices are removed.

▶ **Lemma 18.** *Let $G = (V, E)$ be a forest. Then, $|\{v \in V : \deg(v) \geq 3)\}| \leq \frac{2}{3}|V|$.*

Proof. We have $3 \cdot |\{v \in V : \deg(v) \geq 3)\}| \leq \sum_{v \in V} \deg(v) \leq 2|E| \leq 2|V|$, where the last inequality follows, as each forest has at most $|V| - 1$ edges and the second-last inequality is the well-known handshaking lemma. ◀

Thus, after iteration i of the peeling algorithm, the number of remaining vertices is at most $n \cdot (2/3)^i$. Hence, after running $N = \lceil 2 \log_{3/2} \log n \rceil$ iterations of the peeling algorithm, the number of remaining vertices is at most $O(n/\log^2(n))$. Note that each iteration of the peeling algorithm can easily be implemented in $O(1)$ MPC rounds and linear global memory.

Next, we explain how to color the vertices removed by the peeling algorithm, once all the remaining vertices are colored, using no additional colors. The idea is well-known in the context of LOCAL algorithms. For $i \in [N]$, let W_i denote the set of vertices removed in the i-th iteration. The basic idea is to first color all vertices in W_N, then the ones in W_{N-1} and so on. By coloring the vertices in that order, the number of neighbors that previously got assigned a color is upper bounded by 2. Hence, we can assign each node one of the first 3 colors without creating any conflict. In order for this procedure to be efficient, we need to color multiple nodes simultaneously. However, coloring all nodes in W_i in parallel is problematic, as W_i may contain neighboring nodes. To circumvent this problem, we temporarily color $T[W_i]$ with a constant number of colors – these are not a part of the output coloring but merely used as a schedule in computing the output coloring. In fact, we can color all of $T[W_1], T[W_2], \dots, T[W_N]$ simultaneously, each separately using constant many colors, in $O(\log^* n)$ MPC rounds. This is done by simulating the LOCAL algorithm of Cole and Vishkin [15]. Then, to compute the output coloring, for each $T[W_i]$, we iterate through the constant many schedule colors in $T[W_i]$, and we compute the output color of all the nodes that got assigned the same schedule color in parallel. Hence, one never assigns two neighboring nodes an output color simultaneously. Coloring the nodes of one schedule color class in W_i can be simulated in $O(1)$ MPC rounds. Thus, we can color all the deleted vertices in $O(1) \cdot O(1) \cdot O(\log \log n)$ MPC rounds and using linear global memory. Hence, what remains is to show how to color a forest consisting of $O(n/\log^2(n))$ nodes in $O(\log \log n)$ MPC rounds, using $O(n)$ global memory.

3.1.2 Random Partitioning

In order for our procedure to be efficient, we need each connected component under consideration to have a diameter of $O(\log n)$. We achieve this by randomly partitioning the remaining vertices into two sets V_1 and V_2, with each remaining vertex being in either one of them with a probability of $1/2$, independent of the other vertices. The partitioning leads to each connected component of $T[V_1]$ and $T[V_2]$ having a diameter of $O(\log n)$, with high probability. The argument is simple: each path of length $\omega(\log n)$ is present in $T[V_1]$ or $T[V_2]$ with a probability less than $1/\operatorname{poly}(n)$. Then, we can union bound over the at most $O(n^2)$ distinct paths in the tree to conclude that no path of length $\omega(\log n)$ will be present in $T[V_1]$ or $T[V_2]$, with probability $1 - 1/\operatorname{poly}(n)$.

3.1.3 Handling connected components in parallel

In the next paragraph, we discuss an algorithm that computes a 2-coloring of a given connected component in $O(\log \log n)$ rounds. The algorithm works under the assumption that each node of the component has a unique identifier in $[\eta]$, with η denoting the size of the connected component. Furthermore, it assumes that the edges of the connected component are the only input given to the algorithm. It requires each machine to have a local memory of $O(\min(\eta, n^\delta))$ and uses $O(\eta \log^2 n)$ global memory. Before going into the details of how the algorithm works, we first argue why the existence of such an algorithm readily implies that we can 2-color each connected component in $T[V_1]$ and $T[V_2]$ with 2 colors in $O(\log \log n)$ MPC rounds and $O(m)$ global memory.

Before 2-coloring each connected component, we start by computing the connected components of both $T[V_1]$ and $T[V_2]$ by using the connected component algorithm of [11]. The algorithm runs in $O(\log \log n + \log D)$ rounds, where D denotes the maximal component diameter. We can assume $D = O(\log n)$. Thus, we can find the connected components of both

$T[V_1]$ and $T[V_2]$ in $O(\log\log n)$ rounds. After having computed the connected components, we create one tuple per node with the first entry being equal to its component identifier and the second entry being equal to the identifier of the node itself. By sorting these tuples lexicographically in $O(1)$ rounds using the algorithm of [23], we can identify the size of each connected component. Furthermore, one can assign each connected component a number of machines proportional to its size, such that the total memory capacity of the assigned machines is $O(n)$. That is, each large connected component with $\eta \geq n^\delta$ nodes gets assigned $\Theta(\log^2(n)\eta/n^\delta)$ many machines. After relabeling the vertices of the connected component with unique identifiers between 1 and η, the machines assigned to such a large component can 2-color the component in $O(\log\log n)$ rounds. Each small component with less than n^δ vertices is stored on one machine, which may be responsible for multiple small components. A 2-coloring of such a small component can be computed locally on the corresponding machine.

3.1.4 Rooting and 2-coloring a tree with diameter $O(\log n)$

Next, we focus on a single connected component in either $T[V_1]$ or $T[V_2]$ and show how to root the connected component by using $O(\log^2 n)$ invocations of the connected component algorithm in parallel. As stated in the previous paragraph, we assume that each node of the connected component has a unique identifier in $[\eta]$, with η denoting the size of the component. We pick an arbitrary node as the root, i.e., the root with identifier 1. We want each node, except for the root, to learn which neighboring node is its parent. Our algorithm relies on the following simple, but crucial observation: the parent of a node is the only neighboring node that, when deleting a subset of the edges in the tree, can still remain in the same connected component as the root, while the node itself got disconnected from the root.

To make use of this observation, we remove each edge independently with a probability of $1/\log(n)$. Let v be an arbitrary node. Notice that if v is disconnected from the root but a neighbor u of v is connected to the root, then one can deduce that u is the parent of v. Let $p(v)$ be the parent of v. Node $p(v)$ remains in the same connected component as the root if all of the at most $O(\log n)$ edges on the path from $p(v)$ to the root remain in the graph. This happens with a probability of at least $(1-1/\log n)^{O(\log n)} = \Omega(1)$. Conditioning on this event, node v gets disconnected from the root if the edge between v and $p(v)$ gets removed, which happens with a probability of $1/\log(n)$. If both of these happen, we have a good event and v can identify its parent $p(v)$. Thus, we can conclude that a fixed vertex v can determine its parent with a probability of $\Omega(1/\log(n))$.

By running this procedure $\Theta(\log^2 n)$ times in parallel (and independently), we can conclude – by a Chernoff bound and a union bound over all vertices – that each vertex determines its parent with high probability. Hence, we can 2-color the connected component using $O(\min(\eta, n^\delta))$ memory per machine and $O(\log^2(n)\eta)$ global memory.

Once we have this orientation towards the root, and since the diameter of the tree is $O(\log n)$, each node can compute the distance of it from the root using $\Theta(\log\log n)$ steps of pointer jumping, as we explain next.

▶ **Lemma 19.** *Given a tree of depth $O(\log n)$ with η nodes, which is oriented towards the root, we can compute the distance of each node from the root in $O(\log\log n)$ rounds, using $O(\min(\eta, n^\delta))$ local memory and $O(\eta\log^2 n)$ global memory.*

Proof. We use a *pointer jumping* idea, along with some sorting subroutines of MPC. For each node v, we keep track of one pointer $p(v)$ which we initially set equal to its parent node (for the root r the pointer points to the node itself). Also, we define the ancestor list $AL(v)$, which is a set that initially only contains the node v.

Then, each iteration of pointer jumping consists of two steps: first, each node v updates its ancestor list as $AL(v) \leftarrow AL(v) \cup AL(p(v))$. Then, in the second step, we redefine $p(v) \leftarrow p(p(v))$. It is clear that after $O(\log \log n)$ iterations, $p(v)$ is equal to the root of its component and $AL(v)$ is equal to the set of all of its ancestors, out of which v learns its distance to the root.

Implementing the pointer jumping needs some care. In each iteration of pointer jumping, each node v needs to learn the value of $p(u)$, where $u = p(v)$. While node v wants to learn only one value, node u might have to inform many nodes about the value of $p(u)$. Essentially the same procedure needs to be done for learning $AL(u)$, so we focus on $p(u)$.

We use a basic sorting subroutine [23], which runs in constant time. For the purpose of sorting, for each node v, define an item $\langle p(v), v\rangle$. Here, v and $p(v)$ are the respective identifiers, which are numbers in $\{1, \ldots, \eta\}$. Moreover, add for each node v two extra items $\langle v, -\infty\rangle$ and $\langle v, +\infty\rangle$. Sort all these items lexicographically. At the end of the sorting, each machine that holds an item knows the rank of this item in the sorted list.

Once we have the items sorted, for each node u, in the sorted order, the items start with $\langle u, -\infty\rangle$, end with $\langle u, +\infty\rangle$, and in between these two are the entries of all nodes v such that $p(v) = u$. As a result, the machine that holds node u and generated the two items $\langle u, -\infty\rangle$ and $\langle u, +\infty\rangle$ knows the ranks of these two items in the sorted order.

Now, split the items among the machines, in an ordered way, so that the ith machine receives the items with rank $[S(i-1)+1, Si]$. Finally, use a broadcast tree of constant depth [23] so that the machine that holds node u informs the machines that, in the sorted order of items, hold items with u as the first entry, about the value of $p(u)$. This way, any machine that holds an item $\langle p(v), v\rangle$ where $p(v) = u$ learns $p(u)$. Hence, that machine can add this value $p(u)$ as a third field to create $\langle p(v), v, p(u)\rangle$. At the very end, we send these three-entry messages back to the machine that initially held v. Hence, node v now knows the value of $p(u)$ where $u = p(v)$.

The same process can be used so that node v learns $AL(p(u))$, where $u = p(v)$. In that case, the message has $O(\log n)$ words – the maximum number of ancestors in an $O(\log n)$ depth tree. The algorithm uses $O(\min(\eta, n^{\delta}))$ local memory and $O(\eta \log^2 n)$ global memory, thus proving the lemma. ◀

Once the nodes know their distance from their respective root, a 2-coloring is immediate: nodes at odd distances get one color and nodes at even distances get the other. This gives a separate 2-coloring of each of $T[V_1]$ and $T[V_2]$. Overall, this leads to a 4-coloring of $T[V_1 \cup V_2]$.

Remark about a work of Brandt et al. [14]. The main result of Brandt et al. [14] is a $\Theta(\log^3 \log n)$ algorithm for finding a Maximal Independent Set on trees. As a subroutine, they used a simple, deterministic (graph exponentiation style) algorithm to root a tree with a diameter of D in $O(\log D \log \log n)$ rounds, under the assumption of initially having $\Omega(D^3)$ global memory per node. Using their rooting algorithm as a black-box in our approach, we can directly improve this complexity. Namely, after the step of reducing the number of vertices to $O(n/\operatorname{polylog} n)$, as explained in Section 3.1.1, and randomly splitting the vertices into two sets V_1 and V_2, as explained in Section 3.1.2, instead of our proposed rooting algorithm, one could invoke the rooting procedure of Brandt et al. [14], which runs in $\Theta(\log^2 \log n)$ rounds. Overall, this gives a 4-coloring in $O(\log^2 \log n)$ rounds. Having this coloring, we can easily solve MIS in $O(1)$ additional rounds, simply by iterating through color classes of nodes and greedily adding nodes to the MIS.

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. doi:10.1016/0196-6774(86)90019-2.
2. Alexandr Andoni, Zhao Song, Clifford Stein, Zhengyu Wang, and Peilin Zhong. Parallel graph connectivity in log diameter rounds. In *Proceedings of the 59th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 674–685, 2018. doi:10.1109/FOCS.2018.00070.
3. Sepehr Assadi, MohammadHossein Bateni, Aaron Bernstein, Vahab Mirrokni, and Cliff Stein. Coresets meet EDCS: Algorithms for matching and vertex cover on massive graphs. In *Proceedings of the 30th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1616–1635, 2019. doi:10.1137/1.9781611975482.98.
4. Leonid Barenboim and Michael Elkin. Sublogarithmic Distributed MIS Algorithm for Sparse Graphs Using Nash-Williams Decomposition. *Distributed Computing*, 22(5-6):363–379, 2010. doi:10.1007/s00446-009-0088-2.
5. Leonid Barenboim, Michael Elkin, Seth Pettie, and Johannes Schneider. The locality of distributed symmetry breaking. *Journal of the ACM*, 63(3):20:1–20:45, 2016. doi:10.1145/2903137.
6. MohammadHossein Bateni, Soheil Behnezhad, Mahsa Derakhshan, MohammadTaghi Hajiaghayi, and Vahab S. Mirrokni. Brief announcement: Mapreduce algorithms for massive trees. In *Proceedings of the 45th International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 162:1–162:4, 2018. doi:10.4230/LIPIcs.ICALP.2018.162.
7. MohammadHossein Bateni, Soheil Behnezhad, Mahsa Derakhshan, MohammadTaghi Hajiaghayi, and Vahab S. Mirrokni. Massively parallel dynamic programming on trees. *arXiv preprint*, 2018. arXiv:1809.03685.
8. Paul Beame, Paraschos Koutris, and Dan Suciu. Skew in Parallel Query Processing. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, pages 212–223, 2014. doi:10.1145/2594538.2594558.
9. Soheil Behnezhad, Sebastian Brandt, Mahsa Derakhshan, Manuela Fischer, MohammadTaghi Hajiaghayi, Richard M. Karp, and Jara Uitto. Massively parallel computation of matching and MIS in sparse graphs. In *Proceedings of the 38th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 481–490, 2019. doi:10.1145/3293611.3331609.
10. Soheil Behnezhad, Mahsa Derakhshan, MohammadTaghi Hajiaghayi, and Richard M. Karp. Massively Parallel Symmetry Breaking on Sparse Graphs: MIS and Maximal Matching. *arXiv preprint*, 2018. arXiv:1807.06701.
11. Soheil Behnezhad, Laxman Dhulipala, Hossein Esfandiari, Jakub Łącki, and Vahab Mirrokni. Near-optimal massively parallel graph connectivity. In *Proceedings of the 60th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 1615–1636, 2019. doi:10.1109/FOCS.2019.00095.
12. Soheil Behnezhad, MohammadTaghi Hajiaghayi, and David G. Harris. Exponentially faster massively parallel maximal matching. In *Proceedings of the 60th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 1637–1649, 2019. doi:10.1109/FOCS.2019.00096.
13. Sebastian Brandt, Manuela Fischer, and Jara Uitto. Matching and MIS for uniformly sparse graphs in the low-memory MPC model. *arXiv preprint*, 2018. arXiv:1807.05374.
14. Sebastian Brandt, Manuela Fischer, and Jara Uitto. Breaking the linear-memory barrier in MPC: Fast MIS on trees with strongly sublinear memory. In *Proceedings of the 26th International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, pages 124–138, 2019. doi:10.1007/978-3-030-24922-9_9.
15. Richard Cole and Uzi Vishkin. Deterministic coin tossing and accelerating cascades: Micro and macro techniques for designing parallel algorithms. In *Proceedings of the 18th ACM Symposium on Theory of Computing*, pages 206–219, 1986.

16 Artur Czumaj, Peter Davies, and Merav Parter. Graph sparsification for derandomizing massively parallel computation with low space. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 175–185, 2020. doi:10.1145/3350755.3400282.

17 Artur Czumaj, Jakub Łącki, Aleksander Mądry, Slobodan Mitrović, Krzysztof Onak, and Piotr Sankowski. Round compression for parallel matching algorithms. *SIAM Journal on Computing*, pages STOC18–1–STOC18–44, 2019. doi:10.1137/18M1197655.

18 Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008. doi:10.1145/1327452.1327492.

19 Mohsen Ghaffari. Distributed MIS via all-to-all communication. In *Proceedings of the 36th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 141–149, 2017. doi:10.1145/3087801.3087830.

20 Mohsen Ghaffari, Themis Gouleakis, Christian Konrad, Slobodan Mitrović, and Ronitt Rubinfeld. Improved massively parallel computation algorithms for MIS, matching, and vertex cover. In *Proceedings of the 37th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 129–138, 2018. doi:10.1145/3212734.3212743.

21 Mohsen Ghaffari, Fabian Kuhn, and Jara Uitto. Conditional hardness results for massively parallel computation from distributed lower bounds. In *Proceedings of the 60th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 1650–1663, 2019. doi:10.1109/FOCS.2019.00097.

22 Mohsen Ghaffari and Jara Uitto. Sparsifying distributed algorithms with ramifications in massively parallel computation and centralized local computation. In *Proceedings of the 30th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1636–1653, 2019. doi:10.1137/1.9781611975482.99.

23 Michael T. Goodrich, Nodari Sitchinava, and Qin Zhang. Sorting, Searching, and Simulation in the MapReduce Framework. In *Proceedings of the 22nd International Symposium on Algorithms and Computation*, pages 374–383, 2011. doi:10.1007/978-3-642-25591-5_39.

24 Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 59–72, 2007. doi:10.1145/1272996.1273005.

25 Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. A Model of Computation for MapReduce. In *Proceedings of the 21st ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 938–948, 2010. doi:10.1137/1.9781611973075.76.

26 Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. Local computation: Lower and upper bounds. *Journal of the ACM*, 63(2):17:1–17:44, 2016. doi:10.1145/2742012.

27 Silvio Lattanzi, Benjamin Moseley, Siddharth Suri, and Sergei Vassilvitskii. Filtering: A method for solving graph problems in MapReduce. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 85–94, 2011. doi:10.1145/1989493.1989505.

28 Christoph Lenzen and Roger Wattenhofer. Brief Announcement: Exponential Speed-Up of Local Algorithms Using Non-Local Communication. In *Proceedings of the 29th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 295–296, 2010. doi:10.1145/1835698.1835772.

29 Nathan Linial. Locality in distributed graph algorithms. *SIAM Journal on Computing*, 21(1):193–201, 1992. doi:10.1137/0221015.

30 Michael Luby. A Simple Parallel Algorithm for the Maximal Independent Set Problem. *SIAM Journal on Computing*, 15(4):1036–1053, 1986. doi:10.1137/0215074.

31 Crispin St. John Alvah Nash-Williams. Decomposition of finite graphs into forests. *J. London Math. Soc.*, 39(1):12, 1964. doi:10.1112/jlms/s1-39.1.12.

32 Tim Roughgarden, Sergei Vassilvitskii, and Joshua R. Wang. Shuffles and circuits (on lower bounds for modern parallel computation). *Journal of the ACM*, 65(6):41:1–41:24, 2018. doi:10.1145/3232536.

33 Tom White. *Hadoop: The Definitive Guide*. O'Reilly, 2012.

34 Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, 2010.

Improved Distributed Approximations for Maximum Independent Set

Ken-ichi Kawarabayashi
National Institute of Informatics, Tokyo, Japan
k_keniti@nii.ac.jp

Seri Khoury
University of California, Berkeley, CA, USA
seri_khoury@berkeley.edu

Aaron Schild
University of Washington, Seattle, WA, USA
aschild@uw.edu

Gregory Schwartzman
JAIST, Ishikawa, Japan
gregory.schwartzman@gmail.com

Abstract

We present improved results for approximating maximum-weight independent set (MaxIS) in the CONGEST and LOCAL models of distributed computing. Given an input graph, let n and Δ be the number of nodes and maximum degree, respectively, and let $\text{MIS}(n, \Delta)$ be the running time of finding a *maximal* independent set (MIS) in the CONGEST model. Bar-Yehuda et al. [PODC 2017] showed that there is an algorithm in the CONGEST model that finds a Δ-approximation for MaxIS in $O(\text{MIS}(n, \Delta) \log W)$ rounds, where W is the maximum weight of a node in the graph, which can be as large as $\text{poly}(n)$. Whether their algorithm is deterministic or randomized that succeeds with high probability depends on the MIS algorithm that is used as a black-box. Our results:

1. A deterministic $O(\text{MIS}(n, \Delta)/\epsilon)$-round algorithm that finds a $(1 + \epsilon)\Delta$-approximation for MaxIS in the CONGEST model.
2. A randomized $(\text{poly}(\log \log n)/\epsilon)$-round algorithm that finds, with high probability, a $(1 + \epsilon)\Delta$-approximation for MaxIS in the CONGEST model. That is, by sacrificing only a tiny fraction of the approximation guarantee, we achieve an *exponential* speed-up in the running time over the previous best known result.
3. A randomized $O(\log n \cdot \text{poly}(\log \log n)/\epsilon)$-round algorithm that finds, with high probability, a $8(1 + \epsilon)\alpha$-approximation for MaxIS in the CONGEST model, where α is the arboricity of the graph. For graphs of arboricity $\alpha < \Delta/(8(1 + \epsilon))$, this result improves upon the previous best known result in both the approximation factor and the running time.

One may wonder whether it is possible to approximate MaxIS with high probability in fewer than $\text{poly}(\log \log n)$ rounds. Interestingly, a folklore randomized ranking algorithm by Boppana implies a single round algorithm that gives an expected Δ-approximation in the CONGEST model. However, it is unclear how to convert this algorithm to one that succeeds with high probability without sacrificing a large number of rounds. For unweighted graphs of maximum degree $\Delta \leq n/\log n$, we show a new analysis of the randomized ranking algorithm, which we combine with the local-ratio technique, to provide a $O(1/\epsilon)$-round algorithm in the CONGEST model that, with high probability, finds an independent set of size at least $\frac{n}{(1+\epsilon)(\Delta+1)}$. This result cannot be extended to very high degree graphs, as we show a lower bound of $\Omega(\log^* n)$ rounds for any randomized algorithm that with probability at least $1 - 1/\log n$ finds an independent set of size $\Omega(n/\Delta)$. This lower bound holds even for the LOCAL model. The hard instances that we use to prove our lower bound are graphs of maximum degree $\Delta = \Omega(n/\log^* n)$.

2012 ACM Subject Classification Mathematics of computing → Approximation algorithms; Theory of computation → Distributed computing models

Keywords and phrases Distributed graph algorithms, Approximation algorithms, Lower bounds

© Ken-ichi Kawarabayashi, Seri Khoury, Aaron Schild, and Gregory Schwartzman;
licensed under Creative Commons License CC-BY
34th International Symposium on Distributed Computing (DISC 2020).
Editor: Hagit Attiya; Article No. 35; pp. 35:1–35:16
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.35

Related Version A full version of the paper is available at https://arxiv.org/abs/1906.11524.

Funding *Ken-ichi Kawarabayashi*: Supported by JST ERATO Kawarabayashi Large Graph Project JPMJER1201 and by JSPS Kakenhi JP18H05291.
Gregory Schwartzman: Supported by JSPS Kakenhi Grant Number JP19K20216 and JP18H052.

1 Introduction and Related Work

In this work we study the problem of approximating Maximum Independent Set (MaxIS) in distributed models. An independent set in a network is a subset of the nodes such that no two nodes in the subset are adjacent. In unweighted graphs, a *maximum* independent set is an independent set of maximum size. In weighted graphs, a *maximum-weight* independent set (MaxIS) is an independent set of maximum total weight, where by total we mean the sum of weights of nodes in the independent set.

The major two models of distributed graph algorithms are the LOCAL and CONGEST models. In the LOCAL model [19], there is a synchronized communication network of n computationally-unbounded nodes, where each node has a unique $O(\log n)$-bit identifier. In each communication round, each node can send an unbounded-size message to each of its neighbors. The task of the nodes is to compute some function of the network (e.g., its diameter, the value of a maximum independent set, etc.), while minimizing the number of communication rounds. The CONGEST model [21] is similar to the LOCAL model, where the only difference is that the message-size is bounded by $O(\log n)$ bits.

The problem of approximating MaxIS has been studied in both the LOCAL and CONGEST models [2, 7, 9, 11, 15, 16, 18]. In unweighted graphs, one can find a Δ-approximation for MaxIS by finding a maximal independent set (MIS). In recent years, our understanding of the complexity of MIS has been substantially improving [6, 12, 13, 23], leading to a recent remarkable breakthrough by Rozhon and Ghaffari [23], where they show a deterministic $\text{poly}(\log n)$-round algorithm for finding an MIS, even in the CONGEST model. This result also implies a randomized algorithm that finds an MIS with high probability in $O(\log \Delta) + \text{poly}(\log \log n)$ rounds, in the CONGEST model[1] [10, 13, 14, 23].

In a weighted graph, an MIS doesn't necessarily constitute a Δ-approximation for MaxIS. For the weighted case, Bar-Yehuda et al. [4] showed a Δ-approximation algorithm in the CONGEST model that takes $O(\text{MIS}(n, \Delta) \cdot \log W)$ rounds, where $\text{MIS}(n, \Delta)$ is the running time for finding an MIS in graphs with n nodes and maximum degree Δ, and W is the maximum weight of a node in the graph (which can be as high as $\text{poly}(n)$). Whether their algorithm is deterministic or randomized that succeeds with high probability depends on the MIS algorithm that is used as a black-box.

In this work we present faster algorithms compared to [4], by paying only a $(1 + \epsilon)$ multiplicative overhead in the approximation factor. Our main result (Theorem 2) is a randomized algorithm that achieves an exponential speed-up compared to [4]. One of the ingredients to prove our main result is an improved algorithm for the deterministic case (Theorem 1). Our results:

[1] We say that an algorithm succeeds with high probability if it succeeds with probability $1 - 1/n^c$ for an arbitrary constant $c > 1$.

▶ **Theorem 1.** *There is an $O(\text{MIS}(n, \Delta)/\epsilon)$-round algorithm in the CONGEST model that finds a $(1+\epsilon)\Delta$-approximation for maximum-weight independent set. Whether the algorithm is deterministic or randomized, depends on the* MIS *algorithm that is run as a black-box.*

▶ **Theorem 2.** *There is a randomized $(\text{poly}(\log \log n)/\epsilon)$-round algorithm in the CONGEST model that finds, with high probability, a $(1+\epsilon)\Delta$-approximation for maximum-weight independent set.*

Using the algorithm from Theorem 2, we can also get an improved approximation algorithm for a wide range of arboricity. Let α be the arboricity of the input graph. For graphs of arboricity $\alpha \leq \Delta/(8(1+\epsilon))$, Theorem 3 improves upon [4] in both the running time and approximation factor.

▶ **Theorem 3.** *There is a randomized $O(\log n \cdot \text{poly} \log \log n/\epsilon)$-round algorithm in the CONGEST model that finds, with high probability, an $8(1+\epsilon)\alpha$-approximation for maximum-weight independent set.*

A discussion on the folklore randomized ranking algorithm. A folklore randomized ranking algorithm by Boppana yields a single round algorithm in the CONGEST model that returns a solution with an *expected* approximation factor $\Delta + 1$.[2] In this ranking algorithm, each node v picks a number r_v uniformly at random in $[0, 1]$. If $r_v > r_u$ for any neighbor u of v, then v joins the independent set. Since every node joins the independent set with probability at least $1/(\Delta + 1)$, the expected weight of the independent set is at least $w(V)/(\Delta + 1)$, where $w(V)$ is the total weight of nodes in the graph. Recently, Boppana et al [8] showed that the ranking algorithm even returns a $(\Delta + 1)/2$-approximation in expectation. However, algorithms that work well in expectation don't necessarily work well with good probability. In fact, for the folklore randomized ranking algorithm, it is not very hard to construct examples in which the *variance* of the solution is very high, in which case the algorithm doesn't return the expected value with high probability. In this work we prove the following stronger hardness result (Theorem 4). Our lower bound in Theorem 4 holds only under the assumption that the nodes don't know the exact value of n, but only a polynomial upper bound on it. We emphasize this because some algorithms in the LOCAL and CONGEST models assume the knowledge of n (our algorithms in this work don't need this assumption).

▶ **Theorem 4.** *If the nodes don't know the exact value of n, but only a polynomial upper bound on it, then any algorithm that finds an independent set of size $\Omega(n/\Delta)$ in unweighted graphs, with success probability $p \geq 1 - 1/\log n$ must spend $\Omega(\log^* n)$ rounds, even in the LOCAL model.*

Interestingly, this hardness result applies for graphs of maximum degree $\Delta = \Omega(n/\log^* n)$. One may wonder whether we can extend the lower bound for much smaller maximum degree graphs. We rule out this possibility, with the following theorem. The proof of Theorem 5 relies on a novel idea for analyzing the classical ranking algorithm using *martingales*, and the *local-ratio* technique, on which we elaborate in the technical overview.

▶ **Theorem 5.** *For unweighted graphs of maximum degree $\Delta \leq n/\log n$, there is an $O(1/\epsilon)$-round algorithm in the CONGEST model that finds, with high probability, an independent set of size at least $\frac{n}{(1+\epsilon)(\Delta+1)}$.*

---

[2] To the best of our knowledge, this ranking algorithm has first appeared in the book of Alon and Spencer [1] and is due to Boppana (see also the references for this algorithm in [8]).

Further Related Work. Ghaffari et al. [15], showed that there is an algorithm for the LOCAL model that finds a $(1+\epsilon)$-approximation for MaxIS in $O(\text{poly}(\log n/\epsilon))$ rounds, for a constant ϵ. The results in [11, 18] give a lower bound of $\Omega(\log^* n)$ rounds for any deterministic algorithm that returns an independent set of size at least $n/\log^* n$ on a cycle, and a randomized $O(1)$-round algorithm for $O(1)$-approximations in planar graphs, in the LOCAL model. The results by [7, 16] give fast algorithms for approximating MaxIS in unweighted graphs, where the approximation guarantees are only in expectation.

Road-map. In Section 2 we provide a technical overview. Section 3 contains some basic definitions and useful inequalities. The technical heart of the paper starts in Section 4, where we prove our first two results (Theorems 1 and 2). Due to space limitations, we defer the rest of our proofs to the the full version [17].

2 Technical Overview

Results for weighted graphs. Our first two results (Theorems 1 and 2) share a similar proof structure. First, we show that there are fast algorithms for $O(\Delta)$-approximation. Then we use the *local-ratio* technique [3] to prove a general boosting theorem that takes a T-round algorithm for $O(\Delta)$-approximation, and use it as a black-box to output a $(1+\epsilon)\Delta$-approximation in $O(T/\epsilon)$ rounds. An overview of the local-ratio technique and the boosting theorem is provided in Section 2.2. The key ingredient to show a fast $O(\Delta)$-approximation algorithm is a new *weighted sparsification* technique, where we show that it suffices to find an independent set of a good approximation in a sparse subgraph. An overview of the weighted sparsification technique is provided in Section 2.1.

Our improved approximation algorithm for low-arboricity graphs (Theorem 3) uses Theorem 2 as a black-box, where the main technical ingredient is the local-ratio technique. An overview of this algorithm is also provided in Section 2.2.

Results for unweighted graphs. Our upper bound for unweighted graphs of maximum degree $\Delta \leq n/\log n$ (Theorem 5) has a similar two-step structure as the first two results. We first show an $O(\Delta)$-approximation algorithm, and then we use the local-ratio technique to boost the approximation factor. For the $O(\Delta)$-approximation part, we show that running the classical one-round ranking algorithm (that was used by [8]) for c rounds already returns an $O(\Delta)$-approximation for unweighted graphs of maximum degree $\Delta \leq n/\log n$, with probability $\approx 1 - 1/n^c$. The main technical ingredient for showing this result is a new analysis of the classical ranking algorithm using *martingales*. An overview of this result is provided in Section 2.3. Finally, in Section 2.4, we provide an overview of the lower bound result (Theorem 4).

2.1 Weighted Sparsification for $O(\Delta)$-Approximation

A good way to understand the $O(\Delta)$-approximation algorithm is to first consider the unweighted case. Let $G = (V, E)$ be an unweighted graph. We can find an $O(\Delta)$-approximation for MaxIS in G as follows. First, we sample a sparse subgraph H of G with the following properties. (1) The maximum degree Δ_H of H is small ($O(\log n)$). (2) The ratio between the number of nodes (n_H) and the maximum degree of H is at least as in G, up to a constant multiplicative factor. That is, $n_H/\Delta_H = \Omega(n/\Delta)$. Since any MIS in H has size at least $n_H/\Delta_H = \Omega(n/\Delta)$, it suffices to find an MIS in H, which takes $\text{MIS}(n_H, \Delta_H) \leq \text{MIS}(n, \log n)$ rounds (recall that $\text{MIS}(n, \Delta)$ is the running time of finding

an MIS in graphs of n nodes and maximum degree Δ). By the recent breakthrough of Rozhon and Ghaffari [23], $\text{MIS}(n, \log n) = O(\log\log n) + \text{poly}(\log\log n) = \text{poly}\log\log n$ rounds. Furthermore, sampling a subgraph with the aforementioned properties is almost trivial. Each node joins H with probability $\min\{\log n/\Delta, 1\}$, independently. It is not very hard to show, via standard Chernoff (Fact 1) and Union Bound arguments, that H has the desired properties. While this approach is straightforward for the unweighted case, it runs into challenges when trying to apply it for the weighted case, as we explain next.

The challenge in weighted graphs. Perhaps the first thing that comes into mind when trying to extend the sampling technique to weighted graphs is to try to sample a sparse subgraph H with the following properties. (1) The maximum degree $\Delta_H = O(\log n)$. (2) The ratio between the *total weight* in H and the max degree of H is the same as in G, up to a constant multiplicative factor. That is $w(V_H)/\Delta_H = \Omega(w(V)/\Delta)$, where $w(V_H)$ is the total weight of nodes in H and $w(V)$ is the total weight of nodes in G. However, this approach runs into two challenges. The first challenge is that in the weighted case, an MIS doesn't necessarily constitute a Δ-approximation for MaxIS. Therefore, even if we are able to sample a subgraph H with the desired properties, running an MIS algorithm on H might result in an independent set of a very small weight. To overcome this challenge, we show a very simple $\text{MIS}(n, \Delta)$-round algorithm that finds an $O(\Delta)$-approximation. This algorithm runs an MIS algorithm on the subgraph induced by nodes that are relatively heavy, compared to their neighbors. Specifically, a node is considered relatively heavy compared to its neighbors, if it is of weight at least $\Omega(1/\Delta)$-fraction of the sum of weights of its neighbors. It is not very hard to show that this algorithm returns an independent set of total weight $\Omega(w(V)/\Delta)$, where $w(V)$ is the total weight of nodes in the graph. The proof of this argument is provided in Section 4.1.

Furthermore, another challenge is that the same sampling procedure doesn't work for the weighted case. In particular, if we sample each node with probability $p = \min\{(\log n)/\Delta, 1\}$, then *light*-weight nodes will have the same probability of joining H as *heavy*-weight nodes. Intuitively, we need to take the weights into account. For this, we boost the sampling probability of a node v by an additive factor of $w(v)\log n/w(V)$, where $w(v)$ is the weight of v and $w(V)$ is the total weight of nodes in the graph. In order to show that the sampled subgraph has the desired properties, it doesn't suffice to use standard Chernoff and Union-Bound arguments. Instead, we present a more involved analysis that uses Bernstein's inequality (Fact 2). Observe that the nodes don't know the value $w(V)$. Therefore, we define a notion of *weighted degree* of a node, which is the sum of weights of its neighbors. We show that it suffices for a node v to use the maximum weighted degree in its neighborhood, instead of $w(V)$. The full argument is provided in Section 4.2.

2.2 Boosting the Approximation Factor using Local-Ratio

A useful technique for approximation algorithms is the local-ratio technique [3]. In recent years, the local-ratio technique has been found to be very useful for the distributed setting [4,5], and the Δ-approximation algorithm of [4] also uses this technique. In this work we use local-ratio to boost the approximation guarantee for MaxIS. We start with stating the local-ratio theorem for maximization problems. Here, we state it specifically for MaxIS. Given a weighted graph $G_w = (V, E, w)$, where w is a node-weight function $w : V \to \mathbb{R}$, we say that an independent set $I \subseteq V$ is r-approximate with respect to w if it is r-approximate for the optimal solution in G_w.

▶ **Theorem 6** (Theorem 9 in [3]). *Let $G_w = (V, E, w)$ be a weighted graph. Let w_1 and w_2 be two node-weight functions such that $w = w_1 + w_2$. If an independent set I is r-approximate with respect to w_1 and with respect to w_2 then it is r-approximate with respect to w as well.*

Theorem 6 already gives a simple linear-time sequential algorithm for Δ-approximation for MaxIS, as follows. Pick an arbitrary node v of positive weight, push it onto a stack, and reduce the weight of any node in the inclusive neighborhood of v (v and its neighbors) by $w(v)$. Continue recursively on the obtained graph, until there are no nodes of positive weight. When there are no remaining nodes of positive weight, pop out the stack, and construct an independent set I greedily, as follows. For each node v that is popped out from the stack, add v to I, unless it already contains a neighbor of v.

The reason that this simple algorithm gives a Δ-approximation is as follows. Consider the first iteration, when the algorithm picks an arbitrary node v, pushes it onto a stack, and reduces the weight of any node in the inclusive neighborhood of v by $w(v)$. This first iteration implicitly defines two weight functions: the *reduced* weight function w_1, and the *residual* weight function w_2, where $w = w_1 + w_2$. That is, the reduced weigh of a node u in the first step is $w_1(u) = w(v)$ if it belongs to the inclusive neighborhood of v, and $w_1(u) = 0$ otherwise. The residual weight of a node u is the remaining weight $w_2(v) = w(v) - w_1(v)$. To prove that the algorithm returns a Δ-approximation, we can assume by reverse induction that I is a Δ-approximation with respect to the residual weight function w_2. Furthermore, the independent set is constructed in a way such that it must contain at least one node in the inclusive neighborhood of v, where the weight of this node with respect to w_1 is $w(v)$. Since the degree of v is at most Δ, and the value of the optimal solution with respect to w_1 is at most $\Delta w(v)$, it follows that I is also Δ-approximation with respect to the reduced weight function w_1. Hence, by the local-ratio theorem, the independent set is also a Δ-approximation with respect to $w = w_1 + w_2$.

One can extend this idea, and rather than picking a single node in each step, the algorithm can pick an arbitrary independent set I', push all the nodes in I' onto a stack, and perform local weight reductions in the inclusive neighborhood of any node in I'. The algorithm continues recursively on the obtained graph after the weight reductions, until there are no remaining nodes of positive weight. Then, the algorithm constructs an independent set I by popping out the stack and adding nodes in the stack to I greedily. Using a similar local-ratio argument, one can show that this algorithm also returns a Δ-approximation for MaxIS. The idea of picking an independent set rather than a single node in each step was used by [4] to show a Δ-approximation algorithm in $O(\text{MIS}(n, \Delta) \log W)$ rounds.

In this work, we prove a simple yet powerful property about the local-ratio technique. Specifically, we show that the total weight of the independent set I that is constructed in the pop-out stage (with respect to the original input weight function w), is at least the total weight of the nodes in the stack (with respect to the residual weight function at the time they were pushed onto the stack). That is, let S be set of nodes that are pushed onto the stack. For $v \in S$, let w_{i_v} be the residual weight of v at the time it was pushed onto the stack. We prove that $w(I) \geq \sum_{v \in S} w_{i_v}$. We refer to this property as the *stack property.*

The stack property allows us to show a general boosting theorem, as follows. We use the local-ratio algorithm described above, where in each step we pick an independent set I' that is $(c\Delta)$-approximation for MaxIS, for some constant $c > 1$. Hence, intuitively, after $\approx c/\epsilon$ steps, the total weight in the stack should be at least $\frac{OPT(G_w)}{(1+\epsilon)\Delta}$, where $OPT(G_w)$ is the value of an optimal solution in the input graph G_w.

Low-arboricity graphs. Moreover, the stack property allows us to show an improved approximation algorithm for low-arboricity graphs, as follows. In each step, we run a $(1+\epsilon)\Delta$-approximation algorithm on the subgraph induced by the nodes of degree at most 4α, where α is the arboricity of the graph. We push the nodes in the resulting independent set I' onto the stack, and perform local weight reduction in the neighborhoods of the nodes in I'. Then, we delete all the nodes of degree at most 4α, and continue recursively on the resulting graph. Finally, we construct an independent I by popping out the stack greedily. By a standard Markov argument, after $\log n$ push steps, the graph becomes empty. Furthermore, since in each step the algorithm finds a $(1+\epsilon)4\alpha$ approximation in the subgraph induced by the nodes of degree at most 4α, and this independent set is pushed onto the stack, we are able to use the stack property to show that the constructed independent set I is roughly of the same approximation for G_w.

2.3 Analysis of the Ranking Algorithm using Martingales

In this section we provide an overview of our result for unweighted graphs of maximum degree $\Delta \le n/\log n$ (Theorem 5). First, we find an $O(\Delta)$-approximation, and then we use the boosting theorem to get a $(1+\epsilon)\Delta$-approximation. To find an $O(\Delta)$-approximation, we use the classical ranking algorithm. Recall that in the ranking algorithm, each node v picks a number r_v uniformly at random in $[0,1]$. If $r_v > r_u$ for any neighbor u of v, then v joins the independent set. Let I be the independent set that is returned by the ranking algorithm. The crux of the analysis is in using concentration inequalities to get a high-probability lower bound on the number of nodes in I. However, it is unclear how to make this approach work, as the random variables $X_v = \mathbf{1}_{v\in I}$ are not independent. While these random variables are not independent, one can obtain a weaker result in this direction. Specifically, for graphs of maximum degree at most $n^{1/3}/\text{poly}(\log n)$, one can get a useful bound on the maximum *dependency* among these variables. In particular, one can show that each X_v is dependent on at most $(n^{1/3}/\text{poly}(\log n))^2 = n^{2/3}/\text{poly}(\log n)$ other X_us, which makes it possible to show concentration using the bounded dependence Chernoff bound given in [22]. However, it is unclear how to use this approach for higher degree graphs.

The main idea of our approach is to view the ranking algorithm from a sequential perspective. Instead of picking ranks for the nodes and including a node in I if its rank is higher than that of its neighbors, we draw nodes v from V uniformly at random one at a time and add v to I if it is not adjacent to any previously drawn node. We show that the resulting independent set is identical in distribution to the independent set produced by the ranking algorithm. Note that this is not the same as a sequential greedy algorithm for maximal independent set, which would add v to I if it is not adjacent to any node in I (a weaker condition). The sequential perspective of the ranking algorithm allows us to think about the size of I incrementally. One could directly show concentration if the family of random variables $\{I_t\}_t$ was a martingale. However, this is not the case, as $|I_{t+1}| \ge |I_t|$ so it is not possible for expected increments to be 0. Instead, we create a martingale by shifting the increments so that they have mean 0. More formally, let I_t be the independent set I after the first t nodes have been drawn. Let v_t be the tth node drawn. The random variable

$$Y_t = |I_t| - |I_{t-1}| - \Pr[v_t \in I | I_{t-1}]$$

has mean 0 conditioned on I_{t-1}. Therefore, the Y_ts are increments for the martingale $X_t = \sum_{i=1}^t Y_t$. Using Azuma's Inequality, one can show that X_t concentrates around its mean, which is 0. To lower bound the size of the obtained independent set I, one therefore just needs to get a lower bound on the sum of the increment probabilities $\Pr[v_t \in I | I_{t-1}]$.

This can be lower bounded by $1/2$ when $t = o(n/\Delta)$ because when a node is drawn, it eliminates at most Δ other nodes from inclusion into I. But when $t = \Theta(n/\Delta)$, the sum of these probabilities is already $1/2(\Theta(n/\Delta)) = \Theta(n/\Delta)$, so the independent set is already large enough, as desired. The reason that this technique works for $\Delta \leq n/\log n$ is that the success probability is roughly exponential in n/Δ. Hence, by having $\Delta \leq n/\log n$, we get a high probability success, as desired.

2.4 An Overview of the Lower Bound

In oder to prove out lower bound (Theorem 4), we use a *cycle of cliques* graph. We reduce the problem of finding an MIS in a cycle to the problem of finding an independent set of size $\Omega(n/\Delta)$ in a cycle of cliques. We use Naor's lower bound [20] for finding an MIS in a cycle, which holds even against randomized algorithms. We start by stating Naor's lower bound.

▶ **Theorem 7** (Lower bound for the cycle [20]). *Any randomized algorithm in the LOCAL model for finding a maximal independent set that takes fewer than $\frac{1}{2}(\log^* n) - 4$ rounds, succeeds with probability at most $1/2$, even for a cycle of length n.*

A good way of understanding our reduction from Naor's lower bound is to first consider the following failed attempt for deterministic algorithms.

Failed Attempt 1. Let $\mathcal{A}$ be a deterministic algorithm for approximate MaxIS. Suppose that it takes $T(n)$ rounds in graphs of n nodes. We can use $\mathcal{A}$ to find a maximal independent set in a cycle C of n nodes, as follows. We start by running $\mathcal{A}$ on C to produce an independent set I. Since C is a cycle, there is a natural clockwise ordering for the nodes of I. Between any two consecutive nodes of I, there may be nodes along the cycle that are not adjacent to a node in I. We informally call these nodes the "gaps" between consecutive nodes in I. We can obtain a maximal independent set in C by "filling in" the gap between every two consecutive nodes in I with a maximal independent set (sequentially). To bound the runtime of this algorithm, we need to bound the maximum length of a gap.

One can attempt to bound the length of these gaps using what is called an *indistinguishability argument*. From a local perspective, the nodes cannot distinguish between C and a path of length $\omega(T(n))$. Hence, one can show that if there is a gap of length $\omega(T(n))$, then $\mathcal{A}$ doesn't return the required approximation on a path of length $\omega(T(n))$. As a result, filling in the gaps between nodes in I takes $O(T(n))$ rounds. Therefore, by running $\mathcal{A}$ on C and then filling in the gaps sequentially, we get an MIS in $O(T(n))$ rounds. By Linial's lower bound [19], we have that $T(n) = \Omega(\log^* n)$.

However, this argument fails for two reasons. First, this indistinguishability argument does not work. In the LOCAL model, we assume that the ID of each vertex is a number from 1 to $\text{poly}(n)$. However, this is not the case for subpaths of C with length $O(T(n))$, since $\text{poly}(n) >> \text{poly}(T(n))$. Therefore, the approximation guarantee of $\mathcal{A}$ does not need to apply to short subpaths of C, meaning that there may be large gaps in the independent set output by applying $\mathcal{A}$ on C.

Second, our goal is to show a lower bound for randomized algorithms. The main issue when running a randomized algorithm on a cycle is that the maximum length of a gap between two consecutive nodes in the independent set can be larger than $O(T(n))$. This is because randomized algorithms that succeed with high probability can fail with probability $1/\text{poly}(n)$, where n is the number of nodes in the graph. Hence, $\mathcal{A}$ can fail on a path of length $O(T(n))$ with probability $1/\text{poly}(T(n))$ which is non-negligible when $T(n) \ll n$. In particular, since there are $\Omega(|C|/T(n)) = \Omega(n/T(n))$ subpaths of length $O(T(n))$ in C, it

is likely that $\mathcal{A}$ fails on at least one of these subpaths. If on the other hand the number of nodes in the $O(T(n))$-radius neighborhood of a node was larger, then one could hope to get around this issue, as it would amplify the "local" success probability in the neighborhood of a node.

Successful Attempt 2. Instead of running $\mathcal{A}$ on C, we run it on a cycle of cliques C_1, which is obtained from C as follows. Each node $v \in C$ is replaced with a clique of size $\approx 2^{|C|}$, denoted by $D(v)$, where every two adjacent cliques are connected by a bi-clique. By running $\mathcal{A}$ on C_1 instead of C, it boosts the success probability of $\mathcal{A}$ in a small-radius neighborhood of any given node. As a result, a small-radius neighborhood of any node in C_1 must contain a node in the independent set. Using the independent set I_1 that was found in C_1, we can map it to an independent set I in C, as follows. Every $v \in C$ joins I if and only if I_1 contains a node in $D(v)$. Due to the approximation guarantee of $\mathcal{A}$ in C_1, we can prove that the maximum distance between two consecutive nodes in I_1 is small and therefore, the maximum length of a gap in I is small. Finally, we can run a greedy sequential MIS algorithm to fill the gap between every two consecutive nodes in I and find an MIS in C. Hence, if we can find an approximate-MaxIS in C_1 in $o(\log^* |C_1|)$ rounds, then we can find an MIS in C in $o(\log^*(2^{|C|})) = o(\log^* |C|)$ rounds, contradicting Naor's lower bound (Theorem 7). An illustration of the reduction with all the steps is provided.

This approach deals with the two issues found in our first reduction attempt, because the size of C_1 is bounded by a polynomial of the size of each clique (the first issue) and the large size of each clique ensures that the algorithm succeeds with high probability (the second issue).

3 Preliminaries

Some of our proofs use the following standard probabilistic tools. An excellent source for the following concentration bounds is the book by Alon and Spencer [1]. These bounds can also be found in many lecture notes about basic tail and concentration bounds.

▶ **Fact 1** (Multiplicative Chernoff Bound). *Let $X_1, ..., X_n$ be independent random variables taking values in $\{0, 1\}$. Let X denote their sum and let $\mu = E[X]$ denote the sum's expected value. Then for any $0 \le \epsilon \le 1$, it holds that:*

$$Pr[|X - \mu| \ge \epsilon\mu] \le 2\exp\left(-\frac{\epsilon^2}{2+\epsilon}\mu\right)$$

▶ **Fact 2** (Bernstein's Inequality). *Let $X_1, ..., X_n$ be independent random variables such that $\forall i, X_i \le M$. Let X denote their sum and let $\mu = \mathrm{E}[X]$ denote the sum's expected value. Then for any positive t, it holds that:*

$$Pr[|X - \mu| \ge t] \le 2\exp\left(-\frac{t^2/2}{Mt/3 + \sum_{i=1}^{n} \mathrm{Var}(X_i)}\right)$$

▶ **Fact 3** (One-sided Azuma's Inequality). *Suppose $\{X_i : i = 0, 1, 2, \ldots\}$ is a martingale and that $|X_i - X_{i-1}| \le c_i$ almost surely. Then, for all positive integers N and all positive reals t,*

$$\Pr[X_N - X_0 \le -t] \le \exp\left(-\frac{t^2}{2\sum_{i=1}^{N} c_i^2}\right)$$

Assumptions. In all of our upper and lower bounds, we don't assume that the nodes have any global information. In particular, they don't know n or Δ. The only information that each node has before the algorithm starts is its own identifier, and some polynomial upper bound on n (Since the nodes can send $c \log n$ bits in each round to each of their neighbors, naturally, they know some polynomial upper bound on n).

Some notations. The input graph is denoted by $G_w = (V, E, w)$, where V is the set of nodes, E is the set of edges, and w is the weight function. The reason that we choose to add the weight function in a subscript is that some parts of the analysis deal with graphs that have the same sets of nodes and edges as the input graph, but a different weight function. Hence, such a graph will be denoted by $G_{w'} = (V, E, w')$, to indicate that it is the same as the input graph, but with weight function w' rather than w.

We denote by $N^+(v)$ the inclusive neighborhood of v, which consists of $N(v) \cup \{v\}$, where $N(v)$ is the set of neighbors of v. Furthermore, we denote by $deg(v) = |N(v)|$ the number of neighbors of a node v. Finally, we denote by $w(V')$ the total weight of nodes in $V' \subseteq V$. That is, $w(V') = \sum_{v \in V'} w(v)$.

4 A $(1+\epsilon)\Delta$-Approximation Algorithm

In this section we prove Theorems 1 and 2.

▶ **Theorem 1.** *There is an $O(\text{MIS}(n, \Delta)/\epsilon)$-round algorithm in the CONGEST model that finds a $(1+\epsilon)\Delta$-approximation for maximum-weight independent set. Whether the algorithm is deterministic or randomized, depends on the* MIS *algorithm that is run as a black-box.*

▶ **Theorem 2.** *There is a randomized $(\text{poly}(\log \log n)/\epsilon)$-round algorithm in the CONGEST model that finds, with high probability, a $(1+\epsilon)\Delta$-approximation for maximum-weight independent set.*

Theorems 1 and 2 share a similar proof structure. First, we present algorithms for $O(\Delta)$-approximation in Sections 4.1 and 4.2. Then, by using a general boosting theorem, we get $(1+\epsilon)\Delta$-approximation algorithms.

4.1 An $O(\text{MIS}(n, \Delta))$-Round Algorithm for $O(\Delta)$-Approximation

In this section we show a very simple $O(\text{MIS}(n, \Delta))$-round algorithm that finds an $O(\Delta)$-approximation for MaxIS.

▶ **Theorem 8.** *Given a weighted graph $G_w = (V, E, w)$, there is an $O(\text{MIS}(n, \Delta))$-round algorithm that finds an independent set of weight at least $\frac{w(V)}{4(\Delta+1)}$, in the CONGEST model. Whether the algorithm is deterministic or randomized depends on the* MIS *algorithm that is used as a black-box.*

Algorithm For every $v \in V$, let $\delta(v)$ be the maximum degree of a node in the inclusive neighborhood of v. That is, $\delta(v) = \max\{deg(u) \mid u \in N^+(v)\}$. A node v is called *good* if $w(v) \geq \frac{1}{2(\delta(v)+1)} \sum_{u \in N^+(v)} w(u)$. The algorithm finds a maximal independent set I in the subgraph induced by the set of good nodes. We prove the following lemma.

▶ **Lemma 9.** $w(I) \geq w(V)/4(\Delta + 1)$

Proof. Let V_{good} be the set of good nodes, and let $\overline{V} = V \setminus V_{good}$. Observe that,

$$\sum_{v\in\overline{V}} w(v) \leq \sum_{v\in\overline{V}} \frac{1}{2(\delta(v)+1)} \sum_{u\in N^+(v)} w(u) \leq \sum_{v\in V} \frac{deg(v)+1}{2(deg(v)+1)} w(v) = w(V)/2$$

$$\Rightarrow \sum_{v\in I} w(v) \geq \sum_{v\in I} \frac{1}{2(\delta(v)+1)} \sum_{u\in N^+(v)} w(u) \geq \sum_{v\in I} \frac{1}{2(\Delta+1)} \sum_{u\in N^+(v)\cap V_{good}} w(u)$$

$$\geq \frac{1}{2(\Delta+1)} \sum_{v\in V_{good}} w(v) \geq w(V)/4(\Delta+1)$$

as desired. Since the value of an optimal solution in G_w is at most $w(V)$, the algorithm returns an $O(\Delta)$-approximation for MaxIS. ◀

Success with high probability. Given a graph of n nodes, an algorithm that finds a maximal independent set in the graph with high probability is an algorithm that succeeds with probability at least $1 - 1/n^c$ for some constant $c > 1$. In the algorithm above, the black box can be a randomized algorithm that is run on a subgraph $H = (V_H, E_H)$ of G_w. Since $n_H = |V_H|$ is potentially much smaller than n, one may wonder whether the algorithm above actually succeeds with high probability with respect to n. The main idea is to use an algorithm that is *intended* to work for graphs with n nodes, rather than n_H nodes. We prove the following lemma, whose proof is by a simple padding argument that is deferred to the full version [17].

▶ **Lemma 10.** *Let $\mathcal{A}$ be an $\mathrm{MIS}(n,\Delta)$-round algorithm that finds a maximal independent set with success probability p in a graph of n nodes and maximum degree Δ. Let $H = (V_H, E_H)$ be a graph of $n_H \leq n$ nodes with $(c \log n)$-bit identifiers, for some constant c, and let Δ_H be the maximum degree in H. There is an $O(\mathrm{MIS}(n, \Delta_H))$-round algorithm $\mathcal{A}'$ that finds a maximal independent set in H with success probability p.*

4.2 A $\mathrm{poly}(\log\log n)$-Round Algorithm for $O(\Delta)$-Approximation

In this section we show a $\mathrm{poly}(\log\log n)$-round algorithm that finds an $O(\Delta)$-approximation.

▶ **Theorem 11.** *Given a weighted graph $G_w = (V, E, w)$, there is a constant $c > 1$ and a $\mathrm{poly}(\log\log n)$-round algorithm in the CONGEST model that finds, with high probability, an independent set of weight at least $\frac{w(V)}{c\Delta}$.*

Our algorithm has the following two-step structure.

1. First, we sample a sparse subgraph $H_w = (V_H, E_H, w)$ of G_w with the following two properties:
   a. The maximum degree Δ_H of H_w is at most $O(\log n)$.
   b. $w(V_H)/\Delta_H = \Omega(w(V)/\Delta)$. That is, the ratio between the total weight and maximum degree in H_w is at least, up to a constant factor, as in G_w.
2. Then, we use Theorem 8 to find an independent set in H_w of size at least $\frac{w(V_H)}{4(\Delta_H+1)} = \frac{w(V)}{c\Delta}$, for some constant $c > 1$, in $O(\mathrm{MIS}(n, \Delta_H)) = O(\mathrm{MIS}(n, \log n)) = \mathrm{poly}(\log\log n)$ rounds.

The first step: sampling a subgraph with the desired properties. Recall that $w(N(v))$ is the sum of weights of the neighbors of v, which we call the *weighted degree* of v. For each node $v \in V$, let $w_{max}(v) = \max\{w(N(u)) \mid u \in N^+(v)\}$. It is useful to think about $w_{max}(v)$ as the *maximum weighted degree* of a node in the inclusive neighborhood of v. We sample a

subgraph $H_w = (V_H, E_H, w)$, as follows. Let $\lambda \geq 1$ be a constant to be chosen later. Recall that $\delta(v)$ is the maximum degree of a node in the inclusive neighborhood of v. Each node $v \in V$ joins V_H with probability

$$p(v) = \lambda \log n \cdot (\frac{1}{\delta(v)} + \frac{w(v)}{w_{max}(v)})$$

Where if $p(v) \geq 1$, then v joins H deterministically. In Lemma 12, we show that the maximum degree of H_w is $\Delta_H = O(\log n)$. In Lemma 16, we show that $w(V_H) = \Omega(\min\{w(V), w(V)\log n/\Delta\})$.

▶ **Lemma 12.** *The maximum degree Δ_H in H_w is $O(\log n)$, with high probability.*

Proof. Let $V^+ = \{v \in V \mid p(v) \geq 1\}$. We show that each node u has at most $O(\log n)$ neighbors in $V^+ \cap V_H$, and at most $O(\log n)$ neighbors in $(V \setminus V^+) \cap V_H$. Let $N_H(v)$ be the set of neighbors of v in H.

1. For every $v \in V$, $|N_H(v) \cap V^+| \leq 2\lambda \log n$: Assume towards a contradiction that there are more than $2\lambda \log n$ nodes in $N_H(v) \cap V^+$. Since each node $v \in V^+$ has $p(v) \geq 1$, it holds that

$$\sum_{u \in N(v) \cap V^+} p(u) \geq \sum_{u \in N_H(v) \cap V^+} p(u) > 2\lambda \log n$$

On the other hand,

$$\sum_{u \in N(v) \cap V^+} p(u) \leq \sum_{u \in N(v)} p(u) = \sum_{u \in N(v)} \lambda \log n \cdot (\frac{1}{\delta(v)} + \frac{w(v)}{w_{max}(v)})$$

Since $deg(v) = |N(v)|$ and $w(N(v)) = \sum_{u \in N(v)} w(u)$ are lower bounds on $\delta(v)$ and $w_{max}(v)$, respectively, we have that

$$\sum_{u \in N(v)} \lambda \log n \cdot (\frac{1}{\delta(u)} + \frac{w(u)}{w_{max}(u)}) \leq \sum_{u \in N(v)} \lambda \log n \cdot (\frac{1}{deg(v)} + \frac{w(u)}{w(N(v))}) = 2\lambda \log n$$

which is a contradiction.

2. $|N_H(v) \cap (V \setminus V^+)| \leq 2\lambda \log n$: Observe that the expected number of neighbors of v in $N_H(v) \cap (V \setminus V^+)$ is

$$\sum_{u \in N(v)} p(u) \leq 2\lambda \log n$$

Since $|N_H(v) \cap (V \setminus V^+)|$ is a sum of independent random variables, one can apply Chernoff's bound (Fact 1) to achieve that this number concentrates around its expectation with high probability.

By applying a standard Union-Bound argument over all the nodes, we conclude that the maximum degree in H_w is $\Delta_H = O(\log n)$ with high probability. ◀

The rest of this section is devoted to the task of proving that $w(V_H) = \Omega(\min\{w(V), w(V)\log n/\Delta\})$. This is proved in Lemma 16. First, we start by proving a slightly weaker lemma, that assumes that for all $v \in V$, $p(v) \leq 1$.

▶ **Lemma 13.** *Assume $p(v) \leq 1$, for all $v \in V$. It holds that $w(V_H) = \Omega(w(V)\log n/\Delta)$, with high probability.*

Main idea of the proof of Lemma 13. Let $w_1 \geq w_2 \geq ... \geq w_n$ be a sorting of the weights of nodes in V in a decreasing order (where ties are broken arbitrarily). Let $V_{high} = \{u \in V \mid w(u) \in \{w_1, ..., w_\Delta\}\}$, and let $V_{low} = V \setminus V_{high} = \{u \in V \mid w(u) \in \{w_{\Delta+1}, ..., w_n\}\}$. That is, V_{high} contains the Δ heaviest nodes, and V_{low} contains all the other nodes. The proof is split into the following two cases that are proven separately in Claims 14 and 15.

1. $w(V_{high}) \geq w(V)/2$: In this case, at least half of the total weight is distributed among high-weight nodes. Intuitively, we need to make sure that we get many of these high-weight nodes. Since the number of high-weight nodes that are sampled is a sum of independent random variables, we are able to use Chernoff's bound to prove that many of them are sampled, with high probability. The full proof for this case is presented in Claim 14.
2. $w(V_{low}) \geq w(V)/2$: In this case, at least half of the total weight is distributed among low-weight nodes. Therefore, it is sufficient to show that $w(V_H) = \Omega(w(V_{low}) \log n/\Delta)$. The key property here is that we can bound the maximum weight of a node in V_{low} by $w(V)/\Delta$. We show how to use this property together with Bernstein's inequality to prove Lemma 13 for this case. The full proof for this case is presented in Claim 15.

▷ Claim 14. Assume that for all $v \in V$, $p(v) \leq 1$. Let $V_{high} = \{u \in V \mid w(u) \in \{w_1, ..., w_\Delta\}\}$. If $w(V_{high}) \geq w(V)/2$, then $w(V_H) = \Omega(w(V) \log n/\Delta)$, with high probability.

Proof. Let $S = \{v \in V_{high} \mid w(v) \geq w(V)/4\Delta\}$. We start by showing that at least a constant fraction of the total weight in G_w is distributed among nodes in S. Let $\overline{S} = V_{high} \setminus S$, we start by showing that $w(\overline{S}) \leq w(V)/4$:

$$w(\overline{S}) \leq \sum_{v \in \overline{S}} w(v) \leq \sum_{v \in \overline{S}} \frac{w(V)}{4\Delta} \leq \frac{w(V)}{4}$$

where the last inequality holds because $|\overline{S}| \leq |V_{high}| = \Delta$. Therefore, $w(S) = w(V_{high} \setminus \overline{S}) = w(V_{high}) - w(V(\overline{S})) \geq w(V)/4$. Next, we show that $|S \cap V_H| = \Omega(\log n)$, by using Chernoff's bound. Let x_v be a $\{0, 1\}$ random variable indicating whether $v \in V_H$, and let $X = \sum_{v \in S} x_v$. We show that the expectation of X is at least $c \log n/4$.

$$\begin{aligned}\mathbb{E}[X] &= \sum_{v \in S} \mathbb{E}[x_v] = \sum_{v \in S} p(v) = \sum_{v \in S} \lambda \log n \cdot (\frac{1}{\delta(v)} + \frac{w(v)}{w_{max}(v)}) \\ &\geq \sum_{v \in S} \frac{w(v)\lambda \log n}{w(V)} \geq \frac{\lambda \log n}{w(V)} \cdot \sum_{v \in S} w(v) = \frac{w(S)\lambda \log n}{w(V)} \geq \frac{\lambda \log n}{4}\end{aligned}$$

Furthermore, sine X is a sum of independent $\{0, 1\}$ random variables with expectation $\Omega(\log n)$, by applying Chernoff's bound (Fact 1), we conclude that there are at least $\Omega(\log n)$ nodes in $S \cap V_H$, with high probability. Since each node in S has weight at least $w(V)/4\Delta$, this implies that the total weight in V_H is $w(V_H) \geq w(S \cap V_H) = \Omega(w(V) \log n/\Delta)$, with high probability, as desired. ◁

▷ Claim 15. Assume that for all $v \in V$, $p(v) \leq 1$. Let $V_{low} = \{v \in V \mid w(v) \in \{w_{\Delta+1}, ..., w_n\}\}$. If $w(V_{low}) \geq w(V)/2$, then $w(V_H \cap V_{low}) = \Omega(w(V) \log n/\Delta)$, with high probability.

Proof. Let x_v be a $\{0, 1\}$ random variable indicating whether $v \in V_H$, let $y_v = x_v \cdot w(v)$, and let $Y = \sum_{v \in V_{low}} y_v$. We prove the following 3 properties:

1. $\mathbb{E}(Y) \geq \frac{w(V)\lambda \log n}{2\Delta}$: this is because

$$\mathbb{E}[Y] = \sum_{v\in V_{low}} p(v)\cdot w(v) = \sum_{v\in V_{low}} \lambda \log n \cdot (\frac{1}{\delta(v)} + \frac{w(v)}{w_{max}(v)})\cdot w(v)$$
$$\geq \sum_{v\in V_{low}} \frac{w(v)\lambda\log n}{\Delta} = \frac{w(V_{low})\lambda \log n}{\Delta} \geq \frac{w(V)\lambda\log n}{2\Delta}$$

where the last equality holds since $w(V_{low}) \geq w(V)/2$.

2. For any $v \in V_{low}$, it holds that $w(v) \leq w(V)/\Delta$: Recall that $\{w_1, \cdots, w_n\}$ is an ordering of the weight of nodes by a decreasing order. Hence, for any j, it holds that

$$w_j \cdot j \leq \sum_{i=1}^{j} w_j \leq w(V)$$

where the first inequality holds because w_j is the minimum among $\{w_1, ..., w_j\}$. Hence, since each node $v \in V_{low}$ has weigh w_j where $j > \Delta$, we have that $w(v) \leq w(V)/\Delta$ for any $v \in V_{low}$.

3. It holds that $\sum_{v\in V_{low}} \mathbb{E}[y_v^2] \leq w(V)\cdot \mathbb{E}[Y]/\Delta$: First, observe that

$$\sum_{v\in V_{low}} \mathbb{E}[y_v^2] \leq \max\{w(v) \mid v \in V_{low}\} \cdot \sum_{v\in V_{low}} \mathbb{E}[y_v] = \max\{w(v) \mid v\in V_{low}\}\cdot \mathbb{E}[Y]$$
$$\leq \frac{w(V)\cdot \mathbb{E}[Y]}{\Delta}$$

where the last inequality holds by the second property.

By proving these three properties, we have satisfied all the prerequisites of Bernstein's inequality. A direct application of the inequality yields:

$$Pr\big[|Y - \mathbb{E}[Y]| \geq \mathbb{E}[Y]/2\big] \leq 2\exp\left(-\frac{\mathbb{E}[Y]^2/8}{M\cdot \mathbb{E}[Y]/6 + \sum_{v\in V_{low}} \mathrm{Var}(y_v)}\right)$$

By the second and third properties, we have that

$$\sum_{v\in V_{low}} \mathrm{Var}(y_v) = \sum_{v\in V_{low}} \mathbb{E}(y_v^2) - \mathbb{E}[y_v]^2 \leq \sum_{v\in V_{low}} \mathbb{E}(y_v^2) \leq \frac{w(V)\cdot \mathrm{E}[Y]}{\Delta}$$
$$\Rightarrow Pr\big[|Y-\mathbb{E}[Y]| \geq \mathbb{E}[Y]/2\big] \leq 2\exp\left(-\frac{\mathbb{E}[Y]^2/8}{\frac{w(V)\cdot\mathbb{E}[Y]}{6\Delta} + \frac{w(V)\cdot \mathbb{E}[Y]}{\Delta}}\right) \leq 2\exp\left(-\frac{6\Delta\cdot\mathbb{E}[Y]/8}{7w(V)}\right)$$

Furthermore, by the first property, we have that

$$\mathbb{E}[Y] \geq w(V)\lambda\log n/2\Delta$$
$$\Rightarrow 2\exp\left(-\frac{6\Delta\cdot \mathbb{E}[Y]/8}{7w(V)}\right) \leq 2\exp\left(-\frac{6w(V)\lambda\log n}{56w(V)}\right) = 2\exp\left(-\frac{6\lambda\log n}{56})\right)$$

Finally, choosing $\lambda = 112/6$ implies that:

$$Pr\big[|Y - \mathbb{E}[Y]| \geq \mathbb{E}[Y]/2\big] \leq \frac{1}{n^{2\log e}} < \frac{1}{n^2}$$

as desired. Furthermore, we can boost the success probability to $1 - 1/n^k$ for any constant $k > 1$, by setting $\lambda = \frac{112k}{3}$.

◁

Having proved claims 14 and 15, this finishes the proof of Lemma 13. Lemma 13 makes the assumption that $p(v) \leq 1$ for all $v \in V$. We remove this assumption in the proof of the following lemma.

▶ **Lemma 16.** *It holds that $w(V_H) = \Omega(\min\{w(V), w(V) \log n/\Delta\})$, with high probability.*

Proof. Let $V^+ = \{u \in V \mid p(w) \geq 1\}$. The proof is split into two cases:

1. $w(V^+) \geq w(V)/2$: Since all the nodes in V^+ join V_H deterministically, this implies that $w(V_H) \geq w(V^+) \geq w(V)/2$.
2. $w(V^+) < w(V)/2$: This implies that $w(V \setminus V^+) \geq w(V)/2$. Since each node $w \in V \setminus V^+$ has $p(w) < 1$, we can apply Lemma 13 directly on the nodes in $V \setminus V^+$ to conclude that $w(V_H) = \Omega(w(V \setminus V^+) \log n/\Delta) = \Omega(w(V) \log n/\Delta)$, with high probability, as desired. ◀

Now we are ready to finish the proof of Theorem 11.

Proof of Theorem 11. Since both Lemma 12 and 16 above hold with high probability, we can apply another standard Union-Bound argument to conclude that both of them hold with high probability (simultaneously). Hence, by running the algorithm from Section 4.1 on H_w, we get an independent set of weight $\Omega(w(V_H)/\Delta_H) = \Omega(\min\{w(V), w(V) \log n/\Delta\}/\Delta_H) = \Omega(w(V)/\Delta)$, in $\mathrm{MIS}(n, \Delta_H) = \mathrm{MIS}(n, \log n) = \mathrm{poly}(\log \log n)$ rounds, with high probability, as desired. ◀

References

1 Noga Alon and Joel Spencer. *The Probabilistic Method*. John Wiley, 1992.

2 Nir Bachrach, Keren Censor-Hillel, Michal Dory, Yuval Efron, Dean Leitersdorf, and Ami Paz. Hardness of distributed optimization. In Peter Robinson and Faith Ellen, editors, *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*, pages 238–247. ACM, 2019. doi:10.1145/3293611.3331597.

3 Amotz Bar-Noy, Reuven Bar-Yehuda, Ari Freund, Joseph Naor, and Baruch Schieber. A unified approach to approximating resource allocation and scheduling. *Journal of the ACM (JACM)*, 48(5):1069–1090, 2001.

4 Reuven Bar-Yehuda, Keren Censor-Hillel, Mohsen Ghaffari, and Gregory Schwartzman. Distributed approximation of maximum independent set and maximum matching. In *PODC*, pages 165–174. ACM, 2017.

5 Reuven Bar-Yehuda, Keren Censor-Hillel, and Gregory Schwartzman. A distributed $(2 + \epsilon)$-approximation for vertex cover in o(log Δ / ϵ log log Δ) rounds. *J. ACM*, 64(3):23:1–23:11, 2017.

6 Leonid Barenboim, Michael Elkin, Seth Pettie, and Johannes Schneider. The locality of distributed symmetry breaking. *J. ACM*, 63(3):20:1–20:45, 2016. doi:10.1145/2903137.

7 Marijke HL Bodlaender, Magnús M Halldórsson, Christian Konrad, and Fabian Kuhn. Brief announcement: Local independent set approximation. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, July 25-28, 2016*, To Appear 2016.

8 Ravi B. Boppana, Magnús M. Halldórsson, and Dror Rawitz. Simple and local independent set approximation. In *SIROCCO*, volume 11085 of *Lecture Notes in Computer Science*, pages 88–101. Springer, 2018.

9 Keren Censor-Hillel, Seri Khoury, and Ami Paz. Quadratic and near-quadratic lower bounds for the CONGEST model. In *31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria*, pages 10:1–10:16, 2017. doi:10.4230/LIPIcs.DISC.2017.10.

10 Keren Censor-Hillel, Merav Parter, and Gregory Schwartzman. Derandomizing local distributed algorithms under bandwidth restrictions. In *31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria*, volume 91 of *LIPIcs*, pages 11:1–11:16. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017. doi:10.4230/LIPIcs.DISC.2017.11.

11 Andrzej Czygrinow, Michal Hańćkowiak, and Wojciech Wawrzyniak. Fast distributed approximations in planar graphs. In *Distributed Computing*, pages 78–92. Springer, 2008.

12 Mohsen Ghaffari. An improved distributed algorithm for maximal independent set. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '16, pages 270–277. SIAM, 2016. URL: http://dl.acm.org/citation.cfm?id=2884435.2884455.

13 Mohsen Ghaffari. Distributed maximal independent set using small messages. In *SODA*, pages 805–820. SIAM, 2019.

14 Mohsen Ghaffari. Personal communication, Feb 2020.

15 Mohsen Ghaffari, Fabian Kuhn, and Yannic Maus. On the complexity of local distributed graph problems. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 784–797, 2017. doi:10.1145/3055399.3055471.

16 Magnús M. Halldórsson and Christian Konrad. Computing large independent sets in a single round. *Distributed Computing*, 31(1):69–82, 2018.

17 Ken-ichi Kawarabayashi, Seri Khoury, Aaron Schild, and Gregory Schwartzman. Improved distributed approximation to maximum independent set. *CoRR*, abs/1906.11524, 2019. arXiv:1906.11524.

18 Christoph Lenzen and Roger Wattenhofer. Leveraging linial's locality limit. In *Distributed Computing*, pages 394–407. Springer, 2008.

19 Nathan Linial. Locality in distributed graph algorithms. *SIAM Journal on Computing*, 21(1):193–201, 1992.

20 Moni Naor. A lower bound on probabilistic algorithms for distributive ring coloring. *SIAM J. Discrete Math.*, 4(3):409–412, 1991. doi:10.1137/0404036.

21 David Peleg. *Distributed Computing: A Locality-sensitive Approach.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

22 Sriram V. Pemmaraju. Equitable coloring extends chernoff-hoeffding bounds. In Michel X. Goemans, Klaus Jansen, José D. P. Rolim, and Luca Trevisan, editors, *Approximation, Randomization and Combinatorial Optimization: Algorithms and Techniques, 4th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems, APPROX 2001*, volume 2129, pages 285–296. Springer, 2001. doi:10.1007/3-540-44666-4_31.

23 Václav Rozhon and Mohsen Ghaffari. Polylogarithmic-time deterministic network decomposition and distributed derandomization. In Konstantin Makarychev, Yury Makarychev, Madhur Tulsiani, Gautam Kamath, and Julia Chuzhoy, editors, *Proccedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020, Chicago, IL, USA, June 22-26, 2020*, pages 350–363. ACM, 2020. doi:10.1145/3357713.3384298.

Models of Smoothing in Dynamic Networks

Uri Meir
Tel Aviv University, Israel

Ami Paz
Faculty of Computer Science, Universität Wien, Austria

Gregory Schwartzman
JAIST, Ishikawa, Japan

Abstract

Smoothed analysis is a framework suggested for mediating gaps between worst-case and average-case complexities. In a recent work, Dinitz et al. [Distributed Computing, 2018] suggested to use smoothed analysis in order to study dynamic networks. Their aim was to explain the gaps between real-world networks that function well despite being dynamic, and the strong theoretical lower bounds for arbitrary networks. To this end, they introduced a basic model of smoothing in dynamic networks, where an adversary picks a sequence of graphs, representing the topology of the network over time, and then each of these graphs is slightly perturbed in a random manner.

The model suggested above is based on a per-round noise, and our aim in this work is to extend it to models of noise more suited for multiple rounds. This is motivated by long-lived networks, where the amount and location of noise may vary over time. To this end, we present several different models of noise. First, we extend the previous model to cases where the amount of noise is very small. Then, we move to more refined models, where the amount of noise can change between different rounds, e.g., as a function of the number of changes the network undergoes. We also study a model where the noise is not arbitrarily spread among the network, but focuses in each round in the areas where changes have occurred. Finally, we study the power of an adaptive adversary, who can choose its actions in accordance with the changes that have occurred so far. We use the flooding problem as a running case-study, presenting very different behaviors under the different models of noise, and analyze the flooding time in different models.

2012 ACM Subject Classification Theory of computation → Distributed computing models; Theory of computation → Dynamic graph algorithms

Keywords and phrases Distributed dynamic graph algorithms, Smoothed analysis, Flooding

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.36

Related Version A full version of the paper is available at https://arxiv.org/abs/2009.13006.

Funding *Ami Paz*: Supported by the Austrian Science Fund (FWF): P 33775-N, Fast Algorithms for a Reactive Network Layer.
Gregory Schwartzman: This work was supported by JSPS Kakenhi Grant Number JP19K20216 and JP18H052.

Acknowledgements The authors are thankful to Seth Gilbert for interesting discussions.

1 Introduction

Distributed graph algorithms give a formal theoretical framework for studying networks. There is abundant literature on the topic for static networks, and recently, an increasing amount of work on dynamic networks, under the title of distributed dynamic graph algorithms. Yet, our understanding of the interconnections between the theoretical models and real world networks is unsatisfactory. That is, we have strong theoretical lower bounds for many distributed settings, both static and dynamic, yet real-world communication networks are functioning, and in a satisfactory manner.

© Uri Meir, Ami Paz, and Gregory Schwartzman;
licensed under Creative Commons License CC-BY
34th International Symposium on Distributed Computing (DISC 2020).
Editor: Hagit Attiya; Article No. 36; pp. 36:1–36:16
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

One approach to bridging the gap between the strong theoretical lower bounds and the behaviour of real-world instances, is the idea of *smoothed analysis*. Smoothed analysis was first introduced by Spielman and Teng [18, 19], in an attempt to explain the fact that some problems admit strong theoretical lower bounds, but in practice are solved on a daily basis. The explanation smoothed analysis suggests for this gap, is that lower bounds are proved using very specific, pathological instances of a problem, which are highly unlikely to happen in practice. They support this idea by showing that some lower bound instances are extremely *fragile*, i.e., a small random perturbation turns a hard instance into an easy one. Spielman and Teng applied this idea to the simplex algorithm, and showed that, while requiring an exponential time in the *worst case*, if we apply a small random noise to our instance before executing the simplex algorithm on it, the running time becomes polynomial in expectation.

Smoothed analysis was first introduced to the distributed setting in the seminal work of Dinitz et al. [8], who considered distributed dynamic networks. These are networks that changes over time, and capture many real world systems, from Blockchain, through vehicle networks, and to peer-to-peer networks. They are modeled by a sequence of communication graphs, on which the algorithm needs to solve a problem despite the changes in the communication topology. Adapting the ideas of smoothed analysis in this setting is not an easy task. First, while the classical smoothed analysis is applied to a single input, in the dynamic setting we deal with an infinite sequence of communication rounds, on ever-changing graphs. And second, defining the correct perturbation which the input undergoes is more challenging, as the input is discrete, as opposed to the continuous input of the simplex algorithm, where Gaussian noise is a very natural candidate.

To address the above challenges, Dinitz et al. fix a *noise parameter* $k > 0$, and then, after the adversary picks an infinite sequence of graphs $\{G_i\}$, they derive a new series of graphs $\{H_i\}$ by perturbing every G_i with an addition or deletion of roughly k random edges. Specifically, the perturbation (k-smoothing) is done by choosing uniformly at random a graph within Hamming distance at most k of G_i (i.e., a graph that is different from G_i by at most k edges), which also has some desired properties (say, connectivity). Note that in the typical case $\Omega(k)$ edges are perturbed, as most of the graphs with Hamming distance at most k from G_i have Hamming distance $\Omega(k)$ from it. Using this model, they analyze the problems of flooding, hitting time, and aggregation.

In this paper we present a study of *models of smoothing*, or put differently, a study of models of noise in dynamic networks. To this end, we focus on one of the three fundamental problems presented in the previous work – the flooding problem. In this problem, a source vertex has a message it wishes to disseminate to all other vertices in the network. In each round, every vertex which has the message forewords it to all of its neighbors, until all vertices are informed. First, note that the problem is trivially solvable in static networks, in diameter time. Second, in the dynamic setting, it is easy to construct a sequence of graph where flooding takes $\Omega(n)$ rounds, even if each of them has a small diameter. It is thus not surprising that adding random links between vertices may accelerate the flooding process, and indeed, it was shown in [8] that in k-smoothed dynamic networks, the complexity of flooding drops to $\tilde{\Theta}(n^{2/3}/k^{1/3})$ (where the $\tilde{}$ sign hides $\operatorname{polylog} n$ factors). Note the huge improvement, from $\Omega(n)$ to $\tilde{O}(n^{2/3})$, even for $k = 1$.

1.1 Motivation

The significant step made by Dinitz et al. in introducing smoothed analysis to the distributed dynamic environment left many fascinating questions unanswered. One natural direction of research has to do with applying smoothed analysis for a variety of problems, but here, we

take a different approach. In an attempt to understand the essential properties of smoothed analysis in our setting, we focus on questions related to the very basic concept of smoothing and study several models of noise. We outline some of the related questions below.

The curious gap between $k = 0$ and $k = 1$. Dinitz et al. show a tight bound of $\tilde{\Theta}(n^{2/3}/k^{1/3})$ for the problem of flooding in a dynamic networks with noise k. That is, as k decreases, flooding becomes harder, but only up to $\tilde{\Theta}(n^{2/3})$ for a constant k. However, when there is no noise at all, a *linear*, in n, number of rounds is required. That is, there is a curious gap between just a tiny amount of noise and no noise at all, which stands in sharp contrast with the smooth change in the flooding time as a function of k, when k ranges from 1 and to $\Omega(n)$. One may wonder if there is a natural way to extend the model presented by Dinitz et al. to bridge this gap.

An oblivious adversary. The results of Dinitz et al. assume an oblivious adversary. That is, the evolution of the network must be decided by the adversary in advance, without taking into account the noise added by the smoothing process. It is natural to ask what is the power of an *adaptive* adversary compared to an oblivious one. As smoothed analysis aims to be the middle ground between worst-case and average-case analysis, it is very natural to consider the effect of noise on the strongest possible adversary, i.e., an adaptive adversary.

Change-dependent noise. The type of noise considered by Dinitz et al. remains constant between rounds, regardless of the topology or the actions of the adversary, which corresponds to "background noise" in the network. However, it is very natural to consider cases where the amount of noise itself changes over time. More specifically, if we think of a noise as an artifact of the changes the adversary performs to the system, and thus, it is natural to expect different amounts of noise if the adversary performs a single change, or changes the whole topology of the graph. Hence, a natural model of noise in a network is one where the added noise is *proportional* to the amount of change attempted by the adversary, that is, k is proportional to the Hamming distance between the current graph and the next one. A different, natural definition of a changes-dependent noise, is where the noise occurs only in the changing edges, and not all the graph.

1.2 Our results

To address the above points of interest, we prove upper and lower bounds, as summarized in Table 1. First, we show a natural extension of the previously presented noise model to *fractional* values of k. For $k \geq 1$, our model roughly coincides with the prior model and results, while for $0 < k < 1$, in our model a single random change occurs in the graph with probability k. In our model, we show a bound of $\tilde{\Theta}(\min\{n^{2/3}/k^{1/3}, n\})$ for flooding, for values of k ranging from $k = 0$ to $k = \Theta(n)$, even fractional – see theorems 6, 7, and 8. The flooding time thus has a clean and continuous behavior, even for the range $0 < k < 1$ which was not studied beforehand, providing a very natural extension of the previous model.

Next, we focus our attention on an adaptive adversary, that can choose the next graph depending on the smoothing occurred in the present one. Here, we show that the added power indeed makes the adversary stronger, and yet, flooding can be solved in this case faster than in a setting where no smoothing is applied. Specifically, in theorems 9 and 10 we show that in this setting flooding can be done in $\tilde{O}(n/\sqrt{k})$ rounds, and takes at least $\tilde{\Omega}(n/k)$ rounds. This result presents a strict separation between the power of an adaptive and an oblivious adversaries.

Table 1 Bounds on flooding time in different models of smoothing.

Model	Upper bound	Lower bound	Reference
Integer noise Non-adaptive adv.	$O\left(n^{2/3}(1/k)^{1/3}\log n\right)$	$\Omega\left(n^{2/3}/k^{1/3}\right)$, for $k \leq \sqrt{n}$	Dinitz et al. [8]
Fractional noise Non-adaptive adv.	$O\left(\min\left\{n, n^{2/3}(\log n/k)^{1/3}\right\}\right)$	$\Omega\left(\min\left\{n, n/k, n^{2/3}/k^{1/3}\right\}\right)$	Thm. 6, 7, 8
Fractional noise Adaptive adv.	$O\left(\min\left\{n, n\sqrt{\log n/k}\right\}\right)$	$\Omega(\min\{n, n\log k/k\})$	Thm. 9, 10
Proportional noise Non-adaptive adv.	$O\left(n^{2/3}\cdot(D\log n/\epsilon)^{1/3}\right)$		Thm. 12
Proportional noise Adaptive adv.	$O(n)$	$\Omega(n)$	Thm. 13
Targeted noise	$O\left(\min\left\{n, D\log n/\epsilon^{D^2}\right\}\right)$	$\Omega(n)$, for $D \in \Theta(\log n)$	Thm 14, 15

We then turn our attention to a different type of noise, and introduce two new models of *responsive noise* – noise which depends on the changes the adversary preforms. The goal of studying responsive noise is to better understand systems where the noise is an artifact of the changes, and more changes induce more noise. We consider two, completely different cases: if only the *amount* of noise depends on the changes, then the system is less stable than in the prior models, and flooding can be delayed a lot by the adversary. On the other hand, if the same amount of noise is *targeted* at the changing links themselves, then the system remains stable, and flooding occurs much faster. In both models, our results surprisingly depend on a new attribute – the static diameter of the graph, D.

The first model of responsive noise we introduce is the *proportional noise* model, where the noise is randomly spread among the graph edges as before, and its amount is proportional to the number of proposed changes. We consider two types of adversaries in this setting – adaptive, and oblivious. Theorem 12 shows that $\tilde{O}(\min\{n^{2/3}D^{1/3}/\epsilon^{1/3}, n\})$ rounds are sufficient for flooding in this model with an oblivious adversary. Here, the static diameter D comes into play, since the adversary can now force change-free rounds: if the adversary does not make any change, no noise occurs, the graph remains intact and no random "shortcut edges" are created. Current lower bounds for flooding with oblivious adversaries seem to require many changes, but in the proportional noise model this results in the addition of many random edges, thus speeding up the flooding process. In addition, the upper bound suggests that the static diameter D should come into play in the lower bounds, which is not the case in previous constructions. While we believe our upper bound is tight, proving lower bounds appears to be beyond the reach of current techniques.

In the proportional noise model with adaptive adversary, we encounter another interesting phenomenon. Adjusting the upper bound analysis in a straightforward manner gives the trivial upper bound of $O(n)$, and for a good reason: an adaptive adversary can slow down the flooding time all the way to $\Omega(n)$, as shown in Theorem 13. The adversary for this lower bound makes only a few changes in each round, and only in necessary spots (using its adaptivity). The fact that the noise is proportional boils down to almost no noise occurring, which allows the adversary to maintain control over the network.

The second model of responsive noise we study is that of a *targeted noise*. Here, the expected amount of noise applied is roughly the same as above, but only the edges that undergo changes are perturbed by the noise. More concretely, each change proposed by the

adversary does not happen with some probability ϵ. The aim here is to model networks where every attempted change may have some probability of failure. In this setting, the case of an adaptive adversary seems more natural; however, we prove our upper bound for an adaptive adversary, and the lower bound for an oblivious one, thus handling the harder cases of both sides.

Our upper bound shows significant speedup in flooding on graphs with constant static diameter – $O(\log n)$ rounds suffice for flooding with targeted noise. This phenomenon holds even for higher static diameters: Theorem 14 gives an upper bound of $O((D \log_{1/\epsilon} n)/\epsilon^{D^2})$ rounds for flooding. This improves upon the trivial $O(n)$ whenever $D = O(\sqrt{\log_{1/\epsilon} n})$. Finally, in Theorem 15 we show that for larger static diameter, $D = \Theta(\log n)$, the number of rounds required for flooding is $\Omega(n)$.

Our techniques. Analysing our new models of smoothing require a new and more global techniques. While we adopt the results of [8] for changes in a single round, our models introduce new technical challenges, as they require multiple-round based analysis.

The main technical challenge for proving upper bounds comes from the fact that one cannot even guarantee that noise occurs at every round, and when it does – the amount of noise is not fixed through the execution of the whole algorithm. This requires us to conduct a much more global analysis, taking into account sequences of rounds with a varying amount of noise, instead of analyzing a single round at a time. In several cases, the static diameter D appears as a parameter in our results and analysis: in our model, the adversary can force change-free rounds where no noise occurs, in which case flooding happens on a static graph.

In an attempt to study the exact limitations of our noise models, we present specifically crafted lower bound networks for each model. Note that in many models our adversary is adaptive, potentially making it much stronger. This makes our lower bound more strategy-based, and not merely a fixed instance of a dynamic graph. We revise the original flooding lower bound of Dinitz et al., in order to make it more applicable to other models. We present a more detailed and rigorous proof, that indeed implies tight bounds in most of our models, and specifically, when considering adaptive adversaries.

1.3 Related work

Smoothed analysis was introduced by Spielman and Teng [18,19] in relation to using pivoting rules in the simplex algorithm. Since, it have received much attention in sequential algorithm design; see, e.g., the survey in [19]. The first application of smoothed analysis to the distributed setting is due to Dinitz et al. [8], who apply it to the well studied problems of aggregation, flooding and hitting time in dynamic networks, as described above. Chatterjee et al. [5] considered the problem of a minimum spanning tree construction in a distributed setting of a synchronous network, where the smoothing is in the form of random links added in each round, which can be used for communication but not as a part of the tree.

The field of distributed algorithm design for dynamic networks has received considerable attention in recent years [16]. Most of the works considered models of adversarial changes [1, 2, 4, 9, 11, 12, 14, 15], while others considered random changes, or random interactions between the vertices [3, 6, 7, 10, 13]. Yet, as far as we are aware, only the two aforementioned works take the smoothed analysis approach in this context.

2 Preliminaries

2.1 Dynamic graphs

All graphs are of the form $G = (V, E)$, where $V = [n] = \{1, \ldots, n\}$ is the set of vertices. A dynamic graph $\mathcal{H}$ is a sequence $\mathcal{H} = (G_1, G_2, \ldots)$ of graphs, $G_i = (V, E_i)$ on the same set of vertices, which can be finite or infinite. The *diameter* of a (static) graph is the largest distance between a pair of vertices in it, and the *static diameter* of a dynamic graph $\mathcal{H}$ is the minimal upper bound D on all the diameters of the graphs in the sequence $\mathcal{H}$.

We study synchronous distributed algorithms in this setting, where the time is split into rounds, and in round i the vertices communicate over the edges of the graph G_i.

Given two graphs $G = (V, E)$ and $G' = (V, E')$ on the same set of vertices, we denote $G - G' = (V, E \setminus E')$. We also denote $G \oplus G' = (V, E \oplus E')$, where $E \oplus E'$ is the set of edges appearing in exactly one of the graphs. The size of set $E \oplus E'$ is called the *Hamming distance between G and G'*, and is denoted by $|G \oplus G'|$.

2.2 Models of noise

Our smoothed analysis is based on three models of noise, defined in this section.

For a single step, we recall the definition of t-smoothing [8], and add the new notion of ϵ-smoothing, for a parameter $0 < \epsilon < 1$. At each step, we think of G_{old} as the current graph, and of G_{adv} as the future graph suggested by the adversary. The actual future graph, G_{new}, will be a modified version of G_{adv}, randomly chosen as a function of G_{old} and G_{adv}. In addition, we consider the set $\mathcal{G}_{\text{allowed}}$ of *allowed graphs* for a specific problem. For flooding, this is just the set of all connected graphs.

▶ **Definition 1.** *Let $0 \le \epsilon \le 1$ and $t \in \mathbb{N}$ be two parameters, $\mathcal{G}_{\text{allowed}}$ a family of graphs, and G_{old} and G_{adv} two graphs in $\mathcal{G}_{\text{allowed}}$.*

- *A t-smoothing of G_{adv} is a graph G_{new} which is picked uniformly at random from all the graphs of $\mathcal{G}_{\text{allowed}}$ that are at Hamming distance at most t from G_{adv}. The parameter t is called the* noise parameter.
- *An ϵ-targeted smoothing of a graph G_{adv} with respect to G_{old} is a graph G_{new} which is constructed from G_{adv} by adding to it each edge of $G_{\text{old}} - G_{\text{adv}}$ independently at random with probability ϵ, and removing each edge of $G_{\text{adv}} - G_{\text{old}}$ with the same probability. If the created graph is not in $\mathcal{G}_{\text{allowed}}$, the process is repeated.*

We are now ready to define the three types of smoothing for dynamic graphs. The first extends the definition of [8] for non-integer values $k \in \mathbb{R}_+$, and the other two incorporate noise that depends on the latest modifications in the graph.

For a positive real number x, we define the random variable $\text{round}(x)$, which takes the value $\lceil x \rceil$ with probability $x - \lfloor x \rfloor$ (the fractional part of x), and $\lfloor x \rfloor$ otherwise.

▶ **Definition 2.** *Let $0 \le \epsilon \le 1$ and $k \in \mathbb{R}_+$ be two parameters, and $\mathcal{G}_{\text{allowed}}$ a family of graphs.*

Let $\mathcal{H} = (G_1, G_2, \ldots)$ be a dynamic graph, i.e., sequence of "temporary" graphs. Let G'_0 be a graph (if G'_0 is not explicitly specified, we assume $G'_0 = G_1$).

- *A k-smoothed dynamic graph is the dynamic graph $\mathcal{H}' = (G'_0, G'_1, G'_2, \ldots)$ defined from $\mathcal{H}$, where for each round $i > 0$, G'_i is the t_i-smoothing of G_i, where $t_i = \text{round}(k)$.*
- *An ϵ-*proportionally *smoothed dynamic graph $\mathcal{H}' = (G'_0, G'_1, G'_2, \ldots)$ is defined from $\mathcal{H}$, where for each round $i > 0$, G'_i is the t_i-smoothing of G_i, where $t_i = \text{round}(\epsilon \cdot |G'_{i-1} \oplus G_i|)$.*

- *An* ϵ-targeted *smoothed dynamic graph is the dynamic graph* $\mathcal{H}' = (G'_0, G'_1, G'_2, \dots)$ *defined iteratively from* $\mathcal{H}$*, where for each round* $i > 0$*,* G'_i *is the* ϵ*-targeted smoothing of* G_i *w.r.t.* G'_{i-1}*.*

All the above definitions also have adaptive *versions, where each* $\mathcal{H}$ *and* $\mathcal{H}'$ *are defined in parallel: the graph* G_i *is chosen by an adversary after the graphs* $G_0, G_1, G_2, \dots, G_{i-1}$ *and* $G'_0, G'_1, G'_2, \dots, G'_{i-1}$ *are already chosen, and then* G'_i *is defined as above. We use adaptive versions for the first two definitions.*

2.3 Auxiliary lemmas

We use two basic lemmas, which help analyzing the probability of the noisy to add or remove certain edges from a given set of potential edges. These lemmas address a single round, we apply them with a different parameter in each round. The lemmas appeared as Lemmas 4.1 and 4.3 in [8], where they were used with the same parameter throughout the process.

▶ **Lemma 3.** *There exists a constant* $c_1 > 0$ *such that the following holds. Let* $G_{\text{adv}} \in \mathcal{G}_{\text{allowed}}$ *be a graph, and* $\emptyset \neq S \subseteq \binom{[n]}{2}$ *a set of potential edges. Let* $t \in \mathbb{N}$ *be an integer such that* $t \leq n/16$ *and* $t \cdot |S| \leq n^2/2$*.*

If G_{new} *is a* t*-smoothing of* G_{adv}*, then the probability that* G_{new} *contains at least one edge from* S *is at least* $c_1 \cdot t\,|S|\,/n^2$*.*

▶ **Lemma 4.** *There exists a constant* $c_2 > 0$ *such that the following holds.*

Let $G_{\text{adv}} \in \mathcal{G}_{\text{allowed}}$ *be a graph. Let* $S \subseteq E_{\text{adv}}$ *be a set of potential edges such that* $S \bigcap E_{\text{adv}} = \emptyset$*. Let* $t \in \mathbb{N}$*, such that* $t \leq n/16$*.*

If G_{new} *is an* t*-smoothing of* G_{adv}*, then the probability that* G_{new} *contains at least one edge from* S *is at most* $c_2 \cdot t\,|S|\,/n^2$*.*

2.4 Probability basics

In all our upper bound theorems, we show that flooding succeeds with high probability (w.h.p.), i.e., with probability at least $1 - n^{-c}$ for a constant c of choice. For simplicity, we prove success probability $1 - n^{-1}$, but this can be amplified by increasing the flooding time by a multiplicative constant factor. Our lower bounds show that flooding does not succeed with probability more than $1/2$, so it definitely does not succeeds w.h.p.

We will also use the following Chernoff bound (see, e.g. [17]):

▶ **Lemma 5.** *Suppose* $X_1, \dots, X_n$ *are independent Bernoulli random variables, with* $\mathbb{E}[X_i] = \mu$ *for every* i*. Then for any* $0 \leq \delta \leq 1$*:*

$$\Pr\left[\sum_{i=1}^{n} X_i \leq (1-\delta)\mu n\right] \leq e^{-\delta^2 \mu n/2}.$$

We usually apply this bound with $\delta = 0.9$.

3 Flooding in Dynamic Networks

3.1 Fractional amount of noise

We address an interesting phenomenon which occurs in [8]: by merely adding a single noisy edge per round, the flooding process dramatically speeds up from $\Theta(n)$ in the worst case, to only $\Theta(n^{2/3})$ in the worst case. To explain this gap, we present a natural notion of *fractional*

noise: if the amount of noise k is not an integer, we use an integer close to k, using the function round(k). An easy assertion of the result in [8], is that whenever $k > 1$, the analysis does not change by much, and the same asymptotic bound holds. This leaves open the big gap between $k = 0$ (no noise) and $k = 1$. We address this question in the next two theorems, showing a smooth transition in the worst-case flooding time between these two values of k.

Next, we revise the upper bound in [8] to handling fractional values of k and values smaller than 1. In addition, we show a somewhat cleaner argument, that carries three additional advantages: (1) It can be easily extended to adaptive adversaries; (2) It extends well to other models, as seen in subsequent sections; (3) It decreases the multiplicative logarithmic factor, dropping the asymptotic gap between upper and lower bound to only $O(\log^{1/3}(n))$. Hence, the proof of the next theorem serves as a prototype for later proofs, of upper bounds for adaptive adversaries and proportional noise.

▶ **Theorem 6.** *Fix a real number $0 < k \leq n/16$. For any k-smoothed dynamic graph, flooding takes $O\left(\min\left\{n, n^{2/3}(\log n/k)^{1/3}\right\}\right)$ rounds, w.h.p.*

Note that an n-round upper bound is trivial (using the connectivity guarantee), and that whenever $k \leq c \cdot \log n/n$ for some constant c, the term n is the smaller one, and we need not prove anything more.[1] We therefore focus on the event of $k > c \cdot \log n/n$, and for this case we show an upper bound of $O(n^{2/3}(\log n/k)^{1/3})$.

Proof. Fix u to be the starting vertex, and v to be some other vertex. We show that within $R = 3r$ rounds, the message has been flooded to v with constant probability (over the random noise). We split the $3r$ rounds into three phases as follows.

1. First r rounds, for spreading the message to a large set of vertices we call U.
2. Next r rounds, for transferring the message from the set U to another large set V to be defined next.
3. Last r rounds, for "converging" and passing the message from the potential set V to the specific vertex v.

We now quantify the message flooding in the three phases.

After the first r rounds, the message has reached many vertices. Let U_i denote the set of vertices that have been flooded at time i. Using merely the fact that the graph is connected at all times, and assuming the flooding process is not over, we know that at each round a new vertex receives the message, implying $|U_{i+1}| \geq |U_i| + 1$. Hence, $|U_r| \geq r$ (with probability 1).

In the last r rounds, many vertices can potentially flood the message to v. We denote by V_i the set of vertices from which v can receive the message within the last i rounds $(R - i + 1, \dots, R)$. Formally, fix the sequence of graphs chosen by the oblivious adversary, and apply the random noise on these graphs to attain the last r graphs of our *smoothed* dynamic graph. Define $V_0 = \{v\}$, and for $i > 0$, define V_i as the union of V_{i-1} with the neighbors of it in G_{R-i+1}. That is, V_i is defined analogously to U_i but with the opposite order of graphs. We point out that we are dealing with a stochastic process: each V_i is a random variable that depends on the noise in the last i rounds. Still, the connectivity guarantees that $|V_{i+1}| \geq |V_i| + 1$. Therefore, we have $|V_r| \geq r$ (with probability 1).[2]

---

[1] This is rather intuitive, as if k is very small (e.g., $k \leq c/n$), when considering δn rounds for small enough δ, no noise occurs. It is known that when no noise occurs, a linear amount of rounds is required for flooding. This intuition is formalized in the lower bound later in this section.

[2] Note that here we strongly rely on the obliviousness of the adversary: with an adaptive adversary, one cannot define V_r properly before the end of the execution of the algorithm, as the adversary's choices

The middle rounds. The above processes define two randomly chosen sets, U_r and V_r, each of size at least r. If $U_r \cap V_r \neq \emptyset$, then we are done as v is guaranteed to receive the message even if we ignore the middle rounds. Otherwise, we consider the set $S = U_r \times V_r$ of potential edges, and show that one of them belongs to our smoothed dynamic graph in at least one of the middle graphs, with probability at least $1 - n^{-2}$, for the right value of r.

Let us consider separately the two cases: $k \geq 1$ (note that non-integer k was not handled before), and the case $0 < k < 1$.

The case of $k \geq 1$. In this case, we essentially claim the following: we are guaranteed to have either $\lfloor k \rfloor$ noise or $\lceil k \rceil$ at each and every round. Applying Lemma 3 for each such round, the probability of not adding any edge from S is at most

$$\left(1 - c_1 \lfloor k \rfloor |S| / n^2\right) \leq \left(1 - c_1 k r^2 / 2n^2\right),$$

where the inequality follows from $\lfloor k \rfloor \geq k/2$ and $|S| = |U_r| \cdot |V_r| \geq r^2$. Thus, the probability of not adding any edge from S in any of these r noisy rounds is at most

$$\left(1 - c_1 k r^2 / 2n^2\right)^r \leq e^{-c_1 r^3 k / (2n^2)},$$

which is upper bounded by n^{-2}, whenever $r \geq n^{2/3} \cdot [(4 \log n)/(c_1 k)]^{1/3}$.

The case of $0 < k < 1$. Note that here we can no longer use $\lfloor k \rfloor \geq k/2$, and so we turn to a somewhat more global analysis which guarantees that "enough" rounds produce a (single) noisy edge: at each round we essentially add a noisy edge with probability $k < 1$, and otherwise do nothing. Since we have r rounds, and in each of them noise occurs independently, we can use a standard Chernoff bound to say that with all but probability at most $e^{-0.4kr}$, we have $(k/10)r$ rounds in which a noisy edge was added.[3] We later bound this term. For each of the $(k/10)r$ rounds we again apply Lemma 3 to claim that no edge from S was added, with probability at most

$$\left(1 - c_1 |S| / n^2\right) \leq \left(1 - c_1 r^2 / n^2\right),$$

where the inequality follows again from $|S| = |U_r| \cdot |V_r| \geq r^2$. Thus, the probability of not adding any edge from S in any of these $(k/10)r$ noisy rounds is upper bounded by

$$\left(1 - c_1 r^2 / n^2\right)^{(k/10)r} \leq e^{-c_1 k r^3 / (10n^2)},$$

which is upper bounded by $1/(2n^2)$ whenever $r \geq n^{2/3} \cdot [(10 \log n)/(c_1 k)]^{1/3}$.

Recalling that we only deal with the case $k \geq c \cdot \log n / n$, choosing the constant c according to the constant c_1, we also get $r \geq 10 \log n / k$. This allows us to bound the error of the Chernoff inequality: $e^{-0.4kr} \leq e^{-3 \log n} \leq 1/(2n^2)$. Union bounding on both possible errors, we know that with probability at least $1 - n^{-2}$ the vertex v is informed after $3r$ rounds.

To conclude, we use $r = n^{2/3} \cdot [(10 \log n)/(c_1 k)]^{1/3}$. For any value $k \geq c \cdot \log n / n$, and for any realization of U_r and V_r, the message has been passed to V_r within the r middle rounds, with probability at least $1 - n^{-2}$. So v received the message after $R = 3r$ rounds with the same probability. Taking a union bound over all the vertices implies that R rounds are enough to flood to the whole network with probability at least $1 - 1/n$. ◀

---

were not yet made. The case of an adaptive adversary is discussed in the next section.

[3] The expectation over all rounds is kr noisy edges.

We also show a matching lower bound, restructuring the proof of the lower bound in [8]. We note that their proof actually states a lower bound of $\Omega\left(min\left\{n/k, n^{2/3}/k^{1/3}\right\}\right)$ that apply to any $k < n/16$ (and is indeed tight for $k < O(\sqrt{n})$). We restructure the analysis of the lower bound to fit the different constructions given in the following sections, as follows.

First, we consider the first R rounds, for some parameter R, and show inductively that the following two main claims continue to hold for every round $i < R$ with very high probability.

- Some pre-defined set of vertices stays uninformed
- Given the above, the *expected* number of informed vertices in total does not grow rapidly.

The growth (in the expected number of informed vertices) has a multiplicative-additive progression, potentially becoming an exponential growth (with a small exponential).

We choose R such that the expected number of informed vertices is upper bounded by δn, and use Markov inequality to show that with high probability the flooding is not yet over after R rounds. We then apply a union bound over all the inductive steps, where each has a small probability to fail the inductive argument. Altogether, we show that with high probability, the flooding is not over after R rounds.

In the full version of this paper, we use this argument in order to prove the following extensions of the lower bound from [8] to non-integer values and to fractional values (where $k < 1$).

▶ **Theorem 7.** *Fix* $1 \le k \le n/16$ *(not necessarily an integer). For any k-smoothed dynamic graph, for the flooding process to succeed with probability at least* $1/2$*, it must run for* $\Omega\left(\min\left\{n/k, n^{2/3}/k^{1/3}\right\}\right)$ *rounds.*

In particular, whenever $k = O(\sqrt{n})$, the dominant term is $\Omega(n^{2/3}/k^{1/3})$, matching the upper bound (up to logarithmic factors).

▶ **Theorem 8.** *Fix* $0 < k < 1$*. For any k-smoothed dynamic graph, for the flooding process to succeed with probability at least* $1/2$*, it must run for* $\Omega\left(\min\left\{n, n^{2/3}/k^{1/3}\right\}\right)$ *rounds.*

3.2 Adaptive vs. oblivious adversary

Here we note that the results of [8] are for the case of oblivious (non-adaptive) adversary. We extend our results (in the more generalized, fractional noise regime) to the adaptive case, obtaining bounds that are different than the ones in [8]. Interestingly, our results show separation between adaptive and oblivious adversary for a wide range of network noise: for constant k, in particular, we get a separation for the adaptive case, where no constant amount of noise speeds up the flooding below $\Omega(n)$, and the oblivious case where $k = \omega(1/n)$ already speeds the flooding to $o(n)$ rounds.

▶ **Theorem 9.** *Fix* $0 < k \le n/16$*, not necessarily an integer. For any k-smoothed* adaptive *dynamic graph, flooding takes* $O(\min\left\{n, n\sqrt{\log n/k}\right\})$ *rounds, w.h.p.*

The proof follows an argument similar to the one of Theorem 6, where the process is broken to three phases: u spreads the message to U_r, which carry it to V_r, who are sure to deliver it to the destination v. The major difference here is that we can no longer rely on the last phase, since an adaptive adversary *sees* the informed vertices and is able to act accordingly. We thus turn to a statistical analysis of the second phase instead, where the informed set U_r delivers the message directly to v. The full proof appears in the full version of this paper.

When k is a constant, the above result is tight, which we prove by adjusting the oblivious dynamic network from Theorem 7 to use the adversary's adaptivity, as follows. The basic structure of the hard instance for flooding is achieved by roughly splitting the graph into

informed and *uninformed* vertices. In order to ensure low diameter, all the uninformed vertices are connected as a star, centered at a changing uninformed vertex called a *head*, which in turn connects them to a star composed of the informed vertices. The key idea in the analysis of this process, is that the head at each round should not become informed via noisy edges at an earlier round, as in this case it will immediately propagate the message to all the uninformed vertices, completing the flooding. An oblivious adversary picks a sequence of heads at the beginning of the process, and this invariant is kept since the probability of any of the selected heads to get informed too early is low. However, after roughly $n^{2/3}$ rounds, the probability that a selected head is informed becomes too high. An adaptive adversary, on the other hand, can continue crafting a hard instance graph for a linear number of rounds – in each round, the adversary knows which vertices are uninformed and can pick one of them as the new head, thus overcoming this obstacle.

▶ **Theorem 10.** *Fix $0 < k \leq n/16$, not necessarily an integer. For any k-smoothed* adaptive *dynamic graph, for the flooding process to succeed with probability at least $1/2$ it must run for $\Omega(\min\{n, n\log k/k\})$ rounds.*

Proof. We start by formally defining a hard case for flooding with noise. The base to our definition is the *spooling graph*, which was defined by Dinitz et al. [8]: at time i, the edges are $\{(j,i)\}_{j=1}^{i-1}$, a star around the vertex i (this will be the informed spool) and edges $\{(i+1,j)\}_{j=i+2}^{n}$, a star around the vertex $i+1$, which will be the right spool, with vertex $i+1$ crowned as the *head*. the spools are then connected using the additional edge $(i, i+1)$.

We define the *adaptive spooling graph*, as follows: at first $E_1 = ([n] \setminus \{2\}) \times \{2\}$. After the first iteration, 2 learns the message and is removed to the left (informed) spool. We denote for each round i the set of informed vertices at the end with I_i. We also use u_i for the lowest index of an uninformed vertex at the end of round i. Formally $u_i = \min\left\{\bar{I_i}\right\}$. We define the adaptive spooling graph at time $i+1$ to have the edge set

$$E_{i+1} = \{1\} \times (I_i \setminus \{1\}) \;\cup\; \{(1, u_i)\} \;\cup\; \{u_i\} \times (\bar{I_i} \setminus \{u_i\}).$$

In this graph, essentially, 1 is connected to all already-informed vertices, and also to the next head u_i, which is connected to all other uninformed vertices. The static diameter stays 3 at each iteration, but now we are promised that at iteration $i+1$, the new head u_i is uninformed at the beginning of said round.

We next show that the *expected* number of informed players cannot grow by much.

▷ Claim 11. When taking expectation over the noise we add, we have $\mathbb{E}[|I_{i+1}|] \leq (1 + c_2k/n) \cdot \mathbb{E}[|I_i|] + 1$.

Note that I_i does not depend on the noise of rounds $i+1, \ldots, R$, and so the left side takes expectation over $i+1$ rounds of noise, and the right side over i rounds.

Intuitively, the claim states that the expected growth in the number of *informed* vertices is bounded by an additive-multiplicative progression (i.e., at each step the amount grows multiplicatively and then a constant is added). This is true as the noisy edges induce a multiplicative growth, and connectivity forces that at least 1 additional vertex (u_i itself) receives the message. The proof of this claim can be found in the full version.

Using the claim, we analyze the progression of $A_i = \mathbb{E}[|I_i|]$, splitting to 2 cases:

1. For $A_i \leq n/(c_2k)$, the additive term is larger, and so in this range $A_{i+1} \leq A_i + 2$.
2. For $A_i \geq n/(c_2k)$, the multiplicative term overcomes and we get $A_{i+1} \leq (1 + 2c_2k/n)A_i$.

Note that we also have $A_{i+1} \geq A_i + 1$, using connectivity, showing this bounds on the progression are rather tight.

We next split into cases: if $k < 1/c_2$, the additive term controls the process through $\Theta(n)$ iterations, giving us $A_{n/20} \leq n/10$. Otherwise, the multiplicative term would come into play. We denote the number of additive rounds by r_0: the minimal index such that $A_{r_0} \geq n/(c_2 k)$. Since $A_0 = 1$ (only the vertex 1 knows the message at the beginning), and we have good bounds on the progression, we conclude $r_0 = \Theta(n/k)$.

Next, we allow additional r_1 rounds in which the multiplicative term is dominant. We get

$$A_{r_0+r_1} \leq A_{r_0}(1 + 2c_2k/n)^{r_1} \leq \Theta(n/k)(1 + 2c_2k/n)^{r_1}.$$

Taking $r_1 = \delta(n \log k/k)$ with small enough δ, we have $A_{r_0+r_1} = \mathbb{E}\left[|I_{r_0+r_1}|\right] \leq n/10$.

In both cases, there is a round R with $\mathbb{E}\left[|I_R|\right] \leq n/10$. By Markov inequality, strictly less than n vertices are informed after R rounds, w.p. at least 0.9. Note that for the first case $R = \Theta(n)$, and for the second case $R = r_0 + r_1 = O(n \log k/k)$, concluding the proof. ◀

4 Responsive noise

In this section we consider *responsive* noise in the dynamic network, where some elements of the noise incurred in each round relate to the changes done at this round. We consider this model as complement to one discussed in the previous section: the parameter k of "noise per round" is fit to model some internal background noise in a system. However, in a more realistic model we would expect the noise to work in different patterns in times of major changes in the network, as opposed to rounds where no changes were made at all.

To this end, we introduce two variants of a responsive noise: one is that of *proportional* noise, and the other is *targeted* noise. In the first, the amount of noisy edges would relate to the amount of changes that occurred in the last step. In the second variant, we expect the noise to specifically "target" newly modified edges. This could relate to a somewhat limited adversary (in aspect of ability to change the graph).

In the responsive model, an interesting phenomenon occurs: an adversarial network can choose to stay still, in order to force a round where no noise occurs. For the flooding problem, this strength is limited: whenever the static diameter of each iteration is upper bounded by D, the waiting game can only last $D - 1$ rounds. To that end, we show how this model affects the analysis of the upper bound, and yet again incorporate the new phenomenon described to devise a non-trivial lower bound.

4.1 Proportional noise

▶ **Theorem 12.** *Fix $0 < \epsilon$. For any ϵ-proportionally smoothed dynamic graph with static diameter at most D. If no noise invokes more than $n/16$ changes, the flooding process finishes after $O(n^{2/3} \cdot (D \log n/\epsilon)^{1/3})$ rounds, w.h.p.*

The proof resembles the proof of Theorem 6 with three phases, which handles the "waiting game" mentioned above. Given static diameter D, an adversarial network can only stay intact for $D - 1$ rounds, thus in the second phase at least $1/D$ of the rounds introduce one changed edge (or more), inferring that w.h.p. noise is incurred in $\Omega(\epsilon/D)$ rounds. The proof appears in the full version of this paper.

Adjusting the same argument for an *adaptive* network, using only two phases (same as Theorem 9), would give an upper bound of $O(n \cdot \sqrt{D \log n/\epsilon})$, which in this case does not improve upon the trivial $O(n)$ promised by connectivity alone.

We continue to show that this is tight, and in the proportional noise model, the changes in the network can be so subtle that they would invoke little noise and the flooding would barely speed up at all, asymptotically. We note that the following construction uses constant D, where it is hardest to manipulate.

▶ **Theorem 13.** *Fix $\epsilon \leq 1/5$. There exists an ϵ-proportionally smoothed dynamic graph with an adaptive adversary, where a flooding process must run for $\Omega(n)$ rounds in order to succeed with probability at least $1/2$.*

In the current, adaptive model, the spooling graph no longer stands as a slow-flooding graph. Changing the center of the uninformed star takes $\Omega(n)$ edge changes in early rounds, and the proportional noise invoked by these changes would be devastating (e.g., for $\epsilon = \Omega(1)$).

Instead, in the proof of this theorem, which appears in the full version of our paper, the adversary relies on its adaptivity in order to maintain control over the network. This is done using a sequence of graphs that require only $O(1)$ changes at each round.

We turn to a variant of the spooling graph, composed of two star graphs, one is made of informed vertices and one of uninformed vertices. The last vertex to be informed is the center of the uninformed vertices star, which remains the same vertex throughout the process. Connectivity is kept using a connector vertex, which changes each round. Every time a connector learns the message, it gets disconnected from the uninformed star moved to the informed star, and another vertex becomes the connector.

As before, we need to deal with noisy edges that allow random vertices to learn the message, even when these are not the designated connector vertices. Here, we utilize the adaptivity once more: every time this happens, we immediately remove the newly informed vertex from the uninformed star and add it to the informed one, keeping the center of the uninformed star ignorant of the message. The only bad case is when the center vertex of the uninformed star gets connected to an informed vertex at random, but this happens with very low probability.

The key observation in our analysis is that handling a randomly informed vertex only costs additional $O(1)$ changes. and even combined with the planned changes to the network, one can upper bound the amount of necessary changes at each round. This creates a positive feedback loop: the adversary can maintain the desired structure of the graph, using very few changes, and provoking almost no noise.

Finally, we note that each round at most one vertex learns the message due to a randomly added edge, in addition to the connector. Thus, only a constant amount of vertices (in fact, 2) can become informed in each round, implying the need for a linear number of rounds.

4.2 Targeted noise

For targeted noise this new phenomenon is surpassed by the limited ability of the adversary to make changes. We can think of targeted noise as some sort of "slow down", as if the network repeatedly tries to modify some of the edges, eventually succeeding, but not before a number of rounds has passed. In this last model we show just how strong the waiting game can be: for a graph with constant static diameter (which makes the waiting game obsolete), flooding will take $O(\log n)$ rounds, with high probability. However, for larger value of D, the same analysis would quickly fail. For $D = \Theta\left(\sqrt{\log n}\right)$ we get the trivial bound of $O(n)$. We finish by showing an explicit construction with static diameter $D = \Theta\left(\log n\right)$ that relies strongly on the waiting game and admits a lower bound of $\Omega(n)$ rounds.

▶ **Theorem 14.** *For any ϵ-targeted smoothed dynamic graph with static diameter D, flooding can be done in $O(D \log n / \epsilon^{D^2})$ rounds, w.h.p.*

Note that for $D = o(\sqrt{\log_{1/\epsilon} n})$, this improves upon the trivial fact that n rounds are enough (as at each round, one new vertex must be informed, since the graph is connected). Specifically, for a constant static diameter, we get $O(\log n)$ rounds.

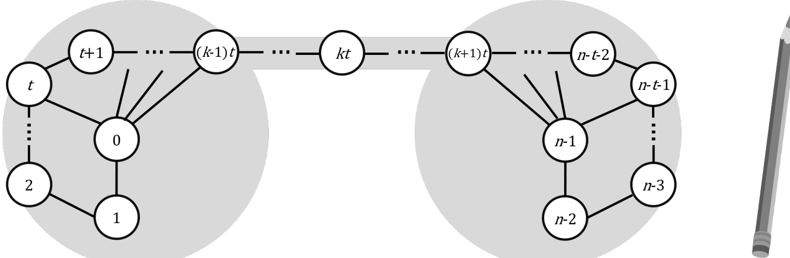


Figure 1 The cassette graph, G_j^t, where $j = kt$ and n is some multiple of t.

Proof. Fix the starting vertex v_0 which is informed. First, we show that the probability of a vertex $v \neq v_0$ staying uninformed after D rounds is at most $1 - \epsilon^{D^2}$.

Consider D consecutive graphs $G_1, \ldots, G_D$ in the smoothed graph. As the static diameter is D in every round, there exists a path P_v from v_0 to v in G_1, whose length is at most D. Since we deal with targeted smoothing, each edge of P_v that exists in a graph G_i for some $1 \leq i < D$ exists in G_{i+1} probability either 1 or ϵ. So, for each such i, if all the edges of P_v exist in G_i then they all exist in G_{i+1} with probability at least ϵ^D. Hence, the probability of the path P_v existing in all D graphs is at least $(\epsilon^D)^D = \epsilon^{D^2}$.

Fix a positive integer t, and consider t consecutive sequences of D graphs each. On each sequence, we apply the above claim, and conclude that the probability of v staying uninformed after theses tD rounds is at most

$$(1 - \epsilon^{D^2})^t \leq e^{-t \cdot \epsilon^{D^2}}.$$

For $t = (c+1) \log n \cdot (1/\epsilon)^{D^2}$ sequences, the probability of v not being informed after tD rounds is at most $n^{-(c+1)}$. A union bound over all $n-1$ vertices that need to be informed implies that tD rounds suffice to fully flood the network with probability at least $1 - n^{-c}$. ◀

The above theorem implies that if the static diameter of all graphs in the sequence is small, roughly $O(\sqrt{\log n})$, flooding is fast. Next, we show that this is almost tight: if the diameter is in $\Omega(\log n)$, flooding cannot be done faster than in non-smoothed graphs.

▶ **Theorem 15.** *For every constant $0 < \epsilon < 1$, there is a value $D \in \Theta(\log n)$ and an ϵ-targeted smoothed dynamic graph such that with high probability, the diameter of the graph is D and flooding on it takes $n - 1$ rounds.*

In order to prove this theorem, we present the *dynamic cassette graph* (see Figure 1). Fix ϵ and n as in the theorem statement, and let $t = \lfloor c \log_{1/\epsilon} n \rfloor$ for a constant c of choice. The dynamic cassette graph on vertices $V = \{0, \ldots, n-1\}$ is the dynamic graph $\mathcal{H} = \{G_1, \ldots, G_n\}$, where $G_i = (V, E_i)$ is defined by

$$\begin{aligned} E_i = &\{(j, j+1) \mid 0 \leq j < n-1\} \cup \\ &\{(0, jt) \mid 1 \leq j \leq \lfloor (i-1)/t \rfloor\} \cup \{(jt, n-1) \mid \lfloor (i-1)/t \rfloor + 2 \leq j \leq \lfloor (n-2)/t \rfloor\}. \end{aligned}$$

This graph is the path on n vertices, with some additional edges connecting the first and last vertices to vertices in the set $\{jt \mid 1 \leq j \leq (n-2)/t\}$; these will be referred to as *shortcut vertices*, and the additional edges to them, *shortcut edges*. At the first graph, G_1, all shortcut vertices but the first are connected to the last vertex, $n-1$. Then, one by one, the shortcut vertices disconnect from $n-1$, and soon after – connect to 0. At each time interval $[(j-1)t+1, jt]$, all the shortcut vertices with index strictly smaller than jt are connected to the vertex 0, and all those with index strictly higher than jt are connected to $n-1$.

Consider the *smoothed cassette graph* $\mathcal{H}'$, i.e., the ϵ-targeted smoothed dynamic graph derived from $\mathcal{H}$. The dynamic graph $\mathcal{H}'$ can be interpreted as undergoing the following process: during each time interval $[(j-1)t+1, jt]$, the adversary repeatedly tries to add a new edge $(0,(j-1)t)$, and remove the edge $((j+1)t, n-1)$. The targeted noise creates a slowdown that might prevent this from happening right away, yet for the right value of t, both changes indeed happen by the end of the time interval w.h.p. We state the following claim and direct the reader to the full version for its proof.

▷ Claim 16. For each $2 \le j \le (n-2)/t$, the smoothed graph G'_{jt} does not contain the edge $(0,(j-1)t)$ with probability at most n^{-c}, and contains the edge $((j+1)t, n-1)$ with the same probability.

If the edge $(0,(j-1)t)$ exists in the smoothed graph G'_{jt} then it also exists in all later graphs, $G_{j'}$ with $j' > jt$. Similarly, if the edge $((j+1)t, n-1)$ does not exist in this graph, it also does not appear in later graphs. A union bound thus extends the last claim as follows.

▷ Claim 17. For each $2 \le j \le (n-2)/t$, the smoothed graph G'_{jt} and all later graphs contain the edge $(0,(j-1)t)$ with probability at least $1-n^{-c+1}$, and all these graphs do not contain the edge $((j+1)t, n-1)$ with the same probability.

Using this claim, Theorem 15 can easily be proven.

Proof of Theorem 15. Consider the smoothed dynamic cassette graph $\mathcal{H}'$. We start by analyzing its diameter.

Let $G'_{j'}$ be a graph in $\mathcal{H}'$, and pick a j such that $jt \le j' < (j+1)t$. By Claim 17, the graph G'_{jt} and all later graphs contain the edge $(0,(j-1)t)$ with probability at least $1-n^{-c+2}$. In addition, all the graphs $G_1, \ldots, G_{(j+1)t-1}$ contain the edge $((j+2)t, n-1)$ (and note that this is not a probabilistic claim).

The distance between every two shortcut vertices is t, so the distance from every vertex in the graph to the closest shortcut vertex is at most $t/2$. Each shortcut vertex is directly connected to either 0 or $n-1$ w.h.p., except for jt, who is connected to both by a $t+1$ path. Finally, between the vertices there is a path of length $2t+2$, through $(j-1)t$ and $(j+1)t$. Let us bound the length of a path between two vertices i, i' (with upper bounds on each part): from i to its closest shortcut vertex ($t/2$ hops), to 0 or $n-1$ ($t+1$ hop), maybe to the other vertex between 0 and $n-1$ ($2t+2$ hops), to the shortcut vertex closest to i' ($t+1$ hop), and to i' ($t/2$ hops). This sums to a path of length at most $5t+4 = \Theta(\log n)$. A more detailed analysis can reduce this to roughly $3t$ hops.

For the flooding time, we use Claim 17 again, but now for the edges $((j+1)t, n-1)$ that do not appear G'_{jt} and all later graphs w.h.p. A simple induction on $j' = 0, \ldots, n-1$ shows that after j' rounds, i.e. in graph $G_{j'}$, only vertices $0, \ldots, j'$ are informed. The base case is trivial. For the step, the only edges connecting informed vertices to uninformed vertices are $(j', j'+1)$, and edges from 0 to shortcut vertices, which have the form $(0, tj)$ with $tj \le j'$ – this yields from the construction and always holds. The only other type of possible edges connecting informed and uninformed vertices are of the form $(jt, n-1)$, with $jt \le j'$. However, the claim implies that w.h.p., by round j' none of these edges exist. Hence, before round $n-1$ not all vertices are informed, and flooding takes $n-1$ rounds w.h.p. ◀

References

1. John Augustine, Gopal Pandurangan, Peter Robinson, and Eli Upfal. Towards robust and efficient computation in dynamic peer-to-peer networks. In *SODA*, pages 551–569. SIAM, 2012.
2. Philipp Bamberger, Fabian Kuhn, and Yannic Maus. Local distributed algorithms in highly dynamic networks. In *IPDPS*, pages 33–42. IEEE, 2019.
3. Leran Cai, Thomas Sauerwald, and Luca Zanetti. Random walks on randomly evolving graphs. In *Structural Information and Communication Complexity - 27th International Colloquium, SIROCCO*, volume 12156 of *Lecture Notes in Computer Science*, pages 111–128. Springer, 2020.
4. Keren Censor-Hillel, Neta Dafni, Victor I. Kolobov, Ami Paz, and Gregory Schwartzman. Fast and simple deterministic algorithms for highly-dynamic networks. *CoRR*, abs/1901.04008, 2019.
5. Soumyottam Chatterjee, Gopal Pandurangan, and Nguyen Dinh Pham. Distributed MST: A smoothed analysis. In *ICDCN 2020: 21st International Conference on Distributed Computing and Networking*, pages 15:1–15:10. ACM, 2020. doi:10.1145/3369740.3369778.
6. Andrea E. F. Clementi, Riccardo Silvestri, and Luca Trevisan. Information spreading in dynamic graphs. *Distributed Computing*, 28(1):55–73, 2015.
7. Oksana Denysyuk and Luís E. T. Rodrigues. Random walks on evolving graphs with recurring topologies. In *DISC*, volume 8784 of *Lecture Notes in Computer Science*, pages 333–345. Springer, 2014.
8. Michael Dinitz, Jeremy T Fineman, Seth Gilbert, and Calvin Newport. Smoothed analysis of dynamic networks. *Distributed Computing*, 31(4):273–287, 2018.
9. Chinmoy Dutta, Gopal Pandurangan, Rajmohan Rajaraman, Zhifeng Sun, and Emanuele Viola. On the complexity of information spreading in dynamic networks. In *SODA*, pages 717–736. SIAM, 2013.
10. Leszek Gasieniec and Grzegorz Stachowiak. Fast space optimal leader election in population protocols. In Artur Czumaj, editor, *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 2653–2667. SIAM, 2018.
11. Mohsen Ghaffari, Nancy A. Lynch, and Calvin C. Newport. The cost of radio network broadcast for different models of unreliable links. In *PODC*, pages 345–354. ACM, 2013.
12. Bernhard Haeupler and David R. Karger. Faster information dissemination in dynamic networks via network coding. In *PODC*, pages 381–390. ACM, 2011.
13. Dariusz R. Kowalski and Miguel A. Mosteiro. Polynomial counting in anonymous dynamic networks with applications to anonymous dynamic algebraic computations. *Journal of the ACM*, 67:1–17, 2020.
14. Fabian Kuhn, Nancy A. Lynch, and Rotem Oshman. Distributed computation in dynamic networks. In *STOC*, pages 513–522. ACM, 2010.
15. Fabian Kuhn, Yoram Moses, and Rotem Oshman. Coordinated consensus in dynamic networks. In *Proceedings of the 30th Annual ACM Symposium on Principles of Distributed Computing, PODC*, pages 1–10. ACM, 2011. doi:10.1145/1993806.1993808.
16. Othon Michail. An introduction to temporal graphs: An algorithmic perspective. *Internet Math.*, 12(4):239–280, 2016.
17. Michael Mitzenmacher and Eli Upfal. *Probability and computing — randomized algorithms and probabilistic analysis*. Cambridge University Press, 2005.
18. Daniel A. Spielman and Shang-Hua Teng. Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time. *J. ACM*, 51(3):385–463, 2004.
19. Daniel A. Spielman and Shang-Hua Teng. Smoothed analysis: an attempt to explain the behavior of algorithms in practice. *Commun. ACM*, 52(10):76–84, 2009.

Distributed Maximum Matching Verification in CONGEST

Mohamad Ahmadi
University of Freiburg, Germany
mahmadi@cs.uni-freiburg.de

Fabian Kuhn
University of Freiburg, Germany
kuhn@cs.uni-freiburg.de

Abstract

We study the maximum cardinality matching problem in a standard distributed setting, where the nodes V of a given n-node network graph $G = (V, E)$ communicate over the edges E in synchronous rounds. More specifically, we consider the distributed CONGEST model, where in each round, each node of G can send an $O(\log n)$-bit message to each of its neighbors. We show that for every graph G and a matching M of G, there is a randomized CONGEST algorithm to *verify* M being a maximum matching of G in time $O(|M|)$ and disprove it in time $O(D + \ell)$, where D is the diameter of G and ℓ is the length of a shortest augmenting path. We hope that our algorithm constitutes a significant step towards developing a CONGEST algorithm to *compute* a maximum matching in time $\tilde{O}(s^*)$, where s^* is the size of a maximum matching.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases distributed matching, distributed graph algorithms, augmenting paths

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.37

Related Version A full version of the paper is available at `https://arxiv.org/abs/2002.07649`.

1 Introduction and Related Work

A matching $M \subseteq E$ of a graph $G = (V, E)$ is a set of pairwise disjoint edges and the maximum matching problem asks for a matching M of maximum cardinality (or of maximum weight if the edges are weighted). Matchings have been at the center of attention in graph theory for more than a century (see, e.g., [38]). Algorithmic problems dealing with the computation of matchings are among the most extensively studied problems in algorithmic graph theory. The problem of finding a maximum matching is on the one hand simple enough so that it can be solved efficiently [16, 17], on the other hand the problem has a rich mathematical structure and led to many important insights in graph theory and theoretical computer science. Apart from work in the standard sequential setting, the problem has been studied in a variety of other settings and computational models. Exact or approximate algorithms have been developed in areas such as online algorithms (e.g., [18, 32]), streaming algorithms (e.g., [41]), sublinear-time algorithms (e.g., [40, 47]), classic parallel algorithms (e.g., [23, 31]), as well as also the recently popular massively parallel computation model (e.g., [5, 12, 25]). In this paper, we consider the problem of verifying whether a given matching is a maximum matching in a standard distributed setting, which we discuss in more detail next.

Distributed maximum matching. In the distributed context, the maximum matching problem is mostly studied for networks in the following synchronous message passing model. The network is modeled as an undirected n-node graph $G = (V, E)$, where each node hosts a distributed process and the processes communicate with each other over the edges of G. We assume in a distributed context that matching M is given as an input to or computed

© Mohamad Ahmadi and Fabian Kuhn;
licensed under Creative Commons License CC-BY
34th International Symposium on Distributed Computing (DISC 2020).
Editor: Hagit Attiya; Article No. 37; pp. 37:1–37:18
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

as an output by G when every node of V knows its adjacent edges in M. As it is common practice, we identify the nodes with their processes and think of the nodes themselves as the distributed agents. Time is divided into synchronous rounds and in each round, each node $v \in V$ can perform some arbitrary internal computation, send a message to each of its neighbors in G, and receive the messages of the neighbors (round r is assumed to start at time $r-1$ and end at time r). If the messages can be of arbitrary size, this model is known as the LOCAL model [35, 44]. In the more realistic CONGEST model [44], in each round, each node can send an arbitrary $O(\log n)$-bit message to each of its neighbors.

Our contribution. As the main result of our paper, we give a distributed maximum matching verification algorithm.

▶ **Theorem 1.** *Given an undirected graph $G = (V, E)$ and a matching M of G, there is a randomized distributed* CONGEST *model algorithm to test whether M is a maximum matching. If M is a maximum matching, the algorithm verifies this in time $O(|M|)$, otherwise, the algorithm disproves it in time $O(D + \ell)$, where D is the diameter of G and ℓ is the length of a shortest augmenting path.*

Note that the size of a maximum matching is always $\Omega(D)$, so we are not trying to perform in sub-diameter time. Our main technical contribution is a distributed algorithm that, given a matching M and a parameter x, determines if there is an augmenting path of length at most x in $O(x)$ rounds of the CONGEST model. If there is an augmenting path of length at most x, the algorithm identifies two free (i.e., unmatched) nodes u and v between which such a path exists. We note that if the algorithm can be extended to also *construct* an augmenting path of length at most x between u and v in time $\tilde{O}(x)$, it would directly lead to an $\tilde{O}(s^*)$-round algorithm for computing a maximum matching, where s^* is the size of a maximum matching. The reason for this follows from the classic framework of Hopcroft and Karp [29]. It is well-known that if we are given a matching M of size $s^* - k$ for some integer $k \geq 1$, there is an augmenting path of length less than $2s^*/k$ [29]. Hence, if we can find such a path and augment along it in time linear in the length of the path, we get a total time of $O(s^* \log s^*)$ by summing over all values of k from 1 to s^*. This approach has been used in [2] to compute a maximum matching in time $O(s^* \log s^*)$ in bipartite graphs in the CONGEST model and it could also be employed without further difficulties for general graphs if we assume that we could actually construct the detected shortest augmenting paths. While finding a shortest augmenting path is quite straightforward in bipartite graphs, getting an efficient CONGEST model algorithm for general graphs turns out to be much more involved. We therefore hope that our algorithm for finding the length and the endpoints of some shortest augmenting path provides a significant step towards also efficiently constructing a shortest augmenting path in the CONGEST model and therefore to obtaining an $\tilde{O}(s^*)$-time CONGEST algorithm to find a maximum matching. Next we give a brief summary of the history of the distributed maximum matching problem.

Distributed maximal matching algorithms. While except for [2, 10], there is no previous work on exact solutions for the distributed maximum matching problem, there is a very extensive and rich literature on computing approximate solutions for the problem. The most basic way to approximate maximum matching is by computing a *maximal matching*, which provides a 1/2-approximation for the maximum matching problem. The work on distributed maximal matching algorithms started with the classic randomized parallel maximal matching and maximal independent set algorithms from the 1980s [3, 30, 39]. While these

algorithms were originally described for the PRAM setting, they directly lead to randomized $O(\log n)$-round algorithms in the CONGEST model. It was later shown by Hańćkowiak, Karoński, and Panconesi [26, 27] that maximal matching can also be solved deterministically in polylogarithmic time in the distributed setting. The current best deterministic algorithm in the CONGEST model (and also in the LOCAL model) is by Fischer [21] and it computes a maximal matching in $O(\log^2 \Delta \log n)$ rounds, where Δ is the maximum degree of the network graph G. At the cost of a higher dependency on Δ, the dependency on n can be reduced and it was shown by Panconesi and Rizzi [43] that a maximal matching can be computed in $O(\Delta + \log^* n)$ rounds. The best known randomized algorithm is by Barenboim et al. [9] and it shows that (by combining with the result of [21]) a maximal matching can be computed in $O(\log \Delta) + O(\log^3 \log n)$ rounds in the CONGEST model. The known bounds in the CONGEST model are close to optimal even when using large messages. It is known that there is no randomized $o\big(\frac{\log \Delta}{\log \log \Delta} + \sqrt{\frac{\log n}{\log \log n}}\big)$-round maximal matching algorithm in the LOCAL model [34]. A very recent result further shows that there are also no randomized $o\big(\Delta + \frac{\log \log n}{\log \log \log n}\big)$-round algorithm and no deterministic $o\big(\Delta + \frac{\log n}{\log \log n}\big)$-round algorithms to compute a maximal matching [7].

Distributed maximum matching approximation algorithms. There is a series of papers that target the distributed maximum matching problem directly and that provide results that go beyond the 1/2-approximation achieved by computing a maximal matching. Most of them are based on the framework of Hopcroft and Karp [29]: after $O(1/\varepsilon)$ iterations of augmenting along a (nearly) maximal set of vertex-disjoint short augmenting paths, one is guaranteed to have a $(1-\varepsilon)$-approximate solution for the maximum matching problem. The first distributed algorithms to use this approach are an $O(\log^{O(1/\varepsilon)} n)$-time deterministic LOCAL algorithm for computing a $(1-\varepsilon)$-approximation in graphs of girth at least $2/\varepsilon - 2$ [13] and an $O(\log^4 n)$-time deterministic LOCAL algorithm for computing a 2/3-approximation in general graphs [14]. The first approximation algorithms in the CONGEST model are by Lotker et al. [36], who give a randomized algorithm to compute a $(1-\varepsilon)$-approximate maximum matching in time $O(\log n)$ for every constant $\varepsilon > 0$. For bipartite graphs, the running time of the algorithm depends polynomially on $1/\varepsilon$, whereas for general graphs it depends exponentially on $1/\varepsilon$.[1] The algorithm was recently improved by Bar Yehuda et al. [8], who give an algorithm with time complexity $O\big(\frac{\log \Delta}{\varepsilon^3 \log \log \Delta}\big)$ for computing a $(1-\varepsilon)$-approximation. As in [36], the time depends polynomially on $1/\varepsilon$ in bipartite graphs and exponentially on $1/\varepsilon$ in general graphs. Note that the time dependency on Δ in [8] matches the lower bound of [34]. In [2], Ahmadi et al. give a deterministic $O\big(\frac{\log \Delta}{\varepsilon^2} + \frac{\log^2 \Delta}{\varepsilon}\big)$-round CONGEST maximum matching algorithm that has an approximation factor of $(1-\varepsilon)$ in bipartite graphs and an approximation factor of $(2/3 - \varepsilon)$ in general graphs. Unlike the previous algorithms, the algorithm of [2] is not based on the framework of Hopcroft and Karp. Instead, the algorithm first computes an almost optimal fractional matching and it then rounds the fractional solution to an integer solution by adapting an algorithm of [21]. There also exist deterministic distributed algorithms to $(1-\varepsilon)$-approximate maximum matching in polylogarithmic time [19, 22, 24], these algorithm however require the LOCAL model. The algorithms of [2, 22, 24] directly also work for the maximum weighted matching problem. Other distributed algorithms that compute constant-factor approximations for the weighted

[1] In the LOCAL model, the algorithm can be implemented in time $O(\log(n)/\operatorname{poly}(\varepsilon))$ also for general graphs. This was independently also shown in a concurrent paper by Nieberg [42].

maximum matching problem appeared in [8, 21, 28, 36, 37, 46]. We note that none of the existing approximation algorithms can be used to solve the exact maximum matching problem in time $o(|E|)$ in the CONGEST model.

Additional related work. Our result can also be seen in the context of verification and the results established regarding decision problems in distributed settings. As an example, in [15], lower bounds for decision problems like connectivity, spanning connected subgraph, and $s - t$ cut verification are presented, and the applications of these results in deriving strong unconditional time lower bounds on the hardness of distributed approximation are shown. Another example can be seen in [33], where tight bounds are presented to verify whether a given subgraph is an MST of the network. For further study in this regard, see [4, 20]. Another context in which our result can be seen is the recent interest in the complexity of computing exact solutions to distributed optimization problems. In particular, it was recently shown that several problems that are closely related to the maximum matching problem have near-quadratic lower bounds in the CONGEST model. In [11], it is shown that computing an optimal solution to the maximum independent set and the minimum vertex cover problem both require time $\tilde{\Omega}(n^2)$ in the CONGEST model. In [6], similar $\tilde{\Omega}(n^2)$ lower bounds are proven for other problems, in particular for computing an optimal solution to the minimum dominating set problem and for computing a $(7/8 + \varepsilon)$-approximation for maximum independent set. Consequently, for maximum independent set, minimum vertex cover and minimum dominating set, the trivial $O(|E|)$-time CONGEST model algorithm is almost optimal. If our result can be extended to actually find the maximum matching in almost linear time, it would show that this is not true for the maximum matching problem.

Mathematical notation. Before giving an outline of our algorithm in Section 2, we introduce some graph-theoretic notation that we will use throughout the remainder of the paper. A walk W from node u to a node v in a graph $G = (V, E)$ is a sequence of nodes $\langle u = v_1, v_2, \ldots, v_k = v\rangle$ such that for all $j < k$, $\{v_j, v_{j+1}\} \in E$. A path P is a walk that is cycle-free, i.e., a walk where the nodes are pairwise distinct. Let $\mathbb{V}(W)$ denote the multi-set of the nodes in a walk W and let $|W|$ denote the length of the walk W, i.e., $|W| = |\mathbb{V}(W)| - 1$. For simplicity, we write $v \in W$ if $v \in \mathbb{V}(W)$. Moreover, we say an edge e is on walk W and write $e \in W$ if e is an edge between two consecutive nodes in W. For two walks $W_1 = \langle u_1, \ldots, u_s\rangle$ and $W_2 = \langle v_1, \ldots, v_t\rangle$ with $u_s = v_1$, we use $W_1 \circ W_2$ to denote the concatenation of the walks W_1 and W_2. Further, for a path $P = \langle u_1, u_2, \ldots, u_i, \ldots, u_j, \ldots, u_k\rangle$, we use $P[u_i, u_j]$ to denote the consecutive subsequence of P starting at node u_i and ending at node u_j, i.e., the subpath of P from u_i to u_j. We use parentheses instead of square brackets to exclude the starting or ending node from the subpath, e.g., $P(u_i, u_j]$, $P[u_i, u_j)$ or $P(u_i, u_j)$.

2 Outline of Our Approach

For a graph G, it is well-known that a matching M is a maximum matching of G if and only if there is no augmenting path in G w.r.t. M. By performing a broadcast/convergecast, the size of the given matching can be learnt by all nodes in the graph in time linear in D, the diameter of G. After all nodes learn the size of the given matching, the algorithm looks for an augmenting path of length at most r in phases for exponentially increasing guesses r, i.e., where r is initially set to D and it doubles from each phase to the next. The algorithm stops as soon as either $r > 4|M|$ or it detects a shortest augmenting path of length at most r. Note that the length of an augmenting path cannot be more than $2|M| + 1$. Therefore, if the

algorithm does not find a shortest augmenting path, then there is no augmenting path. The efficiency of the algorithm depends on how fast one can detect the existence of a shortest augmenting path of length at most ℓ in G for an integer ℓ. Note that although the algorithm looks for a shortest augmenting path, detecting an approximately shortest augmenting path suffices here to efficiently verify the matching. The following lemma states that this central challenging task can be done efficiently.

▶ **Lemma 2.** *Given an arbitrary graph G and a matching M of G, there is a randomized algorithm to detect whether there exists an augmenting path of length at most ℓ in $O(\ell)$ rounds of the* CONGEST *model, with high probability.* [2]

Theorem 1 now follows directly from Lemma 2 and the above algorithm outline. If ℓ is the maximum length for which we apply Lemma 2, the time for the algorithm is $O(D+\ell)$ (as we use exponentially increasing values for ℓ, the time is dominated by the time for the largest length ℓ for which we look for an augmenting path). If M is not maximum, ℓ is at most twice the length of the shortest augmenting path, if M is maximum, we have $\ell = O(|M|)$ and we thus get a time of $O(D+|M|) = O(|M|)$.

2.1 Detecting a Shortest Augmenting Path: The Challenges

Let $G = (V, E)$ be a graph, let $M \subseteq E$ be a matching of G, and let $f \in V$ be a free node (i.e., an unmatched node). Assume that we want to find a shortest augmenting path P connecting f with another free node f'. If the graph G is bipartite, such a path can be found by doing a breadth first search (BFS) along alternating paths from f. This works because in bipartite graphs, for every node v on a shortest augmenting path P connecting f with another free node f', the subpath $P[f, v]$ connecting f and v is also a shortest alternating path between f and v. In [45], Vazirani calls this property, which holds in bipartite graphs, the *BFS-honesty property*. If this property holds, to find a shortest alternating path from a free node f to a node v, it suffices to know shortest alternating paths from f to all the nodes along this path. The BFS-honesty property does not hold in general graphs. A simple example that shows this is given in Figure 1. The shortest alternating path connecting node f with u is of length 3. The shortest alternating path connecting f with v is of length 5 and it contains node u, however the subpath connecting f with u on the alternating path to v is of length 4.

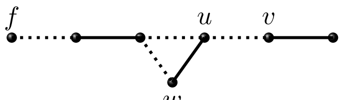


Figure 1 The BFS-honesty property does not hold in general graphs (solid lines depict edges in the matching, and dotted lines depict edges not in the matching).

To show the use of the BFS-honesty property in the distributed setting more clearly, we next sketch the algorithm of [2] for finding a shortest augmenting path in a bipartite graph. The algorithm essentially works as follows. Every free node $f \in V$ in parallel starts its own BFS exploration of G along alternating paths. The exploration of a free node f is done by propagating its ID (i.e., f) along alternating paths from f, where the ID is propagated by one more hop in each synchronous round. Whenever a node u receives the IDs of two

[2] An event is said to happen with high probability (w.h.p.) in graph G of size n if it happens with probability at least $1-(1/n)^c$ for some constant $c > 0$.

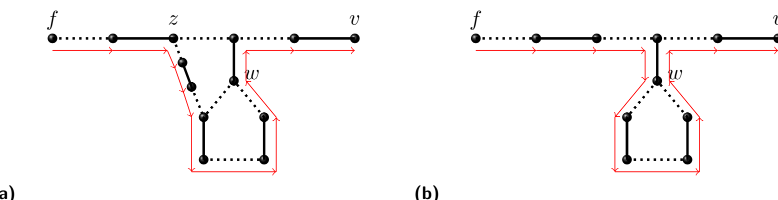


Figure 2 Main challenge: Nodes need to be able to distinguish whether the BFS exploration reaches them on an alternating path or only on alternating walks.

different free nodes f and f' in the same round, it only forwards the ID of one of them. Note that each node only forwards a single free node ID and it only forwards this ID once (in the round after it first receives it). This is sufficient if the BFS-honesty property holds. Moreover, this guarantees that IDs only traverse alternating paths and avoid traversing cycles. Assume that the shortest augmenting path in the graph is of length $\ell = 2k+1$. Let P be such a path and assume that $\{u, v\}$ is the middle edge of P. Note that this implies that the shortest alternating paths of nodes u and v are both of length k. Hence, u and v receive the ID of a free node exactly in round k and they will both forward that ID along edge $\{u, v\}$ in round $k+1$. When this happens, u and v learn about the fact that they are in the middle of an augmenting path of length $2k+1$ and that path can be constructed simply by following back the edges on which the alternating BFS traversals reached nodes u and v.

Let us now discuss some of the challenges when adapting this ID dissemination protocol to general graphs. For simplicity, assume that we are only doing the BFS exploration from a single free node f. Consider again the example in Figure 1. We have seen that the shortest alternating path from f to v passes through node u, however the subpath from f to u is not the shortest alternating path from f to u. In fact, while the shortest alternating path from f to u reaches u on an unmatched edge, in order to reach node v, we have to use the shortest one of the alternating paths from f to u that reaches u on a matched edge. This suggests that each node v should keep track of both kinds of shortest paths from node f and that v should forward f twice. A natural generalization of the protocol would thus be the following: After receiving f on a shortest alternating path ending in an unmatched edge of v, v forwards f on its matched edge and after receiving f on a shortest alternating path ending in the matched edge of v, v forwards f on its unmatched edges. One would hope that this lets each node detect both kinds of shortest alternating paths from node f. However, as Figure 2 shows, this is not necessarily true. While in the Figure 2a, when v receives f over its matched edge, the ID was indeed forwarded on a shortest alternating path from f to v. However, in Figure 2b, the exploration passes through an odd cycle and node v is only reached on an alternating walk instead of an alternating path. In the example of Figure 2b, node w should detect that the BFS traversal passed through the odd cycle and w should therefore not forward f over its matched edge. However, it is not clear how w should distinguish between the cases in Figure 2a and Figure 2b. Note that in the BFS traversal of Figure 2a, f is not only forwarded on the alternating path to w, but also through the odd cycle as in Figure 2b. In fact, the example of Figure 2 is still a relatively simple case as odd cycles can be nested, and closed odd walks can look much more complicated than just passing through a single odd cycle. Detecting whether and when to forward the ID of a free node is the main algorithmic challenge that we face.

A second challenge comes from the fact that we need to do alternating BFS explorations from all free nodes and it is not obvious how to coordinate these parallel BFS explorations while keeping the message size small. In the bipartite case, it was enough for each node v

to only participate in the BFS exploration of a single free node f and to discard all other BFS traversals that reach node v. It is not clear whether the same thing can also be done in general graphs. Luckily it turns out to still be sufficient if each node v only participates in the BFS exploration of the first free node that reaches v.

On a very high level, our approach resembles the basic underlying idea of the classic maximum matching algorithm of Edmonds [17]. As in Edmonds algorithm, we grow a forest of alternating trees from the unmatched (free) nodes. However, in order to find a shortest (or almost shortest) augmenting path, we have to grow these trees in a parallel BFS-like fashion. Note however that in a non-bipartite setting, each node of a graph might have an odd and an even-length alternating path to each free node and the shortest paths of both kinds might be of very different lengths. It is therefore not immediately clear how to grow forests in parallel such that in the end, one finds a *short* augmenting path. The following free node clustering (which we think might be of independent interest) shows that there is a relatively straightforward parallel alternating BFS construction that allows to indeed find a shortest augmenting path. The main technical challenge of our paper will be to construct this free node clustering in a distributed way. Note that unlike in Edmonds algorithm, we cannot just contract odd cycles when we find them. In fact, to be efficient, we cannot even detect odd cycles explicitly, but we have to deal with them in an implicit way.

2.2 The Free Node Clustering

We start the outline of our algorithm to detect a shortest augmenting path by describing the required outcome of the alternating BFS exploration in general graphs in more detail. We intend to perform BFS explorations starting from all the free nodes $f_1, \dots, f_\rho$ in parallel. We will show that it is sufficient for each node $v \in V$ to participate in the BFS exploration for exactly one free node f_i. This implies that at each point in time, the BFS explorations of the different free nodes $f_1, \dots, f_\rho$ induce a clustering of the nodes in V. There is a cluster for each free node f_i, and each node $v \in V$ is either contained in exactly one of the ρ clusters or it is not contained in any cluster (i.e., has not been reached by any of the explorations). We call this induced clustering the *free node clustering*. The clustering is computed in synchronous rounds and we will guarantee that it satisfies the following properties.

(C1) Consider some node $v \in V$ and some round number $r \geq 1$. If v has not joined any cluster in the first $r-1$ rounds, if v has an alternating path P of length r to a cluster center f (i.e., a free node f), and if all nodes of P except node v are in the cluster of node f after $r-1$ rounds, then v joins the cluster of f or some other cluster with the same property. That is, v joins a cluster in round r such that afterwards, it has an alternating path P' of length r to its cluster center such that P' is completely contained in the cluster that v joined. Let C be the cluster that v joins and let $U \subseteq C$ be the set of neighbors u of v such that the cluster contains an alternating path of length r from v through u to the cluster center. The set U is called the *predecessors* of v. If v joins a cluster in round r, we say that v is r-*reachable* (i.e., v's shortest alternating paths to its cluster center that are completely contained in the cluster are of length r). If v's incident edge on the shortest alternating path of v is an unmatched edge (i.e., if r is odd), we say that v is r-0-*reachable* and otherwise, we say that it is r-1-*reachable*.

(C2) Assume that v is r_v-reachable and f is the center of the cluster to which v already belongs. Let $r > r_v$ be the first round after which the cluster of v contains an alternating path P of length r connecting v with f such that if v is r_v-0-reachable, P starts with a

matched edge at node v and if v is r_v-1-reachable, P starts with an unmatched edge at node v (if such a round r exists). Then, after r rounds of the construction, v is aware of the existence of such a path. If v is r_v-0-reachable, we say it is also r-1-*reachable* and if it is r_v-1-reachable, we say that it is also r-0-*reachable*.

To put it differently, a node in a cluster is called r-0-reachable (r-1-reachable) if there is a shortest odd-length (even-length) alternating path of length r from the cluster center to the node that is completely contained in the cluster. The clustering after r rounds of the construction will be called the *r-radius free node clustering*. For the precise definition of the clustering and of the related terminology, we refer to Section 3.1. We will see that the r-radius free node clustering can be constructed in r rounds in the CONGEST model. We give an outline of the distributed construction of the clustering in the following Section 2.3. The details of the distributed construction and its analysis appear in Section 3 and the full version [1]. Before discussing the distributed construction, we next sketch how the free node clustering can be used to detect an augmenting path and why it is sufficient for the detection.

Detecting augmenting paths. After computing the r-radius free node clustering for a sufficiently large radius r, we can use it to detect the existence of an augmenting path as follows. Let u and v be two neighbors in G such that u and v are in different clusters (say for free nodes f and f'). Assume that for two integers $\ell_u, \ell_v \geq 0$ one of the following conditions hold:

1. The edge $\{u, v\}$ is in the matching, u is ℓ_u-0-reachable (in its cluster), and v is ℓ_v-0-reachable (in its cluster).
2. The edge $\{u, v\}$ is not in the matching, u is ℓ_u-1-reachable (in its cluster), and v is ℓ_v-1-reachable (in its cluster).

In both cases the matching directly implies that there exists an augmenting path of length $\ell_u + \ell_v + 1$ between the free nodes f and f'. Further, after $\max\{\ell_u, \ell_v\} + 1$ rounds, u and v are aware of the existence of this path. Note that this only gives the existence of a shortest augmenting path between f and f', but it does not allow to construct such a path. To construct the path, nodes would need to know their next node on the path towards the free nodes. However, since the subpaths of a shortest alternating path of a node to some free node are not shortest alternating paths themselves (i.e., since the BFS-honesty property does not hold), the information provided by the clustering does not suffice to compute the path.

Detectability of a shortest augmenting path. It remains to show that the free node clustering allows to find some shortest augmenting path. However, note that detecting a relatively short augmenting path suffices to efficiently verify the matching. Assume that the length of a shortest augmenting path in G w.r.t. the given matching M is $2k + 1$ for some integer $k \geq 0$. For an augmenting path $P = \langle f = v_0, \ldots, v_\ell = f' \rangle$ of length $\ell = 2k + 1$ between two free nodes f and f', we let $i \geq 0$ and $j \geq 0$ be two integers such that i is the largest integer such that all nodes in $P[v_0, v_i]$ are in the cluster of f and such that j is the largest integer such that all the nodes in $P[v_{\ell-j}, v_\ell]$ are in the cluster of f'. We define the *rank* of the augmenting path P as $i + j$. Note that the path P is detectable if and only if it has rank $2k$. We thus need to show that there exists a shortest augmenting path of rank $2k$.

To prove that there is a shortest augmenting path of rank $2k$, we assume that P is a shortest augmenting path of maximal rank and that the rank of P is less than $2k$ and we show that this leads to a contradition: this either allows to construct an augmenting path of length less than $2k + 1$ or it allows to construct an augmenting path of length $2k + 1$ of larger rank. The actual proof is somewhat technical. It consists of two steps. If we assume,

w.l.o.g., that $i \leq j$, we first show inductively that all the nodes $v_{i+1}, \ldots, v_{\max\{i+1,j\}}$ are in the cluster of f'. If $j \geq \ell - j - 1$, we have proven that $v_{\ell-j-1}$ is in the cluster of f', which is a contradiction to the choice of j. Otherwise, node $v_{\ell-j-1}$ is in a cluster $f'' \neq f'$ (it is however possible that $f'' = f$). We can now derive the desired contradiction by a careful concatenation of parts of the paths connecting f'' with $v_{\ell-j-1}$, parts of the augmenting path between f and f', and parts of a path between f' and v_{i+1} that was constructed in the earlier inductive argument. The details of the arguments appear in Section 3.2.

2.3 Distributed Construction of the Free Node Clustering

We focus on a single step (round) of the the distributed construction of the free node clustering. To that end, consider graph G and matching M, and assume that the first $r-1$ rounds of the clustering construction have been done successfully and the introduced clustering properties (C1) and (C2) of Section 2.2 hold. Therefore, for all integers $t < r$ and $\vartheta \in \{0,1\}$, every t-ϑ-reachable node correctly detects the fact that it is t-ϑ-reachable and knows its predecessors. Then, let us explain the outline of the approach towards implementing the rth step of the distributed construction of the clustering.

Let us first focus on maintaining property (C1). To satisfy (C1), every $(r-1)$-ϑ-reachable node sends its cluster ID over its adjacent matched edge if $\vartheta = 0$ and over its adjacent unmatched edges if $\vartheta = 1$. This way, for every node that receives a cluster ID over its adjacent edge, by joining the corresponding cluster, there would be an alternating path of length r from the cluster center to the node such that the path is completely contained in the cluster. This maintains property (C1) for r steps of the clustering and it can be achieved in a distributed setting as explained. However, maintaining property (C2) is the main challenge as we try to elaborate in the sequel.

Let us consider a node v in the cluster centered at some free node f after $r-1$ steps of the clustering construction such that it is r'-ϑ-reachable for some integers $r' < r$ and $\vartheta \in \{0,1\}$. Let us then assume that after the nodes have joined their corresponding clusters in the rth step, there is an alternating path of length r from f to v that has completely fallen into the cluster centered at f such that the path contains an adjacent matched edge of v if $\vartheta = 0$ and an adjacent unmatched edge of v otherwise. Therefore, v should learn about the existence of such a path to maintain property (C2). Nodes like v can be reached within the cluster from their corresponding cluster center in two ways; first through an alternating path from the cluster center to v such that the path is completely contained in the cluster and contains a matched edge of v, and second through a similar path but containing an unmatched edge of v. Let us call these nodes that can be reached via both kinds of paths *bireachable nodes*.

Let us define an *odd cycle* to be an alternating walk of odd length that is completely contained in a cluster and starts and ends at the same node. Node v that is the first and last node of an odd cycle is called the stem of the odd cycle. An odd cycle is said to be *minimal* if it has no consecutive subsequence that is an odd cycle. Note that a minimal odd cycle can still have a consecutive subsequence that is an even-length cycle. An odd cycle is moreover said to be *reachable* if either the stem is the cluster center or there is an alternating path from the cluster center to the stem of the odd cycle such that (1) it is completely contained in the cluster, (2) it is edge-disjoint from the odd cycle, and (3) it includes the matched edge of the stem of the odd cycle. You can see examples of reachable minimal odd cycles in Figure 2a, one with stem w and another with stem z. All the nodes of an odd cycle except the stem are said to be *strictly inside* the odd cycle. Then, one can show that a node is bireachable if and only if it is strictly inside a reachable minimal odd cycle. We only need this simple observation to explain the intuition behind our approach for maintaining property (C2), and

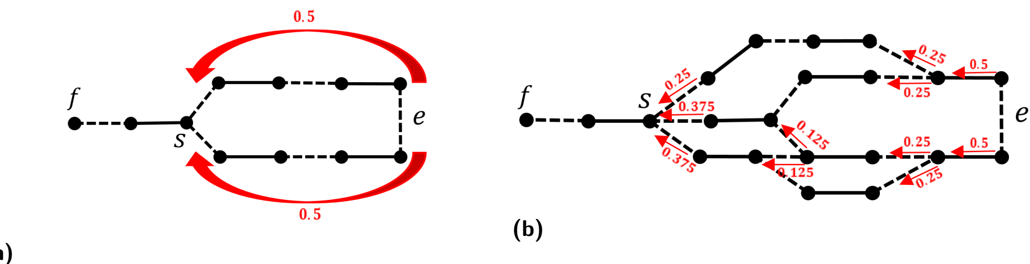


Figure 3 Flow circulation in reachable minimal odd cycles.

in the full version of the paper [1], we formally prove the correctness of the approach. As an example, node w is strictly inside the reachable minimal odd cycle with stem z in Figure 2a and hence bireachable, but w is not bireachable in Figure 2b.

To help the nodes to distinguish whether they are strictly inside a reachable minimal odd cycle or not, we define a flow circulation protocol throughout each cluster. Let us consider the very simple example of a reachable minimal odd cycle in Figure 3a. When the cluster ID is sent over the middle edge of this odd cycle (i.e., e) in both directions in the same round, we consider a flow generation of unit size over the edge and we call it the flow of e. Then, half of the generated flow is sent back towards the stem of the odd cycle on each of the two paths. When the stem receives the whole unit flow of edge e in a single round, it learns that it is the stem of an odd cycle for which the flow is generated and discards the flow (it avoids sending the flow further). Whereas all the other nodes inside the odd cycle receive a flow of value less than 1. They interpret this incomplete flow receipt as being strictly inside a reachable minimal odd cycle. Moreover, they interpret the round in which they receive an incomplete flow for the first time as the length of an existing alternating path from the cluster center. Therefore, to maintain property (C2), a node detects the length of its shortest alternating path through its matched (unmatched) edge by receiving an incomplete flow for the first time if its shortest alternating path contains its unmatched (matched) edge.

Many such odd cycles might share a common middle edge as the cycles in Figure 3b that share e as their middle edge. Then, it is enough that every node divides the value of the received flow of e and sends them backwards until the cycle's stem, i.e., node s, receives the whole unit flow of e. However, in case of having many interconnected and nested reachable minimal odd cycles that do not share a single middle edge, an edge might carry the flows of many different edges in the same round. This is a problem when implementing the idea in the CONGEST model as we cannot bound the number of flows that have to be sent concurrently over an edge. Instead of separately sending flows generated at different edges e, we therefore sum all flows that have to go over the same edge and only send aggregate values. Ideally, we would like to have the following desired differentiation; a node that receives an aggregated flow whose size is not an integer, learns that it is strictly inside at least one reachable minimal odd cycle, and a node that receives an aggregated flow whose size is an integer learns that it is the stem of at least one reachable minimal odd cycle but not strictly inside any such cycle and it discards the flow. To avoid that a set of fractional flows for different edges sums to an integer, we can use randomization. Instead of always equally splitting flow that has to be sent over several edges, we randomly split the flow. This guarantees that w.h.p., flows only sum up to an integer if they consist of all parts of all involved separate flows. Unfortunately, this is still not directly implementable in the CONGEST model because we might need to split a single flow a polynomial in n many times and $O(\log n)$ bits then are not sufficient to forward

the flow value with sufficient accuracy. In order to apply the idea in the CONGEST model, we instead use flow values from a sufficiently large (polynomial size) finite field. In [1], we show that this suffices to w.h.p. obtain the same behavior as if flows for each edge were sent separately. Aggregating flows thus allows to satisfy the congestion requirement, it however causes a number of further challenging problems, which we present and discuss next.

Let us consider the rather basic example of having only two nested reachable minimal odd cycles in Figure 4. Let C denote the odd cycle with stem s and C' denote the odd cycle with stem s'. Observe that e is the middle edge of C and e' is the middle edge of C'. The received flow of e by x must be sent to y_1 whereas the received flow of e' by x must be sent to y_2, which requires node x to treat the two flows differently. That is, node x must recognize that the received flow of e corresponds to the odd cycle containing the alternating path ending at edge $\{y_1, x\}$, and the received flow of e' corresponds to the odd cycle containing the alternating path ending at edge $\{y_2, x\}$. This cannot be achieved due to the flow aggregation enforced by the congestion restriction. Therefore, node x is not capable of correctly directing the flows along the right paths so that the flows of e only traverse the paths of cycle C and the flows of e' only traverse the paths of cycle C'.

To resolve this issue and be able to still aggregate the flows, nodes should be able to treat all flows in the same way. Therefore, since every node knows its predecessors, we would like to establish the generic regulation of always sending flows only towards all predecessors no matter what the flow is. However, by letting node x send the received flow of e' to its only predecessor y_1, the nodes in the alternating path between x and s' through y_2 do not anymore receive any flow of e'. To fix this and keep node x free of treating flows differently, we eliminate the flow generation over e' and simulate it by generating a flow over e''. That is, we shift the flow generation of cycle C' from e' to e''. Then, half of the flow of e'' is sent by y_2 to its predecessor y_4, and half of it is sent by x to its predecessor y_1.

Previously, we let the flow generation only occur over the middle edge of an odd cycle, that can be easily recognized when an edge carries the same cluster ID in opposite directions in the same round. Now by having flow generation over both middle edges like e as well as non-middle edges like e'', we need a more involved flow generation regulation. Let every node send its cluster ID in at most one round over its matched edge and in at most one round over its unmatched edges. A node sends its cluster ID over its matched edges in round r if it is $(r-1)$-0-reachable, and it sends its cluster ID over its unmatched edges in round r' if it is $(r'-1)$-1-reachable. We let a flow be generated over an edge when the two endpoints are not each other's predecessors and they both send the same cluster ID to each other, no matter if they are sent in the same round or not. Neither the endpoints of e nor those of e'' are each other's predecessors while the endpoints send f to each other over e and e''. Therefore, flow generation occur over both e and e'', where e is an example of a middle edge that the endpoints send cluster IDs in the same round, and e'' is an example of a non-middle edge for a shifted flow generation that the endpoints send cluster IDs in different rounds. Also note that since y_3 is the predecessor of y_6, a flow is not anymore generated over e' within this new regulation.

Now to see that shifting flow generation maintains the desired effects and avoids any side effects, let us compare the two cases of flow generation over e' and its simulation over e''. Each half of the flow of e' is sent towards s', one along the path between y_6 and s' through y_5 and one along the path between y_3 and s' through y_4 as depicted by arrows in Figure 4. In the simulation, each half of the flow of e'' is also sent towards s', one along the path between y_2 and s' through y_4 and one along the path between x and s' through y_1 again as depicted by arrows in Figure 4. We need the simulation to serve the purposes of the flow

Figure 4 Nested odd cycles: flow simulation of edge e' on edge e''.

generation of e'. However, there are two crucial differences that might question the desired effects of flow e' if we run the simulation instead. To explain the first difference, consider the nodes inside odd cycle C. Nodes like y_1 do not receive flows of e' but receive flows of e'' in the simulation. Moreover, nodes like y_5 receive flows of e' but not flows of e'' in the simulation. The second difference is that node s' as the stem of C' receives the whole unit flow of e' in a single round as desired to perceive the fact that it is the corresponding stem and discards the flow. However, in the simulation since e'' is not the middle edge of C' and the flows are sent along paths with different lengths, s' does not receive the whole unit flow of e'' in a single round. This avoids node s' to perceive the fact that it is the stem of an odd cycle and the flow is further sent by s'. Let us see how crucial these differences are and how we can resolve them.

Regarding the first difference, the decisive observation is that whenever a node receives a proper fraction of a flow for the first time in round r, it detects the existence of an alternating path of length r. Therefore, only the first receipt of such flow is important and must be at the right time for a node. All those nodes in C that differ in receiving the corresponding flows in the flow circulation of e' and e'' already have received a proper fraction of flow e, and hence the receipt and the time of receiving later flows are irrelevant to them.

However, the second difference is crucial and needs to be resolved. Note that the half flow of e'' is sent along the path between y_2 and s' through y_4 that is the same path traversed by the half flow of e'. Therefore, if y_2 sends the half flow of e'' to y_4 immediately in the next round of receiving the cluster ID from x, it reaches s' at exactly the same time as the half flow of e' would have reached s'. Now assume that x would also have sent the other half flow of e'' along the path between x and s' through y_5. If x would have sent the flow in the very next round of receiving the cluster ID from y_2, then the flow of e'' would also have reached s' in exactly the same round as the flow of e' would have reached s'. However, since x actually sends this flow along the path between x and s' through y_1, it reaches s' sooner. This time difference is the difference of the length of the shortest alternating path between x and f ending in the matched edge of x and such a path ending in the unmatched edge of x. This difference is known by x, and x can therefore delay sending the flow by this number of rounds and repair the unwanted side effects of the simulation. Note that x cannot send the flow in the very next round of receiving the cluster ID from y_2 since it has not yet decided at that time to send its cluster ID to y_2 and hence cannot yet recognize the flow generation over e''. Therefore, it has to anyway send the flow along a shorter path, e.g., the path through y_1.

This discussed simple example inevitably abstracts away some details. In the example, flows are only sent over alternating paths. However, if nodes always send flows to their predecessors, flows do not necessarily traverse alternating paths and the paths along which flows are sent might have consecutive unmatched edges. Then, along a path that a flow is forwarded, every node that has two adjacent unmatched edges on the path delays forwarding the flow.

3 Shortest Augmenting Path Detection

In this section, we present the algorithm to detect a shortest augmenting path in time linear in the length of the path in the CONGEST model. The organization of this section is as follows. In Section 3.1, we formally define the free node clustering that was described in Section 2.2. We define the clustering by giving a deterministic sequential algorithm that constructs the clustering in a step-by-step manner. Note that this deterministic algorithm is only for the purpose of providing a precise definition of the clustering. Then, in Section 3.2, we show that given such a clustering, at least one shortest augmenting path can be detected in a single round of the CONGEST model. In Section 3.3, we provide a distributed algorithm to construct the free node clustering. Due to lack of space, the analysis of the distributed free node clustering algorithm appears in the full version of the paper [1]. For the sake of simplicity, we first consider no restriction on the size of sent messages when we describe the algorithm.

3.1 The r-Radius Free Node Clustering

To define the r-radius free node clustering of a graph G w.r.t. a given matching M of G, we introduce a deterministic sequential r-step algorithm, which we henceforth call the FNC algorithm. The free nodes $f_1, \dots, f_\rho$ are the cluster centers. For all i, let C_i denote the cluster that is centered at free node f_i. Initially every cluster C_i only contains f_i and during the execution of the algorithm more nodes potentially join the cluster. For consistency, we assume that there exists a step 0 in which every free node joins the cluster centered at itself, i.e., initially $\forall i \in [1, \rho] : C_i = \{f_i\}$. Then, in every step $t \geq 1$, every node that has not yet joined any cluster, concurrently joins the cluster centered at f_i if and only if f_i is the minimum-ID free node from which v has an alternating path P of length t such that $\mathbb{V}(P) \setminus \{v\} \subseteq C_i$. Note that there might be some nodes in G that never join any cluster in any step of the FNC algorithm. Throughout, let $C_i(t)$ denote the set of nodes in cluster C_i after t steps of the FNC algorithm. We define $\mathcal{C}(r) := \{C_1(r), \dots, C_\rho(r)\}$ to be the r-radius free node clustering of G.

To simplify the discussions, we introduce the following definitions and a simple observation in the next lemma. In the following definitions, v is an arbitrary node in G and ϑ is an arbitrary integer in $\{0, 1\}$. We say that P is a *path of* v or v *has a path* P if P is a path starting at a free node and ending at node v.

▶ **Definition 3** (Uniform Paths). *We say that a path P is uniform at time $t \geq 0$ if $\mathbb{V}(P) \subseteq C_i(t)$ for some $i \in \{1, \dots, \rho\}$. When the time t is clear from the context, we just say that P is uniform.*

▶ **Lemma 4.** *Let P be an alternating path of length r from any free node to any node. If there is any time (possibly larger than r) at which P is uniform, then P is uniform at time r.*

Proof. Due to lack of space, the proof is presented in the full version of the paper [1]. ◀

▶ **Definition 5** (Almost Uniform Paths). *We say that a path P of v is almost uniform (at time t) if $\mathbb{V}(P) \setminus \{v\}$ is uniform (at time t). Note that every uniform path of v is also almost uniform.*

▶ **Definition 6** (ϑ-Edges). *A free (i.e., an unmatched) edge is called a 0-edge, and a matched edge is called a 1-edge.*

▶ **Definition 7** (ϑ-Paths). *An alternating path P of v is called a ϑ-path of v if P contains a ϑ-edge adjacent to v.*

▶ **Definition 8** (Predecessors). *We say u is a predecessor of v if u is the neighbor of v on a shortest uniform alternating path of v.*

▶ **Definition 9** (Reachability). *For an integer $r \geq 0$, we say that v is r-ϑ-reachable if v has a shortest uniform ϑ-path of length r. Moreover, we say that v is r-reachable if it has a shortest uniform alternating path of length r.*

For the sake of consistency, we assume that every free node is 0-0-reachable and 0-1-reachable

3.2 Detecting a Shortest Augmenting Path

In this section, we show how the existence of a shortest augmenting path of length at most ℓ can be detected in a single round of the CONGEST model if the nodes of G are provided with the ℓ-radius free node clustering with respect to a given matching M of G and if in addition the ℓ-radius free node clustering is *well-formed* in the following sense.

▶ **Definition 10** (Well-Formed Clustering). *The r-radius free node clustering $\mathcal{C}(r)$ is said to be well-formed in a distributed setting if for all $r' \leq r$, $\vartheta \in \{0, 1\}$ and i, every r'-ϑ-reachable node $v \in C_i(r)$, beyond knowing its cluster ID, knows its predecessors and the fact of being r'-ϑ-reachable.*

▶ **Definition 11** (Rank of an Augmenting Path). *For an integer ℓ, consider an arbitrary augmenting path P of length ℓ between any pair of free nodes f_s and f_t. Let us name the nodes of P as $\langle f_s = u_0, \ldots, u_\ell = f_t \rangle$ and let i and j be the largest integers such that the subpaths $P[f_s, u_i]$ and $P[u_{\ell-j}, f_t]$ are uniform. Then, we define the rank of P to be $i + j$.*

▶ **Lemma 12.** *Let all nodes of a given graph G be provided with the well-formed r-radius free node clustering with respect to a given matching M of G. If there is a shortest augmenting path of length $\ell \leq r$, then a shortest augmenting path can be detected in a single round of the* CONGEST *model.*

Proof. Due to lack of space, the proof is presented in the full version of the paper [1]. ◀

3.3 Distributed Free Node Clustering

In this section, we present a distributed deterministic algorithm whose r-round execution provides the well-formed r-radius free node clustering. This algorithm uses large messages. However, as we discussed in Section 2.3, we can use randomness to adapt this algorithm to the CONGEST model.

Distributed r-Radius Free Node Clustering: DFNC Algorithm. The algorithm is run for r rounds. Let the following variables be maintained by the nodes during the execution; $r_v^{(0)}$ and $r_v^{(1)}$ keep track of the v's reachabilities, cid_v holds the cluster ID of v, and $pred_v$ holds the set of v's predecessors. At the beginning of the execution, for every free node v, variables $r_v^{(0)}$ and $r_v^{(1)}$ are set to 0, variable cid_v is set to v, and $pred_v$ is set to $\emptyset$. Moreover, for every matched node, all these variables are initially undefined and set to $\perp$. Every node v participates in the token dissemination based on the following simple rule. For an arbitrary integer $t \geq 1$, in round t:

- If $r_v^{(0)} = t-1$, then v sends cid_v over its adjacent 1-edge (if any). Otherwise, if $r_v^{(1)} = t-1$, then v sends cid_v over all its adjacent 0-edges (if any).

Based on the above simple rule, every node sends tokens to its neighbors in at most two rounds, at most once over its 1-edge and at most once over its 0-edges. Therefore, in the first round of the execution, every free node sends its ID to all its neighbors. Let round t be the first round in which a node v receives tokens. Let $\tau_1, \ldots, \tau_j$ be the tokens that v receives from its neighbors in round t. Then, v sets cid_v to $\min_i \tau_i$, and subsequently sets $pred_v$ to the set of all its neighbors that sent cid_v to v in round t. There are two types of messages sent by the nodes throughout the execution; tokens (i.e., free node IDs) and *flow messages*. Node v sets $r_v^{(1)}$ and $r_v^{(0)}$ based on the received tokens and flow messages. Before we explain how these variables are set by v, let us first define the flow messages by explaining flow generation and circulation throughout the network.

Flow Generation: A *flow* is a key-value pair, where the key is an edge and the value is a real number in $[0, 1]$. Any flow whose key is some edge e is simply called a *flow of edge e*. A *flow message* is then defined to be a set of flows (i.e., key-value pairs) that are sent by a node to its neighbor in a round. A flow generation is an event that can only happen over an edge for which both endpoints belong to the same cluster. Let us fix an arbitrary edge $e = \{u, w\}$ where both endpoints belong to the same cluster. Then, we say that a flow is generated over edge e if and only if (1) none of u or w is the other one's predecessor, and (2) both u and w send tokens to each other. Note that we only consider at most one flow generation for every edge throughout the whole execution. Let us assume that u and w are not each other's predecessors, u sends token to w in round r_u, and w sends token to u in round r_w. Then, the flow generation over e is defined as an event in which u receives a singleton flow message $\{(e, 1/2)\}$ over e in round r_w and w receives a singleton flow message $\{(e, 1/2)\}$ over e in round r_u. It is important to note that nodes u and w might send tokens to each other in different rounds, i.e., $r_u \neq r_w$. However, they cannot perceive the flow generation over e before they make sure that e carries tokens in both directions. In particular, if $r_w < r_u$, in round r_w, node u cannot perceive the flow receipt over e since it does not yet know whether it will send a token to w. Hence, u will perceive this flow receipt of round r_w later in round r_u in which it decides to send token over e and then knows that the edge carries tokens in both directions. However, we will see that u does not need to know about the flow receipt of round r_w before round r_u.

Flow Circulation: No matter if it receives a flow over its adjacent 0-edges or its adjacent 1-edge, every node always forwards the received flow to its predecessors by equally splitting the flow value among them. When the edge over which v receives a flow and the edges connected to its predecessors are not all 0-edges (see Figure 5a and 5b), v forwards the flow immediately in the next round after receiving the flow. Otherwise (see Figure 5c), it delays forwarding the flow for $r_v^{(1)} - r_v^{(0)}$ rounds. A node furthermore avoids forwarding the whole flow of a single edge e in a single round (i.e., a flow of value 1 of e). Let us see the details of the flow circulation in the following. Let $I_v(t)$ denote the set of all the flows that a node v receives in a round t, i.e., the union of all the received flow messages by v in round t. Moreover, let $O_v(t, e)$ denote the *output buffer* of a node v for its adjacent edge e in a round t, which is initially an empty set and eventually sent as a flow message over e by v in round t. Now let us fix an arbitrary node v, where E_v is the set of v's adjacent edges that are connected to its predecessors. Node v updates its output buffers in two ways; (1) it updates them with respect to the received flows and (2) it updates them to avoid forwarding the whole unit flow of a specific edge in a single round. Regarding the former case, fix an arbitrary round t in which v receives flows. If the edges over which v receives flows and the edges in E_v are all 0-edges (Figure 5c), let $t' := t + r_v^{(1)} - r_v^{(0)} + 1$.

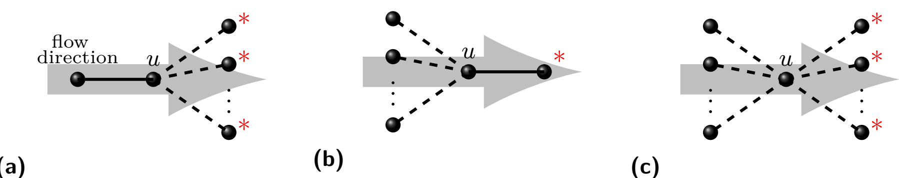


Figure 5 The 3 possibilities of flow forwarding. The predecessors of u are marked by asterisks.

Otherwise (see Figure 5a and 5b), let $t' := t+1$. Then, for every $(e, f) \in I_v(t)$, v updates its output buffers as follows:

$$\forall e' \in E_v : \quad O_v(t', e') \leftarrow O_v(t', e') \cup (e, \frac{f}{|E_v|}) \ .$$

Now regarding the latter way of output buffers update, fix an arbitrary round t'. At the beginning of round t', let $O_v(t')$ be the set of all the flows in the output buffers of v for round t', i.e., $O_v(t') := \bigcup_{e' \in E_v} O_v(t', e')$. Then, let $S_v(e, t')$ be the sum of flow values of a specific edge e that are sent by v in round t', i.e., $S_v(e, t') := \sum_{(e,f) \in O_v(t')} f$. For every edge e, if $S_v(e, t') = 1$, node v removes all flows of edge e from all its output buffers of round t'. That is, v removes all flows of e and we say that v *discards* the flow of e. After discarding all such flows, for every $e' \in E_v$, v forwards $O_v(t', e')$ over edge e' in round t' if $O_v(t', e') \neq \emptyset$.

Setting Variables $r_v^{(1)}$ and $r_v^{(0)}$ (Reachability Detection): We say that round t is an *incomplete round* for v if node v sends flow in round $t+1$. Let t be the first round in which v receives tokens or the first incomplete round for v. If t is an even integer, v assigns t to $r_v^{(1)}$, otherwise, v assigns t to $r_v^{(0)}$. The first round that a node receives a token (if any) is before its first incomplete round (if any) since it receives flows from the nodes it has already sent tokens to.

References

1. M. Ahmadi and F. Kuhn. Distributed maximum matching verification in congest, 2020. arXiv:2002.07649.
2. M. Ahmadi, F. Kuhn, and R. Oshman. Distributed approximate maximum matching in the CONGEST model. In *Proc. 32nd Symp. on Distributed Computing (DISC)*, 2018.
3. N. Alon, L. Babai, and A. Itai. A fast and simple randomized parallel algorithm for the maximal independent set problem. *Journal of Algorithms*, 7(4):567–583, 1986.
4. H. Arfaoui, P. Fraigniaud, and A. Pelc. Local decision and verification with bounded-size outputs. In *Symposium on Self-Stabilizing Systems*. Springer, 2013.
5. S. Assadi, M. Bateni, A. Bernstein, V. Mirrokni, and C. Stein. Coresets meet EDCS: Algorithms for matching and vertex cover on massive graphs. In *Proc. 30th ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 1616–1635, 2019.
6. N. Bachrach, K. Censor-Hillel, M. Dory, Y. Efron, D. Leitersdorf, and A. Paz. Hardness of distributed optimization. *CoRR*, abs/1905.10284, 2019.
7. A. Balliu, S. J. Hirvonen, D. Olivetti, M. Rabie, and J. Suomela. Lower bounds for maximal matchings and maximal independent sets. In *Proc. 60th IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 481–497, 2019.
8. R. Bar-Yehuda, K. Censor-Hillel, M. Ghaffari, and G. Schwartzman. Distributed approximation of maximum independent set and maximum matching. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 165–174, 2017.

9 L. Barenboim, M. Elkin, S. Pettie, and J. Schneider. The locality of distributed symmetry breaking. In *Proceedings of 53th Symposium on Foundations of Computer Science (FOCS)*, 2012.

10 R. Ben-Basat, K. Kawarabayashi, and G. Schwartzman. Parameterized Distributed Algorithms. In *33rd International Symposium on Distributed Computing (DISC 2019)*, Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019.

11 K. Censor-Hillel, S. Khoury, and A. Paz. Quadratic and near-quadratic lower bounds for the CONGEST model. In *Proc. 31st Symp. on Distributed Computing (DISC)*, pages 10:1–10:16, 2017.

12 A. Czumaj, J. Lacki, A. Madry, S. Mitrovic, K. Onak, and P. Sankowski. ound compression for parallel matching algorithms. In *Proc. 50th ACM Symp. on Theory of Computing (STOC)*, pages 471–484, 2018.

13 A. Czygrinow and M. Hańćkowiak. Distributed algorithm for better approximation of the maximum matching. In *9th Annual International Computing and Combinatorics Conference (COCOON)*, pages 242–251, 2003.

14 A. Czygrinow, M. Hańćkowiak, and E. Szymanska. A fast distributed algorithm for approximating the maximum matching. In *Proceedings of 12th Annual European Symposium on Algorithms (ESA)*, pages 252–263, 2004.

15 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 *Proc. 43rd Symp. on Theory of Computing (STOC)*, pages 363–372, 2011.

16 J. Edmonds. Maximum matching and a polyhedron with 0, l-vertices. *J. of Res. the Nat. Bureau of Standards*, 69 B:125–130, 1965.

17 J. Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17(3):449–467, 1965.

18 Y. Emek, S. Kutten, and R. Wattenhofer. Online matching: haste makes waste! In *Proc. 48th Symp. on Theory of Computing (STOC)*, pages 333–344, 2016.

19 G. Even, M. Medina, and D. Ron. Distributed maximum matching in bounded degree graphs. In *Proceedings of the 2015 International Conference on Distributed Computing and Networking (ICDCN)*, pages 18:1–18:10, 2015.

20 L. Feuilloley and P. Fraigniaud. Survey of distributed decision. *Bulletin of the EATCS*, 2016.

21 M. Fischer. Improved deterministic distributed matching via rounding. In *Proc. 31st Symp. on Distributed Computing (DISC)*, pages 17:1–17:15, 2017.

22 M. Fischer, M. Ghaffari, and F. Kuhn. Deterministic distributed edge-coloring via hypergraph maximal matching. In *Proceedings of 58th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 180–191, 2017.

23 T. Fischer, A. V. Goldberg, D. J. Haglin, and S. Plotkin. Approximating matchings in parallel. *Information Processing Letters*, 46(3):115–118, 1993.

24 M. Ghaffari, D. G. Harris, and F. Kuhn. On derandomizing local distributed algorithms. In *Proc. 59th Symp. on Foundations of Computer Science (FOCS)*, pages 662–673, 2018.

25 M. Ghaffari and J. Uitto. Sparsifying distributed algorithms with ramifications in massively parallel computation and centralized local computation. In *Proc. 30th ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 1636–1653, 2019.

26 M. Hańćkowiak, M. Karoński, and A. Panconesi. On the distributed complexity of computing maximal matchings. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 219–225, 1998.

27 M. Hańćkowiak, M. Karoński, and A. Panconesi. A faster distributed algorithm for computing maximal matchings deterministically. In *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 219–228, 1999.

28 J.H. Hoepman, S. Kutten, and Z. Lotker. Efficient distributed weighted matchings on trees. In *Proceedings of 13th International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, pages 115–129, 2006.

29 J. E. Hopcroft and R. M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 1973.

30 A. Israeli and A. Itai. A fast and simple randomized parallel algorithm for maximal matching. *Inf. Process. Lett.*, 22(2):77–80, 1986.

31 R. M. Karp, E. Upfal, and A. Widgerson. Constructing a perfect matching is in random NC. In *Proc. 17th ACM Symp. on Theory of Computing (STOC)*, pages 22–32, 1985.

32 R. M. Karp, U. V. Vazirani, and V. V. Vazirani. An optimal algorithm for on-line bipartite matching. In *Proc. 22nd ACM Symp. on Theory of Computing (STOC)*, pages 352–358, 1990.

33 L. Kor, A. Korman, and D. Peleg. Tight Bounds For Distributed MST Verification. In *28th International Symposium on Theoretical Aspects of Computer Science (STACS11)*, 2011.

34 F. Kuhn, T. Moscibroda, and R. Wattenhofer. Local computation: Lower and upper bounds. *J. of the ACM*, 63(2), 2016.

35 N. Linial. Locality in distributed graph algorithms. *SIAM Journal on Computing*, 21(1):193–201, 1992.

36 Z. Lotker, B. Patt-Shamir, and S. Pettie. Improved distributed approximate matching. In *Proceedings of the 20th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 129–136, 2008.

37 Z. Lotker, B. Patt-Shamir, and A. Rosén. Distributed approximate matching. *SIAM Journal on Computing*, 39(2):445–460, 2009.

38 L. Lovász and M.D. Plummer. *Matching Theory.* North-Holand, 1986.

39 M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM Journal on Computing*, 15:1036–1053, 1986.

40 Y. Mansour and S. Vardi. A local computation approximation scheme to maximum matching. *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pages 260–273, 2013.

41 A. McGregor. Finding graph matchings in data streams. In *Approximation, Randomization and Combinatorial Optimization. Algorithms and Techniques*, pages 170–181, 2005.

42 T. Nieberg. Local, distributed weighted matching on general and wireless topologies. In *Proceedings of the DIALM-POMC Joint Workshop on Foundations of Mobile Computing*, pages 87–92, 2008.

43 A. Panconesi and R. Rizzi. Some simple distributed algorithms for sparse networks. *Distributed Computing*, 14(2):97–100, 2001.

44 D. Peleg. *Distributed Computing: A Locality-Sensitive Approach.* SIAM, 2000.

45 V. V. Vazirani. A simplification of the mv matching algorithm and its proof. *CoRR*, abs/1210.4594, 2012.

46 M. Wattenhofer and R. Wattenhofer. Distributed weighted matching. In *Proc. 18th Conf. on Distributed Computing (DISC)*, pages 335–348, 2004.

47 Y. Yoshida, M. Yamamoto, and H. Ito. An improved constant-time approximation algorithm for maximum matchings. In *Proc. 41st ACM Symp. on Theory of Computing (STOC)*, pages 225–234, 2009.

Distributed Planar Reachability in Nearly Optimal Time

Merav Parter
Weizmann Institute of Science, Rehovot, Israel
merav.parter@weizmann.ac.il

Abstract

We present nearly optimal distributed algorithms for fundamental reachability problems in planar graphs. In the *single-source reachability problem* given is an n-vertex directed graph $G = (V, E)$ and a source node s, it is required to determine the subset of nodes that are reachable from s in G. We present the first distributed reachability algorithm for planar graphs that runs in nearly optimal time of $\widetilde{O}(D)$ rounds, where D is the undirected diameter of the graph. This improves the complexity of $\widetilde{O}(D^2)$ rounds implied by the recent work of [Li and Parter, STOC'19].

We also consider the more general reachability problem of identifying the *strongly connected components* (SCCs) of the graph. We present an $\widetilde{O}(D)$-round algorithm that computes for each node in the graph an identifier of its strongly connected component in G. No non-trivial upper bound for this problem (even in general graphs) has been known before.

Our algorithms are based on characterizing the structural interactions between balanced cycle separators. We show that the reachability relations between separator nodes can be compressed due to a Monge-like property of their directed shortest paths. The algorithmic results are obtained by combining this structural characterization with the recursive graph partitioning machinery of [Li and Parter, STOC'19].

2012 ACM Subject Classification Networks → Network algorithms

Keywords and phrases Distributed Graph Algorithms, Planar Graphs, Reachability

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.38

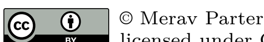
Funding Partially funded by the ISF, grant no. 713130

1 Introduction

Reachability problems in directed graphs are among the most fundamental graph problems, and as such receive quite a lot of attention in various computational settings. In this paper, we consider two canonical reachability problems for the family of planar graphs. In the *single-source reachability* problem, given a digraph $G = (V, E)$ and a source vertex s, it is required to identify the set of vertices reachable from s. In the *strong connectivity identification* problem, given a digraph $G = (V, E)$, it is required to identify the strongly connected components of G.

Throughout this paper, we consider the standard CONGEST model of distributed computing [34]. In this model, the network is abstracted as an n-vertex graph $G = (V, E)$, with one processor on each vertex. Initially, these processors only know their incident edges in the graph, and the algorithm proceeds in synchronous communication rounds over the graph $G = (V, E)$. In each round, vertices are allowed to exchange $O(\log n)$ bits with their neighbors and perform local computation. We follow the standard assumption that the communication graph is bidirectional[1].

[1] This is the common assumption in the setting of distributed graph algorithms for directed graphs, cf. [9, 5].

© Merav Parter;
licensed under Creative Commons License CC-BY
34th International Symposium on Distributed Computing (DISC 2020).
Editor: Hagit Attiya; Article No. 38; pp. 38:1–38:17

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

The single-source reachability problem along with its cousin the single-source shortest path (SSSP) problem are among the most actively studied problems in the CONGEST model [6, 33, 17, 27, 22, 7, 10, 15, 5, 25, 23]. In their seminal work, Das-Sarma et al. [6] showed a lower bound of $\Omega(D + \sqrt{n}/\log n)$ rounds for the single-source reachability problem for *general* n-vertex graphs with undirected diameter D. Nanongkai [33] gave the first non-trivial upper bound of $O(D + \sqrt{nD})$ rounds for the problem. Shortly after, Ghaffari and Udwani [17] improved the round complexity to $O(D + \sqrt{n}D^{1/4})$ rounds. This remained the state-of-the-art, and many efforts went into providing an SSSP algorithm with a matching round complexity. Very recently, Jambulapati, Liu and Sidford [25] gave an improved bound of $\widetilde{O}(\sqrt{n} + n^{1/3+o(1)} \cdot D^{2/3})$ rounds[2] for the single-source reachability problem.

In a recent work, Forster and Nanongkai [10], building on prior works of Gabow [11] and Klein and Subramanian [26], have established a strong connection between the problems of approximate SSSP in directed graphs and exact SSSP in undirected graphs. Specifically, they gave a quite general transformation that converts an approximate SSSP solution into an exact one, based on the recursive scaling approach [11, 26]. For the reduction to hold, the approximate SSSP algorithm is required to work for directed graphs, even if the initial input graph is undirected. This further strengthens the motivation for understanding reachability problems in the distributed setting.

In this work, we provide the first reachability algorithms for the family planar graphs which has been studied thoroughly in the distributed setting. The primary motivation for studying this family is two-fold. First, planar graphs are ubiquitous in real life communication. Secondly, this graph family escapes the well-known lower bound of $\Omega(D + \sqrt{n})$-rounds by Das Sarma et al. [35], giving the hope to solve many of the classical global graph problems in optimal time of $O(D)$ rounds. The area of *distributed planar algorithms* for global graph problems was introduced in a sequence of two works of Ghaffari and Haeupler [13, 14]. In their work, they introduced the useful notion of *low-congestion shortcuts* which provide a unified framework for solving global problems in the CONGEST model. Using this machinery, [14] presented improved algorithms for the minimum spanning tree and minimum-cut problems. Low-congestion shortcuts and their algorithmic applications have been further studied in [19, 20, 21, 28, 18]. An additional key algorithmic tool for planar graphs is given by the notion of *balanced separators* [30]. Ghaffari and Parter [16] presented a distributed construction of balanced cycle separators in the nearly optimal time of $\widetilde{O}(D)$ rounds.

The most relevant work to our paper is by Li and Parter [29] that presented poly(D) algorithms for distance computation in planar graphs. In their work, they introduced a very useful tool that allows one to efficiently apply a divide and conquer based approach using the balanced cycle separator of [16]. Specifically, as already observed in [16], the key limitation of applying the cycle separator algorithm in a recursive manner is rooted in the fact that the diameter of the graphs, obtained throughout the recursive process might be very large (e.g., once the cycle separator is removed). This ultimately leads to a large cycle separator, which increases the round complexity of the algorithm. To mitigate this technicality, [29] presented the notion of *bounded diameter decomposition* (BDD). Roughly speaking, the BDD is recursive decomposition of the graph in a balanced manner, in a way that guarantees that (i) the diameter of each subgraph is $\widetilde{O}(D)$ and (ii) all subgraphs at the same recursion layer are nearly edge disjoint. Using the BDD, [29] showed a poly$(D, 1/\epsilon)$-round algorithm for computing a $(1+\epsilon)$ approximation of the diameter in weighted graphs; and $\widetilde{O}(D^2)$ algorithms for computing the *exact* distance labels and SSSP trees. Very recently, Abboud, Cohen-Addad

[2] The $\widetilde{O}$ notation hides poly logarithmic factors in the number of vertices n.

and Klien [1] showed that the exact computation of the weighted diameter requires $\Omega(n)$ rounds, even if the underlying unweighted diameter is constant. This demonstrates a gap between the problem of exact SSSP and diameter computation in weighted planar graphs.

Finally, another line of research related to our work is given by the metric compression schemes for planar graphs. Abboud et al. [2] gave an efficient (centralized) compression scheme to encode the $S \times S$ distances in a unweighted undirected graphs. Using the unit-Monge property, they showed that the $S \times S$ distances can be encoded in $O(\min\{|S|^2, \sqrt{|S|n}\})$ bits. In this paper we consider *directed* graphs, and aimed at compressing reachability information in the *distributed* setting. Nevertheless, the heart of our compression technique is inspired by that of [2] and shares several technical similarities. That is, the directed paths in our work have a Monge-like property in a similar manner to the unweighted undirected shortest paths of [2]. Whereas the Monge property has been heavily utilized in many centralized algorithms for planar graphs [8, 32, 31, 24, 3, 4], our paper shows that it is also useful in the distributed setting.

1.1 Our Results

We present distributed algorithms for several fundamental graph problems in *directed* planar graphs. Starting with the single-source reachability problem, we provide an improved algorithm for computing reachability labels of size $\widetilde{O}(D)$ bits within $\widetilde{O}(D)$ rounds. Given the labels of any vertex pair u and v, one determine if there is a directed path from u to v in G and vice-versa. Given these labels, the single source reachability problem can be solved by broadcasting the $\widetilde{O}(D)$-bit label of the source to the entire network.

▶ **Theorem 1.** *[Reachability Labels and Single Source Reachability] There exists a randomized distributed algorithm that for every D-diameter planar digraph computes $\widetilde{O}(D)$-bit reachability labels within $\widetilde{O}(D)$ rounds, w.h.p.*[3] *This yields a single-source reachability algorithm with $\widetilde{O}(D)$ rounds.*

We then turn to consider the generalization of the problem to multiple sources S. A direct application of Thm. 1 gives an $\widetilde{O}(|S|D)$-round solution. We then show that when the S sources lie (not necessarily consecutively) on the boundary of a single face, the round complexity can be improved to $\widetilde{O}(D + |S|)$ rounds. This variant becomes useful for the identification of the strongly connected components of G, as will be explained next. Throughout the paper we use the notion of *virtual edges*. These are edges (u, v) that are not in G but their addition to G preservers its planarity. In the distributed setting, the endpoints of the virtual edges know their incident virtual edges and their (combinatorial) embedding in the planar graph.

▶ **Lemma 2.** *[Multi-Source Reachability] Let S be a collection of source vertices lying on a boundary of a single face of a planar graph G (possibly with one virtual edge). Then, there exists a randomized distributed algorithm that computes the $S \times V$ reachability in $\widetilde{O}(D + |S|)$ rounds w.h.p.*

Finally, we have almost all the necessary ingredients to identify the strongly connected components (SCCs) of the graph. To the best of our knowledge, there are no non-trivial algorithms for this problem, even for general graphs. One of the challenges in handling

[3] As usual, w.h.p. refers to success guarantee of $1 - 1/n^c$ for any given constant c, where n is the number of vertices.

this task is the following. When applying a recursive graph partitioning to the input graph, the SCCs of G are not necessarily extensions of the SCCs of the descendent subgraphs in this recursion. That is, the directed path P_1, P_2 from u to v and from v to u can become intact in different levels of the recursive partitioning. For that purpose, we designed a more delicate recursive scheme based on the multi-source reachability algorithm of Lemma 2. Our algorithm has a nearly optimal round complexity, which matches also the complexity of connectivity identification in undirected graphs by [14].

▶ **Lemma 3.** *[Strong Connectivity Decomposition] There exists a randomized algorithm that computes the strongly connected components (SCCs) of a planar graph G within $\widetilde{O}(D)$ rounds, w.h.p.*

Weighted Digraphs. We also consider reachability problems in *weighted* digraphs such as computing the directed SSSP tree. While our compression scheme cannot be extended to incorporate distances, we observe that the SSSP algorithm of [29] designated for undirected graphs, can in fact be easily adapted to the digraphs as well. We also show an algorithm for computing the divide minimum-weight directed cycle within $\widetilde{O}(D^2)$ rounds. We have:

▶ **Lemma 4** (Directed MSSP and Minimum Directed Girth)**.** *Given a weighted planar digraph $G = (V, E, w)$, there are randomized algorithms that w.h.p. compute (1) an exact MSSP w.r.t a subset of sources $S \subseteq V$ within $\widetilde{O}(D^2 + D \cdot |S|)$ rounds, and (2) a minimum weighted directed cycle within $\widetilde{O}(D^2)$ rounds.*

The most intriguing open question left by our work concerns the computation of SSSP in $o(D^2)$ rounds. We hope that our nearly optimal solution for the single-source reachability problem will serve the first step in that direction.

1.2 Technical Overview

Our algorithms are based on several existing tools by [16, 29] as well as several new tools, on which we elaborate more.

1.2.1 Balanced Separators and Bounded Tree Decompositions [16, 29]

A balanced separator of a planar graph G is a subset of vertices $S \subseteq V(G)$ whose removal breaks G into components of size at most $2/3n$. The celebrated result of Lipton and Tarjan [30] demonstrates the existence of a balanced cycle separator of size $O(D)$ for every planar graph with undirected diameter D. More generally, their proof shows that given a spanning tree T in G, there exist two tree paths in G plus one additional edge[4] (possibly not in G) that form a balanced *cycle* separator C for G. Ghaffari and Parter [16] gave an $\widetilde{O}(D)$-round randomized algorithm for computing these separators for biconnected planar graphs. This was later generalized for any planar graphs by [29]. In what follows, we denote the cycle separator of G by $\text{sep}(G)$.

Li and Parter [29] introduced the notion of *bounded-diameter decomposition* (BDD) of a D-diameter graph G. Viewed on the high-level, the BDD algorithm recursively applies the separator algorithm of [16] to decompose the graph into smaller and "almost" edge-disjoint subgraphs with bounded diameters. This recursive decomposition is represented by a tree $\mathcal{T}$, where each vertex of this tree, denoted as a *bag*, corresponds to subgraphs G' of G. We refer

[4] Adding that extra edge to G preserves its planarity. We refer to that edge as a *virtual edge*.

to the bags by their corresponding subgraphs. The root bag corresponds to G and the leaf bags correspond to subgraphs of G of size $O(D \log n)$. The child bags of each bag $G' \subseteq G$ are defined based on the cycle separator $\mathrm{sep}(G')$ of G'. In an ideal scenario, the separator decomposes G' into two child bags G'_{in} and G'_{out} that are embedded in the interior and the exterior of the cycle $\mathrm{sep}(G')$, respectively. The actual algorithm is more involved as it needs to satisfy several constraints to guarantee the efficiency of the distributed computation inside these bags. Due to these complications, the BDD algorithm might define several child bags $G'_1, \ldots, G'_\ell$ rather than just two.

The bounded diameter decomposition has several useful properties. The tree $\mathcal{T}$ has a logarithmic depth, as the child bags are defined based on *balanced* separators. The leaf bags contain $O(D \log n)$ vertices. In addition, in every level i of the tree $\mathcal{T}$, all the bags of that level are nearly edge-disjoint which allows one to work on all subgraphs of the same level in *parallel*. An important property of the decomposition is its *diameter preserving*: the diameter of each bag G' is bounded by $O(D \log n)$. Consequently, the separator size is bounded by $|\mathrm{sep}(G')| = O(D \log n)$ for every bag G'. This property is crucial for the efficiency of divide-and-conquer based computations in G. Another useful property of the BDD algorithm is the following. For every bag G', each separator vertex $u \in \mathrm{sep}(G')$ appears on at most three child bags of G'. Every other vertex belongs to a *unique* child bag. Keeping these properties in mind, we are now ready to highlight the key algorithmic ideas in our constructions.

1.2.2 New Approach for Reachability Labels via Reachability Preservers

At the heart of the single-source reachability algorithm lies an algorithm that computes short reachability labels for all the vertices in the graph. In the centralized setting, Thorup [36] gave an ingenious (centralized) technique to compute reachability labels of $\widetilde{O}(1)$ bits. These labels are designed based on applying various reachability computations on a modified graph $\widehat{G}$ obtained from G. Since in our setting we use the labels for the purpose of computing reachability, we will be using instead the labeling scheme by Gavoille et al. [12]. Although the latter labels have $\widetilde{O}(D)$ bits, their distributed computation is more natural, and as we will see can be done in $\widetilde{O}(D)$ rounds.

High-level description the distributed label computation. Our algorithm applies two recursive phases. The first phase of the algorithm computes in a bottom-up manner a succinct *reachability preserver* for the separator vertices. The second phase computes the reachability labels by applying a top-down recursion. Interestingly, the second phase requires no communication, and can be applied locally at each vertex based on their local knowledge on the BDD, and the reachability preservers constructed in the first phase.

For our recursive framework to hold, we introduce the notion of *extended separator set* that contains the separator vertices of G', as well as the separator vertices of all the *ancestor bags* of G' in the BDD tree. That is, letting $anc(G')$ be these ancestor bags of G', then the extended separator set, denoted by, $\mathrm{sep}^+(G')$ is the union of $\mathrm{sep}(G'')$ for every $G'' \in anc(G')$ and the vertices of $\mathrm{sep}(G')$. The important observation is that since the depth of the BDD tree is logarithmic, the extended separator set of each bag G' is bounded by $\widetilde{O}(D)$. The main technical part of the algorithm is in the computation of the reachability preservers with respect to the extended separator sets. For a given sub-graph $G' \subseteq G$ and a subset of vertices S', a reachability preserver $H(S', G')$ is a graph whose vertex set is S' and for every $s, s' \in S'$, the edge (s, s') is in $H(S', G')$ iff there exists a directed path from s to s' in G'. Our goal is to compute for every bag G', a reachability preserver $H(G') = H(\mathrm{sep}^+(G'), G')$

that captures the reachability relations between the vertices of $\text{sep}^+(G')$ in the graph G'. In the output format, it is required for every vertex u to learn the preserver $H(\text{sep}^+(G'), G')$ for every bag G' that contains u.

The preservers of the bags G' are built in a bottom-up manner, and their efficient computation is based on being able to locally compress the reachability information of the $\text{sep}^+(G')$ vertices. The leaf bags of the BDD contain just $O(D \log n)$ vertices and thus, every vertex can collect its leaf bag information and locally compute the preserver $H(G')$. Then, in every independent step of the recursion we are given a bag G' with child bags $G'_1, \ldots, G'_\ell$. By the induction, the vertices of G'_j have already computed the preservers $H(G'_j)$ for every j. It is then required to compute $H(G')$ within $\widetilde{O}(D)$ rounds. For every child bag G'_j, let $\widehat{H}(G'_j)$ be the preserver $H(G'_j)$ induced on the vertices of $\text{sep}^+(G') \cap V(G'_j)$. We then show that the preserver $H(G')$ can be obtained by computing the $\text{sep}^+(G') \times \text{sep}^+(G')$ reachability in the union of the graphs $\bigcup_j \widehat{H}(G'_j)$. Thus, the computation of the preserver $H(G')$ boils down into the following task for every child G'_j:

Every vertex in $\text{sep}^+(G') \cap V(G'_j)$ is required to send its incoming and outgoing neighbors[5] in the graph $\widehat{H}(G'_j)$ to all vertices in G'.

Since each vertex in $\text{sep}^+(G') \cap V(G'_j)$ might have $\Omega(D)$ neighbors in $\widehat{H}(G'_j)$, sending these neighbors, explicitly, leads to a total of $|\text{sep}^+(G') \cap V(G'_j)| \cdot \Omega(D) = \Omega(D^2)$ bits of information. Our key observation is that the information on the incoming and outgoing neighbors in the graph $\widehat{H}(G'_j)$ can be compressed into just $\widetilde{O}(1)$ bits! Using this compression scheme, the entire information on the edges of $\widehat{H}(G'_j)$, for every child bag G'_j, can be encoded in $\widetilde{O}(D)$ bits, in total. We also show that the encoding has a nice structure that allows all vertices in G' to locally decode it, reconstruct the graphs $\widehat{H}(G'_j)$, and consequently compute the desired preserver $H(G')$.

Once the labels are computed, the single-source reachability is computed within extra $\widetilde{O}(D)$ rounds, by sending the label of s to all the vertices.

Strongly Connected Components (SCCs) Identification. The most challenging reachability problem studied in this paper concerns the identification of the strongly connected components in G. The main obstacle for identifying these components in a divide-and-conquer based approach is rooted in the fact that the SCC of G are not necessarily extensions of the SCC of its child components G'. That is, it might be the case that $u, v \in G'$ do not belong to the same SCC in G', although they belong to the same SCC in G. Our approach for solving this problem in the nearly optimal time of $\widetilde{O}(D)$ rounds is based on the following steps. First, for every bag G' the vertices compute the reachability preservers $H(G')$ that describe the reachability in G' between all pairs of vertices in $\text{sep}^+(G')$. Next, we use the multi-source reachability algorithm to compute the $\{u\} \times \text{sep}(G')$ reachability for every vertex u and every bag G' that computes u. The separator vertices $\text{sep}(G')$ play the role of the multiple sources S. We will use here the fact that the $\text{sep}(G')$ vertices lie on a face (in fact, an almost-face) in both[6] G'_{in} and G'_{out}. This allows us to safely apply the multi-source algorithm w.r.t the vertices of $\text{sep}(G')$ in both G'_{in} and G'_{out}. The output of these two computations can be combined to obtain the reachability from u to the $\text{sep}^+(G')$ vertices in G'. The algorithm

[5] For a directed graph H and a vertex u, the incoming (outgoing) neighbors of u are all vertices v such that $(v, u) \in H$ (resp., $(u, v) \in H$).

[6] Recall that G'_{in} and G'_{out} are the subgraphs that embedded in the interior (reps., exterior) of the cycle separator $\text{sep}(G')$.

then applies a top-down recursive procedure whose goal is to let each vertex $u \in G'$ to learn the reachability relations in G, rather than in G'. That is, at the end of this procedure, each vertex $u \in G'$ learns (i) a reachability preserver for the $\mathrm{sep}^+(G') \times \mathrm{sep}^+(G')$ reachability in G; and (ii) its reachability in G to the $\mathrm{sep}(G')$ vertices.

1.3 Preliminaries and Tools

Graph Notation. For a graph $G = (V, E)$ and a subset of vertices $X \subseteq V$, let $G[X]$ be the induced subgraph on X. Throughout, we assume that the communication graph $G = (V, E)$ is connected (in the undirected sense) and has (undirected) diameter D. Otherwise, our algorithms can be applied on each (undirected) connected component of G. For a directed graph $G = (V, E)$, a subgraph $G' \subseteq G$ and $u, v \in V$, we say that $u \preceq_{G'} v$ if there is a directed path from u to v in G'. We denote by $u \rightsquigarrow v$ a directed path from u to v. The set of vertices that have a directed path to a vertex u will be denoted by the incoming vertices to u. The outgoing vertices of u are defined in an analogous manner. Let $N_{in}(u, G), N_{out}(u, G)$ be the set of incoming (resp., outgoing) neighbors of u in G. When G is clear form the context, we may simply write $N_{in}(u)$ and $N_{out}(u)$. Throughout, we use the notion of combinatorial embedding where each vertex knows the clockwise orientation of its neighbors. This embedding can be computed in $\widetilde{O}(D)$ rounds [13].

▶ **Definition 5** (Reachability Preservers). *Given a directed graph $G = (V, E)$, a subgraph G' and $S \subseteq V$, the* reachability preserver *for S in the subgraph G', denoted by $H(S, G')$, is a graph with a vertex set S that captures the $S \times S$ reachability in G'. That is, $V(H(S, G')) = S$ and for every $u, v \in S \cap G'$, $(u, v) \in H(S, G')$ iff $u \preceq_{G'} v$.*

▶ **Definition 6** (Almost-Faces). *For a planar graph $G = (V, E)$, an* almost-face *is a subset of edges E' in G such that there exists a planar graph $G' = (V, E \cup E'')$ where $E' \cup E''$ is a face in G'. That is, a set of edges E' is almost-face if one can add edges to the planar graph between the endpoints of E' to make it a face.*

We will use the fact that an almost-face has either an empty interior or exterior in the planar embedding. Our reachability algorithms make an extensive use of several distributed tools designed for *undirected* planar graphs. Interestingly, despite the fact that our goal is to compute reachability information, we will still be using useful procedures for undirected graphs such as the computation of the cycle separator, and the bounded diameter decomposition of the (underlying undirected) input graph.

Distributed Cycle Separators. For a graph $G = (V, E)$, a subset of vertices $S \subseteq V$ is a *balanced separator* if the removal of S breaks G into connected components that are constant factor smaller than the number of vertices in G. For a graph G and a spanning tree $T \subseteq G$, a *balanced cycle separator* S is a cycle formed by two tree-paths $\pi(x, y)$ and $\pi(y, z)$ (where possibly $z = y$) plus an additional edge (x, z) which is not necessarily in G (which we call a virtual edge). This cycle defines two regions in G, the region inside the cycle and the region outside, where the number of vertices in both these regions is bounded by $2n/3$. For biconnected graphs, [16] gave a randomized algorithm for computing a balanced cycle separator in $\widetilde{O}(D)$ rounds. Recently, [29] extended it to general planar graphs (i.e., 1-connected). We therefore have:

▶ **Lemma 7.** *[16, 29] Given a D-diameter planar graph G and a spanning tree $T \subseteq G$, there exists a randomized algorithm that is $\widetilde{O}(D)$ rounds w.h.p. computes a balanced cycle separator that consists of two tree paths of T plus one additional edge (which might be not in G).*

Bounded Diameter Decomposition. For an undirected graph G of diameter D, *a bounded diameter decomposition* is a recursive balanced decomposition of the vertices into subgraphs of bounded diameter. The decomposition is represented by a tree $\mathcal{T}$, whose vertices, denoted as *bags*[7], correspond to subgraphs in G. This decomposition should satisfy two crucial properties. First, the diameter of each bag is bounded by $O(D \log n)$ which enables the computation of the $O(D \log n)$-size cycle separator in a recursive manner. The second property is that each edge e belongs to at most two subgraphs in each recursion level. This allows one to work on all subgraphs of the same level simultaneously. Each bag G' in the decomposition has a topological closure $\overline{\mathcal{O}(G')}$ that might contain only vertices that appear on the separator of the ancestor bags of G' in the BDD tree $\mathcal{T}$. An additional useful property of the BDD that will be heavily exploited by our algorithms is that each vertex of G' appears on at most three child bags of G'.

We will use the following notation for a BDD tree $\mathcal{T}$. Let $anc(G')$ be the set of ancestor bags of G' on $\mathcal{T}$, and let $\text{sep}(G')$ be the cycle separator of G' obtained by the algorithm of Lemma 7. We slightly override notation by sometimes treating $\text{sep}(G')$ as the the set of cycle edges, and sometime as the set of separator vertices. Note that one of the edges of the cycle separator might not be in G. We refer to edges not in G by *virtual edges*. These virtual edges can be safely added to the graph without breaking planarity, and are used mainly for analysis purposes. The cycle separator $\text{sep}(G')$ defines two regions interior and exterior to the cycle. In the BDD tree, G' has exactly one *internal* child bag G'_{in} which is embedded in the interior of $\text{sep}(G')$, however G' might have several *external* child bags (i.e., the lie in the exterior of $\text{sep}(G')$). Handling these many *external* child bags leads to several challenges in our arguments. In the full paper we provide a formal definition of the BDD.

▶ **Theorem 8** (Bounded diameter decomposition for planar graphs, [29]). *Let $G = (V, E)$ be an unweighted planar graph with diameter D. There is a randomized distributed algorithm that w.h.p. computes the recursive partitioning of G represented by a tree $\mathcal{T}$ of height $O(\log n)$ within $\widetilde{O}(D)$ rounds. In particular, every bag $X \in V_{\mathcal{T}}$ has a unique ID and every vertex knows the IDs of all the bags that contain it.*

2 Nearly Optimal Reachability in $\widetilde{O}(D)$ Rounds

[29] presented an $\widetilde{O}(D^2)$ round algorithm for computing (exact) distance labels in undirected graphs. Just like our reachability labels, their labels are also based on the labeling scheme of Gavoille et al. [12]. In the algorithm of [29] the labels are computed in a recursive manner, where in each step of the recursion the distance labels of the separator vertices $\text{sep}(G')$ are broadcast over G' for every bag G' in the BDD. Since every label has $\widetilde{O}(D)$ bits and $|\text{sep}(G')| = \widetilde{O}(D)$, this led to a round complexity of $\widetilde{O}(D^2)$ rounds. The key contribution in this section is in providing an algorithm for computing the reachability labels without explicitly sending the labels of the separator vertices. In our algorithm, the separator vertices compress their reachability information into $\widetilde{O}(1)$ bits which will allow the vertices to reconstruct the label information.

For a given bag G' in the BDD tree $\mathcal{T}$, let $anc(G')$ be the ancestor bags of G' in $\mathcal{T}$. Define the *extended separator* vertices of G' by

$$\text{sep}^+(G') = \text{sep}(G') \cup \bigcup_{G'' \in anc(G')} \text{sep}(G'') \ .$$

[7] We note that the BDD construction has *no* relation to the tree decomposition of bounded treewidth graphs. The term bags is used in [29] to denote the vertices of the BDD.

Observe that $|\text{sep}^+(G')| = \widetilde{O}(D)$ since the height of the BDD is logarithmic and the separator of each bag is of size $O(D \log n)$. The key computational step is the computation of the reachability preservers $H(\text{sep}^+(G'), G')$ for every bag G' in the BDD tree (see Def. 5). Using this information, the vertices will be able to compute the reachability labels with no further communication. The structure of this section is as follows. In Sec. 2.1, we provide the basic characterization for compressing the reachability information between separator vertices. Then in Sec. 2.2 we describe the algorithm for computing the labels. The single-source reachability algorithm follows immediately given these labels.

Remark. Note that in the $\text{sep}^+(G')$ definition, we do not restrict the separator vertices $\text{sep}(G'')$ for $G'' \in anc(G')$ to be included in $V(G')$. However, by the definition of $H(\text{sep}^+(G'), G')$, only vertices of $\text{sep}^+(G') \cap V(G')$ are included. We still choose to define it in this way for the purpose of the future sections, in which it will be important to take into account the $\text{sep}^+(G')$ vertices that are not in $V(G')$.

2.1 Distributed Compression of Reachability Information

The efficient computation of the reachability preservers are based on several structural claims that allow the vertices in the extended separator sets to compress their reachability information. The lemmas provided in this section are built upon several properties of the BDD algorithm, and in particular to the way in which the child bags of a given bag G' are defined. While the proof arguments contain the necessary details, it will be recommended for the reader to go through the short description of the BDD algorithm to be provided in the full paper and in [29]. The arguments in this subsections have two parts. In the first part we show that the extended separator set $\text{sep}^+(G')$ lie on a collection of $O(\log n)$ almost-faces in G'. In the second part we show that the reachability information of vertices lying on two faces can be compressed efficiently. Putting these two parts together imply that the compression of the reachability information of the $\text{sep}^+(G') \times \text{sep}^+(G')$ pairs. The next lemma shows that the separator vertices of all the ancestor bags of a bag G' lie on $O(\log n)$ almost-faces in G'. The main challenge arises in the case where the BDD algorithm defines several external child bags throughout the decomposition (i.e., bags that are embedded in the exterior of the cycle separator). Recall that by virtual edge we refer to edges not in G but whose addition to G preserve its planarity. In our algorithms, the vertices know their incident virtual edges and their combinatorial embedding.

▶ **Lemma 9.** *The separator vertices of $anc(G')$ lie on a collection $O(\log n)$ almost-faces in G'. On each such face, the separator vertices lie* consecutively. *In addition each such almost-face has $O(\log n)$ virtual edges (i.e., that are not in G).*

Proof. For every level-j bag G', we will prove by induction on $i \in \{1, \ldots, j\}$ that the separator vertices $S_{[j-1,j-i]}$ of all its ancestor bags of G' in levels $j-1, \ldots, j-i$ lie consecutively on at most i almost-faces of G'.

Base case: For every bag G_1 with a parent bag G_2, we show that the vertices of $\text{sep}(G_2) \cap V(G_1)$ lie consecutively on a single almost-face of G_1 that has at most one virtual edge. Recall that the cycle separator $\text{sep}(G_2)$ is defined by two shortest paths and one additional edge e', possibly not in G. For the sake of showing that the separator vertices lie on an almost-face, we assume w.l.o.g. that $\text{sep}(G_2)$ forms a cycle in G. We now distinguish between two cases. The first case is where $\text{sep}(G_2)$ intersects the topological closure $\overline{\mathcal{O}(G_2)}$ in at most a single segment. In this case, G_2 has exactly two child bags $G_{2,in}$ and $G_{2,out}$ that lie in the interior (resp., exterior) region defined by $\text{sep}(G_2)$. We then have that the vertices of $\text{sep}(G_2)$ define the topological closure of $G_{2,in}$ and an internal face in $G_{2,out}$, so the claim holds.

Next, consider the case where $\text{sep}(G_2)$ intersects with the topological closure $\overline{\mathcal{O}(G_2)}$ in at least two non-consecutive segment. In this case, the BDD algorithm defines several bags in the exterior of the cycle $\text{sep}(G_2)$, one bag per connected region of $\mathcal{O}(G_2) \setminus \overline{\mathcal{O}(G_{2,in})}$. The vertices of $\text{sep}(G_2)$ will appear on the topological closure (and thus on the outer face) of these bags in a consecutive manner. To see this, consider a clockwise walk along the cycle $\text{sep}(G_2)$ and observe that any pair of non-continuous intersection points with the topological closure $\overline{\mathcal{O}(G_2)}$ defines a bag that is connected in $(G_2 \cup \text{sep}(G_2)) \setminus \mathcal{O}(G_{2,in})$. For these bags, it is easy to see that the vertices of $\text{sep}(G_2)$ lie consecutively on their topological closure. Finally, since the cycle $\text{sep}(G_2)$ contains at most one virtual edge, there exists at most one bag that might not be connected in $G_2 \setminus \mathcal{O}(G_{2,in})$ (although connected in $(G_2 \cup \text{sep}(G_2)) \setminus \mathcal{O}(G_{2,in})$ due to the virtual edge on $\text{sep}(G_2)$). The BDD algorithm splits this bag into *two* connected components. This breaks $\text{sep}(G_2)$ into two segments each will be on the topological closure of their bags. The induction base holds.

Inductive Step: Assume that the claim holds for each level-j bag G' w.r.t all the separator vertices of its ancestor bags in levels $j-1, \ldots, j-i$. We will prove the claim for all ancestor bags in levels $j-1, \ldots, j-i-1$. Consider a directed path $G_{j-i-1}, \ldots, G_j$ of length i in the BDD tree, that starts at a level-$(j-i-1)$ bag G_{j-i-1} and ends at a level j bag G_j.

By the induction assumption, we assume that the vertices of $\bigcup_{k=j-i-1}^{j-2} \text{sep}(G_k)$ lie consecutively on at most $i-1$ *almost-faces* $f_1, \ldots, f_{i-1}$ in G_{j-1}. The cycle separator $\text{sep}(G_{j-1})$ might have several types of interaction with each face $f_{j'}$ (e.g., containment, with or without intersection, etc.). See Fig. 1 for an illustration of the below mentioned cases. The simpler case is where $f_{j'}$ is embedded in the interior of the cycle $\text{sep}(G_{j-1})$.

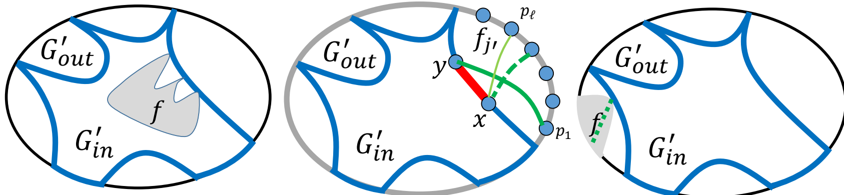


Figure 1 (1) The face f is strictly embedded in the interior of the cycle $\text{sep}(G')$. In case where vertices of f appear also in the external bags of G', it must be that these vertices are contained in $\text{sep}(G')$. (2) $f = \overline{\mathcal{O}(G')}$, that is, all vertices of f are lying on the topological closure. Note that as f is a topological face, it holds that if all vertices of f are on the topological closure then necessarily $f = \overline{\mathcal{O}(G')}$. In the case where one of the regions is not connected in G', the segment of f is split into two consecutive segments, each appearing on the topological closure one child bag. This property follows by a simple path-crossing argument. Let (x, y) be the virtual edge on $\text{sep}(G')$, in this case we will have one bag containing x and the other bag containing y. For example, if p_1 is connected in G' to y, it must also hold that all other vertices $p_{i\geq 1}$ are connected to y. This is because any path from p_i to x would intersect the path connecting p_1 and y. (3) In the last case, f is fully contained in one of the regions enclosed by the topological closure and the cycle separator. The case where the vertices embedded in this region are not connected in G' is further illustrated in Fig. 2.

The induction step then holds for $G_{j-1,in}$ w.r.t the separator vertices on $f_{j'}$. Letting $G_{j-1,out} = G_{j-1} \setminus \mathcal{O}(G_{j-1,in})$, then since $f_{j'}$ is embedded in the interior of the cycle defined by $\text{sep}(G_{j-1})$, we get that $f_{j'} \cap V(G_{j-1,out}) \subset \text{sep}(G_{j-1})$. By the induction assumption, the vertices of $\text{sep}(G_{j-1})$ lie consecutively on the external bags of G_{j-1}. We therefore have that the vertices of $f_{j'} \cup \text{sep}(G_{j-1})$ lie consecutively in the external bags of G_{j-1}. From now on, assume that that $f_{j'}$ is embedded in the exterior of the cycle $\text{sep}(G_{j-1})$.

Case 1: $\text{sep}(G_{j-1})$ and $\overline{\mathcal{O}(G_{j-1})}$ have at most one mutual segment. In this case G_{j-1} has exactly two child bags in the BDD, namely, $G_{j-1,in}$ and $G_{j-1,out}$. Since $\text{sep}(G_{j-1})$ and $f_{j'}$ are cycles, it holds that $f_{j'}$ is either fully contained in $G_{j-1,in}$ or $G_{j-1,out}$. In the case where $f_{j'} \subset G_{j-1,out}$, we have that $f_{j'} \cap G_{j-1,in} \subset \text{sep}(G_{j-1})$ and since $\text{sep}(G_{j-1})$ forms an almost-face in $G_{j-1,in}$ it holds that $(f_{j'} \cup \text{sep}(G_{j-1})) \cap G_{j-1,in}$ forms an almost-face in both $G_{j-1,in}$. The argument works in a symmetric manner in case where $f_{j'} \subset G_{j-1,in}$.

Case 2: $\text{sep}(G_{j-1})$ and $\overline{\mathcal{O}(G_{j-1})}$ have more than two mutual segments. This is the most involved case in the BDD algorithm where an external child bag is defined for each region enclosed by $\text{sep}(G_{j-1})$ and the topological closure $\overline{\mathcal{O}(G_{j-1})}$. There are two sub-cases.

Case 2.1: all the vertices of $f_{j'}$ are embedded on $\overline{\mathcal{O}(G_{j-1})}$. Note that since $f_{j'}$ is an almost-face, it holds that in this case $f_{j'} = \overline{\mathcal{O}(G_{j-1})}$. For example, this case always occur for the interior child in the BDD algorithm whose topological closure is defined by the separator of its parent bag. In this case, the boundary might be split in a consecutive manner to several child bags. See middle figure of Fig. 1. There might be at most one child bag that would be disconnected into two components. We claim that the segment of $f_{j'}$ will be decomposed into two consecutive segments, one per component. To see this, let (x, y) be the virtual edge on $\text{sep}(G_{j-1})$, and let $p_1, \ldots, p_k$ be the vertices on $f_{j'}$ lying on the topological closure from left to right, also in this ordering x appears to the left of y. Observe that if p_1 is in the component of y then all other vertices p_i for $i \geq 2$ must be in the component of y, as otherwise their path to x must intersect with the path connecting y and p_1. See Fig. 1(2). Let ℓ be the maximum index such that p_ℓ is in the component of x. By the same argument (of crossing paths) it holds that all vertices p_i for $i \in \{1, \ldots, \ell\}$ must be in the component of x as well. Thus the segment of $f_{j'}$ is cut into two consecutive segments appearing on the topological boundary of two child bags.

Case 2.2: $f_{j'}$ has at least one vertex that is not on $\overline{\mathcal{O}(G_{j-1})}$. In this case, as $f_{j'}$ is an almost-face, $f_{j'}$ is fully embedded in one of the regions enclosed by the topological boundary $\overline{\mathcal{O}(G_{j-1})}$ and the cycle separator $\text{sep}(G_{j-1})$. See Fig. 1(1,3) and Fig. 2 for an illustration. Since $\text{sep}(G_{j-1})$ contains at most one virtual edge, there exists at most one region, such that the set of vertices embedded in this region are not connected in G_{j-1} (although connected in $G_{j-1} \cup \text{sep}(G_{j-1})$ due to the virtual edge). The BDD algorithm splits this region into two regions such that all vertices embedded in those regions are now connected in G. Specifically, letting (x, y) be the virtual edge on $\text{sep}(G_{j-1})$, then one region is connected in G to x and other is connected to y. We next claim that by the Monge-like property, the separator vertices that by induction assumption lie consecutively the face $f_{j'}$ are divided into at most two consecutive segments. Note that if $f_{j'}$ contains at most one virtual edge, then all vertices in $f_{j'}$ are connected in G_{j-1}, and thus fully contained in one of the external child bags. So it remains to consider the case where $f_{j'}$ contains at least two virtual edges. Let $\sigma_1, \ldots, \sigma_\ell$ be the segments of $f_{j'}$ ordered in a clockwise manner obtained by removing the virtual edges on that face. By the Monge property it then holds that for any pair of non-neighboring segments σ_j and σ_{j+k} that are connected to x, it must hold that all the vertices on the intermediate segments $\sigma_{j+1}, \ldots, \sigma_{j+k-1}$ are connected to x as well. We then have that the vertices of f'_j can be divided into two segments $[x_1, x_2]$ and $[y_1, y_2]$, where all vertices on the segment $[x_1, x_2]$ (resp., $[y_1, y_2]$) are connected to x (resp., y) and $(x_1, y_1), (x_2, y_2)$ are virtual edge on f'_j. We can then add a virtual edge (x_1, x_2) and (y_1, y_2) and omit the virtual edges $(x_1, y_1), (x_2, y_2)$. This defines two new

almost-faces $f_{j'_1}$ and $f_{j'_2}$ that cover all the vertices on $f_{j'}$, each is fully contained in a unique external child bag of G_{j-1}. (Note that one of the virtual edges $(x_1, y_1), (x_2, y_2)$ might form a multiple edge in case where $[y_1, y_2]$ is simply an edge.) See Fig. 2 for an illustration.

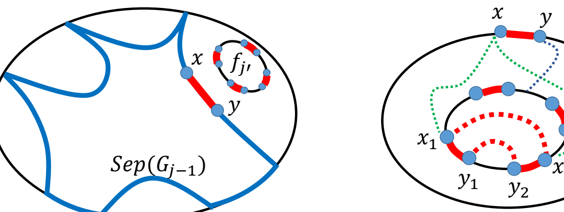


Figure 2 Left: The subgraph G_{j-1}, the topological closure of G_{j-1}, namely, $\overline{\mathcal{O}(G_{j-1})}$ is shown in black, and the cycle separator $\text{sep}(G_{j-1})$ is in blue. The face $f_{j'}$ is fully contained in a region whose induced subgraph (i.e., on the vertices embedded in this region) is not connected in G_{j-1}. Right: Zoom into this region. The red edges indicate virtual edges. The virtual edge (x, y) lies on the boundary of a connected region in $\mathcal{O}(G_{j-1}) \setminus \overline{\mathcal{O}(G_{j-1,in})}$. The removal of (x, y) partitions the vertices embedded in this region into two components, one rooted at x and the other rooted at y. By the Monge property the set of vertices on the face $f_{j'}$ that are connected to x lies on a consecutive segment of the face. This allows us to re-define two almost-faces each will be fully contained in a unique child bag of G_{j-1}.

Finally, it is easy to see by induction on i that each level-i bag contains at most i virtual edges, since each separator $\text{sep}(G'_{j'})$ adds at most one virtual edge to its descendent bags in the BDD tree. Also, since the face split (as described above) occurs only when the almost-face has at least two virtual edges, the number of virtual edges is kept. ◀

Monge-Like Properties of Reachability between Faces. Our compression scheme of the reachability information is based on the well known Monge properties. We begin with encoding the reachability relations between disjoint sets of vertices on a single face, then encoding the reachability between all-pairs on a face, and finally encoding the reachability between vertices on two faces. The next auxiliary lemmas are modifications of Lemmas 2.1, 2.2 and 2.4 in [2]. In [2] the goal was to encode distances in unweighted graphs, whereas here our goal is to encode reachability information. This leads to small modifications both in the statements of the lemmas as well as in the arguments.

▶ **Lemma 10.** *[Analogue of Lemma 2.1, [2]] Let $C = (v_1, v_2, \ldots, v_{|C|})$ be the cyclic walk of a face partitioned into two parts $C_1 = (v_1, v_2, \ldots, v_\ell)$ and $C_2 = (v_{\ell+1}, v_{\ell+2}, \ldots, v_{|C|})$. Then, for any subsets $C'_1 \subseteq C_1$ and $C'_2 \subseteq C_2$ the reachability relations between $C'_1 \times C'_2$ can be encoded using $\widetilde{O}(|C'_1| + |C'_2|)$ bits.*

▶ **Lemma 11.** *[Analogue of Lemmas 2.2, [2]] Let $C = (v_1, \ldots, v_{|C|}\}$ be the cycle walk of a face of a planar graph. Then, all reachability relations for a subset $C' = (v_{q_1}, \ldots, v_{q_s})$ can be encoded in $\widetilde{O}(|C'|)$ bits.*

▶ **Lemma 12.** *[Analogue of Lemmas 2.4 in [2]] Let $C_{in} = (v_1, \ldots, v_q)$ and $C_{ext} = (u_1, \ldots, u_\ell)$ be the cyclic walk of two faces of a planar graph. Then, the reachability relations between any subsets $C'_{in} \subseteq C_{in}$ and $C'_{ext} \subseteq C_{ext}$ can be encoded with $\widetilde{O}(|C'_{in}| + |C'_{ext}|)$ bits*[8].

[8] In Lemma 2.3 of [2], it is required that C'_{ext} is a prefix of C_{ext}. For our purposes, as we care for

The algorithmic implication: We are next ready to state the algorithmic implication of the structural properties provided above, which will be extensively used in our algorithms. We need the following notation. Let u be a vertex and f be a face in G'. Let $V(f)$ be the set of vertices on the boundary of f. A vertex $u \in G'$ is *incoming-active* w.r.t f if there exists some vertex $s \in V(f) \cap \mathrm{sep}^+(G')$ satisfying that $s \preceq_{G'} u$. Otherwise, the vertex is *incoming-inactive*. The notions of outgoing-active and outgoing-inactive are defined accordingly. For two faces f_1 and f_2 in a subgraph G', let $A_{in}(f_1, f_2, G')$ be the set of *incoming-active vertices* in $\mathrm{sep}^+(G') \cap V(f_1)$ w.r.t. f_2 formally defined as follow:

$$A_{in}(f_1, f_2, G') = \{u \in \mathrm{sep}^+(G') \cap V(f_1) \mid \exists v \in \mathrm{sep}^+(G') \cap V(f_2) \text{ satisfying } v \preceq_{G'} u\} . \quad (1)$$

The set of outgoing-active vertices $A_{out}(f_1, f_2, G')$ is defined analogously. These definitions are analougoes when the faces are almost-faces. Throughout, because of the symmetry between computing the incoming vertices and the outgoing vertex, w.l.o.g., when saying active vertices we mean outgoing-active vertices.

▶ **Lemma 13.** *Let G' be bag with a child bag G'_j in the BDD tree, and let $f'_{1,j}, \ldots, f'_{k,j}$ be the almost-faces of G'_j on which the vertices of $\mathrm{sep}^+(G') \cap V(G'_j)$ are lying. Assume that all vertices of G' know: (1) the ordering of the $\mathrm{sep}^+(G') \cap V(G'_j)$ vertices on the $f'_{i,j}$ faces, (2) the set of incoming-active $A_{in}(f'_{i_1,j}, f'_{i_2,j}, G'_j)$ and outgoing-active $A_{out}(f'_{i_1,j}, f'_{i_2,j}, G'_j)$ vertices for every pair of almost-faces $f'_{i_1,j}, f'_{i_2,j}$, $i_1, i_2 \in \{1, \ldots, k\}$. Then, every vertex $u \in f'_{i_2,j}$ can encode the set of all its outgoing vertices from $\mathrm{sep}^+(G') \cap V(f'_{i_1,j})$ using $\widetilde{O}(1)$ bits.*

2.2 The Single Source Reachability Algorithm

The reachability algorithm has two phases. The first phase computes, in a top-bottom manner, a reachability preserver $H(G') = H(\mathrm{sep}^+(G'), G')$ for every bag G' where $(u, v) \in H(G')$ iff $u \preceq_{G'} v$, for every $u, v \in \mathrm{sep}^+(G')$. At the end of this phase, each vertex u knows the preserver $H(G')$ for every bag G' containing u. The second phase is performed locally at each vertex, and does not require any communication. Each vertex u will compute in a top-down manner a reachability label $L_G(u)$ of $\widetilde{O}(D)$ bits, such that given a label $L_G(v)$ it can determine the u-v reachability in G. Thus, by letting the source vertex s send its reachability label, all vertices can determine their reachability to s.

Step 1: Computing the Reachability Preservers

The reachability preservers are computed in a **bottom-up manner** on the BDD tree of depth $d = O(\log n)$. Starting from the leaf level, in every independent level $i \in \{1, \ldots, d\}$ of the recursion, the goal is to compute the reachability preserver $H(G') = H(\mathrm{sep}^+(G'), G')$ for a level-$(d - i + 1)$ bag G' given the reachability preservers of its child bags. Note that the reachability preserver $H(G')$ of each bag is computed based on the reachability in G' rather than G. Our observations for every bag G' and its child bags $G'_1, \ldots, G'_k$, are as follows:

▶ **Observation 14.** *(1) $\mathrm{sep}^+(G') \subseteq \bigcup_{j=1}^{\ell} \mathrm{sep}^+(G'_j)$. (2) For every $s_1, s_2 \in \mathrm{sep}^+(G')$ such that $s_1 \preceq_{G'} s_2$ it holds that $s_1 \preceq_{\bigcup_j \widehat{H}(G'_j)} s_2$ where $\widehat{H}(G'_j) = \{(u, v) \in H(G'_j) \mid u, v \in \mathrm{sep}^+(G') \cap V(G'_j)\}$, i.e., $\widehat{H}(G'_j) = H(G'_j)[\mathrm{sep}^+(G')]$.*

reachability rather than distances, C'_{ext} can be an arbitrary subset of C_{ext}.

Proof. (1) follows by the definition of the set $\mathrm{sep}^+(\cdot)$ sets. To see (2), it is sufficient to prove that for every $s_1, s_2 \in \mathrm{sep}^+(G')$ such that $s_1 \preceq_{G'} s_2$ but $(s_1, s_2) \notin \bigcup_j \widehat{H}(G'_j)$, it holds that $s_1 \preceq_{\bigcup_j \widehat{H}(G'_j)} s_2$.

Let P be the directed path from s_1 to s_2 in G'. Since $\bigcup_j G'_j = G'$, the path P can be decomposed into sub-paths $P = P_1 \circ P_2 \ldots \circ P_\ell$ such that endpoints of all these paths P_k are in $\mathrm{sep}^+(G')$ (in fact, for $k \in \{2, \ldots, \ell-1\}$ both endpoints are in $\mathrm{sep}(G')$). In addition, letting $s_{1,k}, s_{2,k}$ be the endpoint of P_k, then there exists a child bag G'_{j_k} such that $P_k \subseteq G'_{j_k}$. Therefore, $(s_{1,k}, s_{2,k}) \in \widehat{H}(G'_{j_k})$ for every $k \in \{1, \ldots, \ell\}$. Concluding that $s_1 \preceq_{\bigcup_j \widehat{H}(G'_j)} s_2$. ◀

For a child bag G'_j of G', let $f_{1,j}, \ldots, f_{\ell,j}$ be the almost-faces of G'_j on which the vertices of $\mathrm{sep}^+(G') \cap V(G'_j)$ are embedded. The vertices of G' are then required to know the following for every bag G'_j for $j \in \{1, \ldots, k\}$:

- (I1) The orientation of the vertices of $\mathrm{sep}^+(G') \cap V(G'_j)$ on the almost-faces $f_{1,j}, \ldots, f_{\ell,j}$.
- (I2) The active incoming and outgoing sets $A_{in}(f_{i_1,j}, f_{i_2,j}, G'_j)$ and $A_{out}(f_{i_1,j}, f_{i_2,j}, G'_j)$ for every $i_1, i_2 \in \{1, \ldots, \ell\}$ (see Eq. (1) to recall the definition of these sets).

By Lemma 13, each vertex $u \in \mathrm{sep}^+(G') \cap V(G'_j)$ can compress its $\{u\} \times \mathrm{sep}^+(G')$ reachability in the G'_j subgraph using $\widetilde{O}(1)$ bits. As each vertex in G' appears on at most three child bags, all the vertices of $\mathrm{sep}^+(G')$ can send the compression of their $\mathrm{sep}^+(G')$ reachability in every child bag G'_j within $\widetilde{O}(D)$ rounds. As a result, all vertices of G' know the graph $\widehat{H}(G'_j) = H(G'_j)[\mathrm{sep}^+(G')]$ for every child bag G'_j. Finally, each vertex u computes the desired preserver $H(G')$ by locally computing $\mathrm{sep}^+(G') \times \mathrm{sep}^+(G')$ reachability in the union of graphs $\bigcup_j \widehat{H}(G'_j)$. The correctness of this step follows by Obs. 14(2). This completes the description of the algorithm. We next show that all vertices in G' can learn the required information (I1) and (I2) within $\widetilde{O}(D)$ rounds.

▷ Claim 15. (I1, I2) can be obtained in $\widetilde{O}(D)$ rounds;

Proof. By Lemma 9, the vertices of $\mathrm{sep}^+(G')$ lie *consecutively* on $O(\log n)$ almost-faces in G'_j. Note by the proof of Lemma 9, the vertices also know their incident virtual edges on these faces. Each vertex $u \in \mathrm{sep}^+(G') \cap V(G'_j)$ simply sends its edges on these faces. Since each vertex belongs to at most three child bags, and since there are $O(\log n)$ faces, each vertex in $\mathrm{sep}^+(G')$ sends $\widetilde{O}(1)$ bits. Since the vertices of $\mathrm{sep}^+(G')$ lie consecutively on those faces, by knowing all their edges on the faces, the vertices can decide a fixed orientation (i.e., increasing clock-wise ordering from the largest vertex ID. To see (I2), since all vertices of G'_j know the subgraph $H(G'_j)$, and as $\mathrm{sep}^+(G'_j) \subseteq \mathrm{sep}^+(G')$, they know their incoming and outgoing vertices from $\mathrm{sep}^+(G')$ in G'_j. In addition, all vertices know (I1), that is, a fixed ordering of the $\mathrm{sep}^+(G')$ vertices on $O(\log n)$ faces. Therefore each vertex $u \in f'_{i_1,j}$ can send a message containing the indicators of all other faces $f'_{i_2,j}$ for which $u \in A_{in}(f_{i_1,j}, f_{i_2,j}, G'_j)$ and in the same manner the list of faces for which $u \in A_{out}(f_{i_1,j}, f_{i_2,j}, G'_j)$. This message has $O(\log n)$ bits, and since each u in G' belongs to at most three child bags, the entire information on all $A_{in}(f_{i_1,j}, f_{i_2,j}, G'_j)$ and $A_{out}(f_{i_1,j}, f_{i_2,j}, G'_j)$ can be delivered in $\widetilde{O}(D)$ rounds. ◁

We therefore have the following.

▶ **Lemma 16.** *Given a BDD tree $\mathcal{T}$ for a planar graph G, there exists a randomized algorithm that computes for each bag $G' \in \mathcal{T}$ the reachability preserver $H(G') = H(\mathrm{sep}^+(G'), G')$, where each vertex u knows the preserver edges $H(G')$ for every bag G' containing u.*

Step 2: (Locally) Computing the Reachability Labels. We next show that given that each vertex u knows the reachability preservers $H(G') = H(\text{sep}^+(G'), G')$ for every bag G' that contains u, it can *locally* compute its reachability label $L_G(u)$ of $\widetilde{O}(D)$ bits, without any additional communication. We follow the same recursive scheme of the labels by [12, 29] with one key difference as will be explained soon. The label $L_G(u)$ of each vertex $u \notin \text{sep}(G)$ consists of three fields: (i) lists of incoming and outgoing vertices to v from $\text{sep}^+(G)$: $In(u,G) = \{v \in \text{sep}^+(G) \mid v \preceq_G u\}$ and $Out(u,G) = \{v \in \text{sep}^+(G) \mid u \preceq_G v\}$, (ii) the identifier of G's child bag that contains u, and (iii) the (recursive) label $L_{G'}(u)$ where G' is the child component of G that contains u in the BDD decomposition. Since $u \notin \text{sep}(G)$, there exists exactly one such bag. Denoting the first two fields in the label by $\widehat{L}_G(u)$, the label u consists of $k = O(\log n)$ sub-labels $L_G(u) = \widehat{L}_{G_0}(u) \circ \widehat{L}_{G_1}(u) \ldots \circ \widehat{L}_{G_k}(u)$. For $u \in \text{sep}(G)$ the label $L_G(u)$ contains only the first field (i). The key difference compared to [12, 29] is that each vertex u keeps the information in each sub-label $\widehat{L}_{G'}(u)$ with respect to the extended-separator set $\text{sep}^+(G')$ rather than the separator set $\text{sep}(G')$. As will see, this adaptation is critical for our scheme to go through.

We focus on a vertex u and explain how it can compute its label in a bottom-up manner on the BDD tree. To avoid cumbersome notation for a leaf bag G', let $\text{sep}(G') = V(G')$. Let ℓ be the minimum value $i \in \{0, \ldots, d\}$ satisfying that $u \in \text{sep}(G')$ for some i-level bag G'. Note that by definition of the BDD, u belongs to exactly one bag in each of the levels $i \in \{0, \ldots, \ell\}$. Let $G = G_0, G_1, \ldots, G_\ell$ be the unique bags that contain u in level $i \in \{1, \ldots, \ell\}$.

We start with the base case of computing the label $L_{G_\ell}(u)$. There are two cases. If $\ell = d$, then G_d is a leaf bag, and u knows the entire leaf bag information from the graph $H(G_d)$. Otherwise, if $\ell \leq d-1$, then $u \in \text{sep}(G_\ell)$. In this case, the label $L_{G_\ell}(u)$ should contain the lists $In(u, G_\ell)$ and $Out(u, G_\ell)$ which can be obtained directly from the preserver $H(G_\ell)$. Assume that u has locally computed the labels $L_{G_\ell}(u), \ldots, L_{G_{i+1}}(u)$. We now explain how it can compute the label $L_{G_i}(u)$. Observe that the label $L_{G_i}(u)$ has the form $L_{G_i}(u) = \widehat{L}_{G_i}(v) \circ L_{G_{i+1}}(v)$. Therefore, it is sufficient to compute the sets $In(u, G_i)$ and $Out(u, G_i)$. By the sub-label $\widehat{L}_{G_{i+1}}(u)$ of the $L_{G_{i+1}}(u)$ label, u knows the sets $In(u, G_{i+1}), Out(u, G_{i+1})$. The sets $In(u, G_i), Out(u, G_i)$ are then obtained by locally computing the reachability of u w.r.t the $\text{sep}(G_i)$ vertices in the graph

$$H(G_i, u) = H(\text{sep}^+(G_i), G_i) \cup \{(u,v) \mid v \in Out(u, G_{i+1})\} \cup \{(v,u) \mid v \in In(u, G_{i+1})\} .$$

We next show that for every $v \in \text{sep}^+(G_i)$ such that $v \preceq_{G_i} u$, it holds that $v \preceq_{H(G_i,u)} u$, and thus $v \in In(u, G_i)$ as desired. The same argument will apply to the $Out(u, G_i)$ set. Let P be a directed $v \rightsquigarrow u$ path in G_i. If $P \subseteq G_{i+1}$, then since $v \in \text{sep}^+(G_{i+1})$, by induction assumption, $v \in In(u, G_{i+1})$ and thus $(v,u) \in H(G_i, u)$. Otherwise, P must contain at least one vertex from $\text{sep}(G_i)$. Let s^* be the last vertex (closest to u) from $\text{sep}(G_i)$ on P. Then, by the selection of s^*, $P[s^*, u] \subset G_{i+1}$. By induction assumption, as $s^* \in \text{sep}^+(G_{i+1})$, it holds that $s^* \in In(u, G_{i+1})$ and thus $(s^*, u) \in H(G_i, u)$. In addition, since $v, s^* \in \text{sep}^+(G_i)$, we have that $(v, s^*) \in H(G_i)$. Overall, there is a directed path $(v, s^*) \circ (s^*, u)$ in $H(G_i, u)$, and thus $v \in In(u, G_i)$. The claim follows. This completes the $\widetilde{O}(D)$-round algorithm for computing the the reachability labels. By broadcasting the reachability label of the source vertex s, we also solve the single-source reachability problem, and establish Thm. 1. The multi-source reachability algorithm and the algorithms for weighted digraphs are deferred to the full version.

References

1 Amir Abboud, Vincent Cohen-Addad, and Philip N Klein. New hardness results for planar graph problems in p and an algorithm for sparsest cut. In *Proc. of the Symp. on Theory of Comp. (STOC)*, 2020. URL: http://www.wisdom.weizmann.ac.il/~robi/Bertinoro2019_FineGrained/program/program-bertinoro19.html#Cohen-Addad.

2 Amir Abboud, Pawel Gawrychowski, Shay Mozes, and Oren Weimann. Near-optimal compression for the planar graph metric. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 530–549. SIAM, 2018.

3 Glencora Borradaile, Philip N Klein, Shay Mozes, Yahav Nussbaum, and Christian Wulff-Nilsen. Multiple-source multiple-sink maximum flow in directed planar graphs in near-linear time. *SIAM Journal on Computing*, 46(4):1280–1303, 2017.

4 Sergio Cabello. Subquadratic algorithms for the diameter and the sum of pairwise distances in planar graphs. *ACM Transactions on Algorithms (TALG)*, 15(2):1–38, 2018.

5 Shiri Chechik and Doron Mukhtar. Reachability and shortest paths in the broadcast CONGEST model. In *33rd International Symposium on Distributed Computing, DISC 2019, October 14-18, 2019, Budapest, Hungary*, pages 11:1–11:13, 2019.

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 *Proc. of the Symp. on Theory of Comp. (STOC)*, pages 363–372, 2011.

7 Michael Elkin. Distributed exact shortest paths in sublinear time. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, pages 757–770, 2017.

8 Jittat Fakcharoenphol and Satish Rao. Planar graphs, negative weight edges, shortest paths, and near linear time. *Journal of Computer and System Sciences*, 72(5):868–889, 2006.

9 Orr Fischer and Rotem Oshman. A distributed algorithm for directed minimum-weight spanning tree. In *33rd International Symposium on Distributed Computing, DISC 2019, October 14-18, 2019, Budapest, Hungary*, pages 16:1–16:16, 2019.

10 Sebastian Forster and Danupon Nanongkai. A faster distributed single-source shortest paths algorithm. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 686–697. IEEE, 2018.

11 Harold N Gabow. Scaling algorithms for network problems. In *24th Annual Symposium on Foundations of Computer Science (sfcs 1983)*, pages 248–258. IEEE, 1983.

12 Cyril Gavoille, David Peleg, Stéphane Pérennes, and Ran Raz. Distance labeling in graphs. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 210–219. Society for Industrial and Applied Mathematics, 2001.

13 Mohsen Ghaffari and Bernhard Haeupler. Distributed algorithms for planar networks i: Planar embedding. In *the Proc. of the Int'l Symp. on Princ. of Dist. Comp. (PODC)*, pages 29–38, 2016.

14 Mohsen Ghaffari and Bernhard Haeupler. Distributed algorithms for planar networks ii: Low-congestion shortcuts, mst, and min-cut. In *Proc. of ACM-SIAM Symp. on Disc. Alg. (SODA)*, pages 202–219, 2016.

15 Mohsen Ghaffari and Jason Li. Improved distributed algorithms for exact shortest paths. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*, pages 431–444, 2018.

16 Mohsen Ghaffari and Merav Parter. Near-optimal distributed DFS in planar graphs. In *31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria*, pages 21:1–21:16, 2017. URL: http://www.weizmann.ac.il/math/parter/sites/math.parter/files/uploads/planarDFS_DISC17.pdf.

17 Mohsen Ghaffari and Rajan Udwani. Brief announcement: Distributed single-source reachability. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastián, Spain, July 21 - 23, 2015*, pages 163–165, 2015.

18 Bernhard Haeupler, D. Ellis Hershkowitz, and David Wajc. Round- and message-optimal distributed graph algorithms. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing, PODC 2018, Egham, United Kingdom, July 23-27, 2018*, pages 119–128, 2018.

19 Bernhard Haeupler, Taisuke Izumi, and Goran Zuzic. Low-congestion shortcuts without embedding. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, pages 451–460. ACM, 2016.

20 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.

21 Bernhard Haeupler, Jason Li, and Goran Zuzic. Minor excluded network families admit fast distributed algorithms. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing, PODC 2018, Egham, United Kingdom, July 23-27, 2018*, pages 465–474, 2018.

22 Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. A deterministic almost-tight distributed algorithm for approximating single-source shortest paths. In *Proc. of the Symp. on Theory of Comp. (STOC)*, pages 489–498, 2016.

23 Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. A deterministic almost-tight distributed algorithm for approximating single-source shortest paths. *SIAM Journal on Computing*, STOC16:98–137, 2019. doi:10.1137/16M1097808.

24 Giuseppe F Italiano, Yahav Nussbaum, Piotr Sankowski, and Christian Wulff-Nilsen. Improved algorithms for min cut and max flow in undirected planar graphs. In *Proceedings of the forty-third annual ACM symposium on Theory of computing*, pages 313–322, 2011.

25 Arun Jambulapati, Yang P Liu, and Aaron Sidford. Parallel reachability in almost linear work and square root depth. *arXiv preprint arXiv:1905.08841*, 2019.

26 Philip N Klein and Sairam Subramanian. A randomized parallel algorithm for single-source shortest paths. *Journal of Algorithms*, 25(2):205–220, 1997.

27 Christoph Lenzen and Boaz Patt-Shamir. Fast partial distance estimation and applications. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, pages 153–162, 2015.

28 Jason Li. Distributed treewidth computation. *arXiv preprint arXiv:1805.10708*, 2018.

29 Jason Li and Merav Parter. Planar diameter via metric compression. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA, June 23-26, 2019.*, pages 152–163, 2019.

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 Shay Mozes and Christian Sommer. Exact distance oracles for planar graphs. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete algorithms*, pages 209–222. SIAM, 2012.

32 Shay Mozes and Christian Wulff-Nilsen. Shortest paths in planar graphs with real lengths in o (nlog 2 n/loglogn) time. In *European Symposium on Algorithms*, pages 206–217. Springer, 2010.

33 Danupon Nanongkai. Distributed approximation algorithms for weighted shortest paths. In *Proc. of the Symp. on Theory of Comp. (STOC)*, 2014.

34 David Peleg. *Distributed Computing: A Locality-sensitive Approach.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

35 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.

36 Mikkel Thorup. Compact oracles for reachability and approximate distances in planar digraphs. *Journal of the ACM (JACM)*, 51(6):993–1024, 2004.

Coloring Fast Without Learning Your Neighbors' Colors

Magnús M. Halldórsson
Reykjavik University, Iceland
mmh@ru.is

Fabian Kuhn
University of Freiburg, Germany
kuhn@cs.uni-freiburg.de

Yannic Maus
Technion – Israel Institute of Technology, Haifa, Israel
yannic.maus@cs.technion.ac.il

Alexandre Nolin
Reykjavik University, Iceland
alexandren@ru.is

Abstract

We give an improved randomized CONGEST algorithm for distance-2 coloring that uses Δ^2+1 colors and runs in $O(\log n)$ rounds, improving the recent $O(\log \Delta \cdot \log n)$-round algorithm in [Halldórsson, Kuhn, Maus; PODC '20]. We then improve the time complexity to $O(\log \Delta) + 2^{O(\sqrt{\log \log n})}$.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases distributed graph coloring, distance 2 coloring, congestion

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.39

Related Version Full version available under `https://arxiv.org/abs/2008.04303`.

Funding This project was supported by the European Union's Horizon 2020 Research and Innovation Programme under grant agreement no. 755839 and by the Icelandic Research Fund grant 174484.

1 Introduction

The distributed coloring problem is arguably the most intensively studied problem in the area of distributed graph algorithms and certainly also one of the most intensively studied problems in distributed computing more generally. The standard assumption is that the *coloring graph* – the graph on which we want to compute a coloring – is also the *communication network* – the graph forming the network topology. We explore in this paper the case when the latter is weaker than the former: the communication is constrained, and direct links are not available to all the "neighbors" that are to be colored differently.

The primary setting for this is the *distance-2 coloring* problem in the standard distributed CONGEST model. Given a graph $G = (V, E)$, in the *d2-coloring* problem on G, the objective is to assign a color x_v to each node $v \in V$ such that any two nodes u and v at distance at most 2 in G are assigned different colors $x_u \neq x_v$. Equivalently, d2-coloring asks for a coloring of the nodes of G such that for every $u \in V$, all the nodes in the set $\{u\} \cup N(u)$ (where $N(u)$ denotes the set of neighbors of u) are assigned distinct colors. Further note that d2-coloring on G is also equivalent to the usual vertex coloring problem on the graph G^2, where $V(G^2) = V$ and there is an edge $\{u, v\} \in E(G^2)$ whenever $d_G(u, v) \leq 2$.

© Magnús M. Halldórsson, Fabian Kuhn, Yannic Maus, and Alexandre Nolin;
licensed under Creative Commons License CC-BY
34th International Symposium on Distributed Computing (DISC 2020).
Editor: Hagit Attiya; Article No. 39; pp. 39:1–39:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The CONGEST model is a standard synchronous message passing model [32]. The graph on which we want to compute a coloring is also assumed to form the network topology. Each node $u \in V$ of the graph has a unique $O(\log n)$-bit identifier $\text{ID}(u)$, where $n = |V|$ is the number of nodes of G. Time is divided into synchronous rounds and in each round, each node $u \in V$ of G can do some arbitrary internal computation, send a (potentially different) message to each of its neighbors $v \in N(u)$, and receive the messages sent by its neighbors in the current round. If the size of the messages is not restricted, the model is known as the LOCAL model [29, 32]. In the CONGEST model, it is further assumed that each message consists of at most $O(\log n)$ bits.

As our main result, we give an efficient $O(\log n)$-time randomized algorithm for d2-coloring G with at most $\Delta^2 + 1$ colors, where Δ is the maximum degree of G. This improves on a recent $O(\log \Delta \cdot \log n)$-time algorithm [23] and it matches the best known bound for ordinary distance-1 $(\Delta + 1)$-coloring in CONGEST as a function of n alone. We further explore more efficient algorithms when $\Delta \ll n$. Combining our main method with a range of powerful recent techniques, we obtain an algorithm that runs in time $O(\log \Delta) + 2^{O(\sqrt{\log \log n})}$.

Before discussing our results in more detail, we first discuss why we believe d2-coloring is interesting, what is known for the corresponding coloring problems on G and why it is challenging to transform CONGEST algorithms to color G into CONGEST algorithms for d2-coloring.

Wireless networking is a major motivation for distance-2 coloring, where nodes with a common neighbor should not simultaneously communicate to avoid a collision at the common neighbor [15, 31]. While the coloring is to be used for scheduling, the wireless channel need not be the medium for *computing* the coloring. With the advent of software-defined radio and hierarchical / heterogeneous networks, it is well motivated to consider coloring computation in a communication model more powerful than radio networks. Yet, asking for the different-message-to/from-all-neighbors feature of CONGEST may be hoping for too much. More generally, we view it as a major question in distributed graph algorithms whether one can relax the communication requirements for graph coloring. We ask:

How constrained can the communication structure be to allow for fast (logarithmic, sublogarithmic) distributed graph coloring computation?

Distributed d2-coloring is an interesting and important problem for several other reasons. The d2-coloring problem for example also occurs naturally when single-round randomized algorithms are derandomized using the method of conditional expectation [21]. d2-coloring in CONGEST is further of special interest as it appears to lie at the edge of what is computable efficiently, i.e., in polylogarithmic time, while distance-3 coloring is even hard to verify [18].

Distance-k problems have not been addressed widely in a distributed setting, partly because distance-k communication can be simulated in k steps of the LOCAL model. In CONGEST, the situation changes drastically as simulating a single round of a distance-1 coloring algorithm can incur a factor $\Theta(\Delta^{k-1})$ overhead, i.e., even for $k = 2$, the overhead can be linear in Δ. Even the very simple algorithm where each node picks a random available color cannot be efficiently used for d2-coloring as it is in general not possible to keep track of the set of colors chosen by 2-hop neighbors in time $o(\Delta)$. Recently, Halldórsson, Kuhn and Maus [23] treated d2-coloring in CONGEST and gave a randomized algorithm using $\Delta^2 + 1$-colors in $O(\log \Delta \cdot \log n)$ rounds, as well as a deterministic algorithm using $(1 + \epsilon)\Delta^2$ colors in $\mathit{poly}(\log n)$ rounds. Our main approach builds heavily on their framework, while simplifying certain features and strengthening structural properties. Distributed graph optimization problems on G^2 (with CONGEST-communication in G) such as vertex cover and minimum dominating set have recently been studied in [5].

Distributed graph coloring. The standard variant of the distributed coloring problem on G asks for computing a vertex coloring with at most $\Delta+1$ colors, which is computed by a simple sequential greedy algorithm. The main focus in the literature on distributed coloring has been on the LOCAL model, where by now the problem is understood relatively well. The best randomized $(\Delta+1)$-coloring algorithm known in the LOCAL model, due to Chang, Li, and Pettie [14], runs in $\text{poly}\log\log n$ rounds. The complexity given in [14] is $2^{O(\sqrt{\log\log n})}$, while the improvement to $\text{poly}\log\log n$ immediately follows from the recent breakthrough work on deterministic *network decomposition* of Rozhoň and Ghaffari [33]. For a more detailed discussion of related work on distributed coloring, we refer to [8, 14, 27].

While most known distributed coloring algorithms were developed for the LOCAL model, many of them work directly in the CONGEST model, including those in [1, 30, 29, 26, 28, 7, 10, 6, 9, 27, 4]. Still, the best complexity known for coloring in CONGEST, as a function of n alone, is $O(\log n)$, which is achieved by the following very simple OneShotColoring algorithm: Initially all nodes are uncolored. The algorithm runs in synchronous phases, where in each phase, each still uncolored node v chooses a uniform random color among its available colors (i.e., among the colors that have not already been picked by a neighbor) and v keeps the color if no of its uncolored neighbors tries the same color at the same time [26, 11].

The only known published algorithm in CONGEST with a better bound is due to Ghaffari [19], who obtains a $(\Delta+1)$-coloring in time $O(\log\Delta)+2^{O(\sqrt{\log\log n})}$. The second term is due to a network decomposition algorithm also introduced in [19]. Unlike for results in the LOCAL model, it is *not* directly possible to replace this decomposition with the recent construction in [33] to improve the dependence on n. The reason is that the complexity of the network decomposition construction of [33] grows at least linearly in the length of the node identifier bit strings. In the LOCAL model, it is possible to use a standard coloring algorithm of [29] to first map the IDs to $O(\log\log n)$-bit values that are unique up to a sufficient distance so that one can afterwards apply the algorithm of [33]. Subsequent to the publication of our results [20] improved upon the network decomposition algorithm from [33] (to deal with large IDs in the CONGEST model) and as a result obtains a $O(\log\Delta)+\text{poly}\log\log n$ CONGEST algorithm for $(\Delta+1)$-coloring. Note that if we have graphs of size N and if IDs and colors can be represented with $\text{poly}\log N$ bits, there is a recent *deterministic* $(deg+1)$-list coloring algorithm running in $\text{poly}\log N$ time in CONGEST [4].

1.1 Contributions

We provide two efficient randomized CONGEST model algorithms to compute a d2-coloring of a given n-node graph $G=(V,E)$. If Δ is the maximum degree of G, the maximum degree of any node in G^2 is at most $\Delta+\Delta\cdot(\Delta-1)=\Delta^2$. As a natural analog to studying $(\Delta+1)$-coloring on G, we study the problem of computing a d2-coloring with Δ^2+1 colors.

▶ **Theorem 1.1.** *There is a randomized CONGEST algorithm that d2-colors a graph with Δ^2+1 colors in $O(\log n)$ rounds, with high probability.*

The algorithm is given in Sec. 2, with the key ideas and challenges outlined at the start of the section. Our second algorithm is more efficient if $\Delta \ll n$.

▶ **Theorem 1.2.** *There is a randomized CONGEST algorithm that d2-colors a graph with Δ^2+1 colors in $O(\log\Delta)+2^{O(\sqrt{\log\log n})}$ rounds, with high probability.*

Theorem 1.2 relies on the network decomposition algorithm of [22] that can compute a suitable network decomposition of G^2 despite a large ID space in $2^{O(\sqrt{\log\log n})}$ rounds. The unpublished result in [20] computes similar network decompositions despite a large ID space

in poly log log n rounds, but only for G and not for G^2. If the results of [20] can be extended to G^2, which in fact is likely, the runtime of Theorem 1.2 improves to $O(\log\Delta)+\text{poly}\log\log n$, and it then again – as at the time of submission of this manuscript – matches the complexity of ordinary distance-1 coloring in CONGEST. The key ideas of Theorem 1.2 appears in Sec. 3. Formal proofs of all statements as well as efficient implementations are discussed in detail in the full version [24].

2 Logarithmic Time Randomized Algorithm

We give randomized CONGEST algorithms that form a d2-coloring using Δ^2+1 colors. We first introduce notation that we use frequently throughout the proofs in this section.

Notation. The *palette* of available colors is $\{0, 1, 2, \ldots, \Delta^2\}$. The neighbors in G of a node are called *immediate neighbors*, while the neighbors in G^2 are *d2-neighbors*. For a (sub)graph K, let $N_K(v)$ denote the set of neighbors of v in K, and let $K[v] = K[N_K(v)]$ denote the subgraph induced by these neighbors. A node is *live* or *uncolored* until it becomes *colored*. An edge in G^2 corresponds to a 2-path (path of length 2) in G; thus, G^2 can have parallel edges.

A node has *slack* q if the number of colors of d2-neighbors plus the number of live d2-neighbors is Δ^2+1-q. In other words, a node has slack q if its palette size is an additive q larger than the number of its uncolored $d2$-neighbors.

An event holds *w.h.p.* (with high probability), if for any $c > 0$, we can choose the constants involved so that the event holds with probability $1-O(n^{-c})$.

2.1 Overview

Our algorithm builds on the approach of [23], which we first summarize. The simple *informed* color guessing approach – each node tries a random color not used by its d2-neighbors – fails because the nodes do not have the bandwidth to learn those colors. A simple *uninformed* approach – trying any random color – works fine if there is sufficient slack, either because the palette is strictly larger than the degree, or in the beginning when few neighbors have been colored. In this case, even trying a uniformly random color is successful with constant probability. If the node has a *sparse* neighborhood then in the very first round, many pairs of d2-neighbors will conveniently adopt the same color, as proved by Elkin, Pettie and Su [17], creating the needed slack. We are then left to deal with denser neighborhoods, of varying average non-degree.

The key idea of [23] is to have the colored nodes "help" the *live* nodes by checking random colors on their neighborhoods. This provides a probabilistic filter that helps reduce the load of the live nodes. It turns out that this alone is not sufficient due to *false negatives*: the helper may reject good colors because it has neighbors with those colors. The solution is for the helper to also query one of its neighbor w, and forward its color if w is not a d2-neighbor of the live node. It is shown that one of these forms of advice is good with constant probability, but could only argue that for those live nodes with a sparsity in a given range considered. This meant that the round complexity of the method had an extra $O(\log\Delta)$ factor for ranging through the different sparsity levels, on top of the $\log n$ factor for finishing off all nodes of that sparsity.

The main technical ingredient behind our $O(\log n)$-round algorithm is the adaptation and extension of the *almost-clique decomposition* (ACD) method initially proposed by Harris, Schneider and Su [25] for the LOCAL model and expanded by Assadi, Lee and Khanna

[2] for streaming and massively parallel settings. The nodes are partitioned into a set of sparse nodes – which can be handled by uninformed guesses – and low-diameter clusters of dense nodes. The ACD achieves the same aims as the *similarity graphs* of [23] that guide the querying and ensure effective filtering, but attain some additional crucial properties such as near-regular high degree. Our extension to ACD is to ensure that all nodes outside clusters have a low degree into the cores of the clusters, strengthening the divide between inside and outside. The decomposition additionally simplifies the technical arguments, including load balancing and probabilistic independence. The key property that we then obtain is that in each iteration, every live node (with at least logarithmic size palette) becomes colored with constant probability. That makes even faster algorithms possible, as we show in the next section. To finish off the nodes with a palette of at most logarithmic size, we apply a second method of [23] black-box, which *learns* the palette of the live nodes and then performs informed color guessing.

2.2 Algorithm Description

We now outline our algorithm, followed by details on the implementation.

Each live node v repeatedly *tries* a suggested color, which means to first *validate* it and then *contest* it. Validating a color means sending it to all immediate neighbors, who then report back if they or any of their neighbors had already adopted that color. Contesting a validated color means proposing it to intermediate neighbors, who report back if any other node also proposes it. If all answers are negative, then v adopts the color.

In what follows, let $\epsilon = 1/60$, $c_0 = 48e^4/\epsilon^2$, and c_3 be a constant to be determined. Also, c_2 is a sufficiently large constant needed for concentration.

Algorithm D2-COLOR

If $\Delta^2 \geq c_2 \log n$ then
1. Compute an almost-clique decomposition.
2. repeat $c_0 \log n$ times:
   Each live node picks a random color and *tries* it.
3. repeat $c_3 \log n$ times
   REDUCE-PHASE()
4. LEARNPALETTE()

FINISHCOLORING()

We will discuss and analyze Steps 1–3 of the above algorithm in detail in the following. The remaining steps, LEARNPALETTE() and FINISHCOLORING(), are from [23]. In LEARNPALETTE(), each live node *learns the palette* of still available colors by cooperatively tallying the colors of d2-neighbors. In FINISHCOLORING(), the maximum degree is sufficiently small so that we can efficiently simulate the classic algorithm of informed color guessing to color the remaining live nodes.

The first step of the algorithm is to compute a decomposition of the nodes into: a) a set of nodes inducing a subgraph (in G^2) that is sufficiently sparse and b) a disjoint collection of almost-cliques (also in G^2). In the following, each of the almost-cliques is called a component C and we use two graphs H and $\hat{H}$ (closely related to the ones in [23]) that both essentially consist of all the components (i.e., all the almost-cliques) and all the G^2-edges connecting two nodes in the same component. Also computed within each component is a spanning tree for a fast aggregation. The exact definitions of the decomposition and of the graphs H and $\hat{H}$ appear in Subsection 2.3.

We next detail the steps of REDUCE-PHASE(), which is the core piece of our algorithm.

Algorithm REDUCE-PHASE()

1. Each live node randomly decides to be *active* with probability 1/8. All other nodes are *inactive*.
2. Compute ϕ, the number of active live nodes in the component C, and distribute it to the nodes of C.
3. Each inactive node $u \in C$ computes ϕ_u, the number of 2-paths to active nodes (by asking its immediate neighbors of their active immediate neighbors). u flips a biased coin: with probability $\min(1, \phi_u/(4\phi))$ it picks one of the ϕ_u paths uniformly at random, while with probability $\max(0, 1 - \phi_u/(4\phi))$, u stops the execution of this iteration. Let v denote the active node at the other end of the path chosen. u verifies that it has only one 2-path to u (by inquiring to its immediate neighbors), and otherwise stops execution of this iteration.
4. u picks a random color $\hat{c}$ different from its own. If that color is not used by any of its $\hat{H}$-neighbors, then u sends the color to v as a proposal, assigning it a uniformly random priority. v tries the proposed color of highest priority (if any).
5. u sends query (v, u) along a random 2-path to an inactive $\hat{H}$-neighbor w, and assigns it a random priority.
6. Upon receipt of a query, node w selects the highest priority query (v, u), checks if v is a d2-neighbor, and if v and w are not d2-neighbors, it sends its color $c(w)$ to v (through u).
7. The active node v tries a color chosen uniformly random among the received proposed colors from Step 6 (if any).

A colored node assists an active node v in two ways: a) guesses and validates a random color for v to try, and b) sees if a random d2-neighbor is also a d2-neighbor of v. This is a probabilistic filter that reduces the workload of the active nodes. The key idea is that one of these forms of assistance is likely to be successful.

Complexity. We discuss the almost-clique decomposition in the next subsection and show how to implement it in $O(\log n)$ rounds, w.h.p. The second step clearly takes $\Theta(\log n)$ rounds. The procedure REDUCE-PHASE takes 24 rounds, or 8 (Step 2), 2 (Step 3), 2 (Step 4), 2 (Step 5), 6 (Step 6), and 4 (Step 7, including the notification of a new color).

Outline of this section: In Sec. 2.3 we describe the almost-clique decomposition and derive key structural properties of dense subgraphs, and in Sec. 2.4 we prove the correctness of the algorithms, i.e., we show that any node is colored w.h.p. after $O(\log n)$ rounds.

2.3 Almost-Clique Decomposition

We next define the notion of local sparsity and the almost-clique decomposition that we use in our paper. The first definition is a slight adaptation of a similar definition in [17].

▶ **Definition 2.1.** *A node v is ζ-*sparse *(or* has sparsity *ζ) if $G^2[v]$ contains $\binom{\Delta^2}{2} - \Delta^2 \cdot \zeta$ (distinct) edges.*

Sparsity is a rational number that indicates how many edges are missing from $G^2[v]$, compared with the densest case (when v's d2-neighborhood is a Δ^2-clique). If no pairs of d2-neighbors of v are adjacent, then $\zeta = (\Delta^2 - 1)/2$, while if $G^2[v]$ forms a Δ^2-clique, then $\zeta = 0$. Elkin, Pettie and Su [17] formalized the connection between sparsity and slack that appears after trying one uniformly random color.

▶ **Proposition 2.2** ([17], Lemma 3.1). *Let v be a vertex of sparsity ζ and let Z be the slack of v after trying a single random color. Then,* $\Pr[Z \leq \zeta/(4e^3)] \leq e^{-\Omega(\zeta)}$.

We require the constant c_2 to be such that if $\zeta \geq c_2 \log n$, then the contrapositive of Prop. 2.2 yields that $Z \geq \zeta/(4e^3)$, w.h.p.

Decomposition. We adapt the almost-clique decomposition of [2] (building on [25]) for the distance-2 setting in CONGEST and endow it with an additional property.

▶ **Definition 2.3.** *Assume $\epsilon \leq 1/60$. Nodes u and v are* ϵ-similar *if they share at least $(1-\epsilon)\Delta^2$ common d2-neighbors. An* almost-clique decomposition *(ACD) with parameter ϵ is a collection of sets $V_*, \hat{C}_1, \hat{C}_2, \dots, \hat{C}_k$ that cover V and where the $\hat{C}_i$ are disjoint. Denote $C_i = \hat{C}_i \setminus V_*$, for $i = 1, \dots, k$. The decomposition satisfies the following properties:*

1. *The nodes in V_* have sparsity at least $\epsilon^2\Delta^2/4$.*
2. *For any $i \in [k]$, C_i and $\hat{C}_i$ satisfy:*
   a. $|C_i| \geq (1-2\epsilon)\Delta^2$.
   b. *The nodes in $\hat{C}_i$ are mutually 10ϵ-similar.*
   c. *Each $v \in \hat{C}_i$ has at most $28\epsilon\Delta^2$ d2-non-neighbors in $\hat{C}_i$ (i.e., $|\hat{C}_i \setminus N_{\hat{C}_i}(v)| \leq 28\epsilon\Delta^2$).*
   d. *Each $v \in \hat{C}_i$ has at least $(1-10\epsilon)\Delta^2$ d2-neighbors in C_i.*
   e. *Each $v \in C_i$ is ϵ-dissimilar to every node outside $\hat{C}_i$.*

We refer to each C_i as a *component* and $\hat{C}_i$ as an *extended component*. The properties imply additional ones: Each extended component is of size at most $(1+28\epsilon)\Delta^2$; and any two nodes in an extended component are within two hops (in G^2). The additional property we need that is not in the formulations of [2] or [25] is Property 2(e).

Let H denote the subgraph of G^2 induced by the components $C_1, \dots, C_k$, i.e., $H = \cup_i G^2[C_i] = (V \setminus V_*, E_H)$ where E_H consists of the pairs of d2-neighbors within the same component. Similarly, let $\hat{H} = \cup_i G^2[\hat{C}_i]$. We consider H, $\hat{H}$ and G^2 to be *simple* graphs, ignoring multiple 2-paths between the same pair of nodes.

▶ **Lemma 2.4.** *There is an $O(\log n)$-round CONGEST algorithm to form an almost-clique decomposition, for any fixed $\epsilon > 0$. Afterwards, each node knows its component ID.*

It is somewhat surprising that such a decomposition can be established efficiently in CONGEST. The key implementation ideas are in [2] for other models, which are essentially based on randomly sampling nodes. In the distance-2 setting we have the additional challenge of communication with one's d2-neighbors, but the key is to have both parties communicate only with the intermediate node that makes the deciding.

We strengthen the ACD-properties for dense nodes and show that they scale with the node sparsity. Note that dense nodes can have non-trivial sparsity and it is crucial in our argument to leverage the corresponding slack.

▶ **Lemma 2.5.** *Let $\epsilon \leq 1/30$. Let v be a node of sparsity ζ in an almost-clique C and extended component $\hat{C}$. Then,*

1. *v has at least $\Delta^2 - (2\zeta+1)/\epsilon$ $\hat{H}$-neighbors (in $\hat{C}$),*
2. *v has at most $|\hat{C} \setminus N_{G^2}(v)| \leq 3\zeta$ $\hat{H}$-non-neighbors, and*
3. *The number of edges in $\hat{H}[v]$ is at least $|E(\hat{H}[v])| \geq \binom{\Delta^2}{2} - (2/\epsilon+1)\zeta\Delta^2$.*

Proof. Recall that by the definition of sparsity, $G^2[v]$ has exactly $\Delta^2((\Delta^2-1)/2-\zeta)$ edges.

1. A d2-neighbor of v that is not ϵ-similar to v can share at most $(1-\epsilon)\Delta^2$ common d2-neighbors with v by ACD property 2(e). In other words, the d2-neighbors of v that are not $\hat{H}$-neighbors can have degree at most $(1-\epsilon)\Delta^2$ in $G^2[v]$. The number of edges in $G^2[v]$ is then at most

$$\frac{1}{2}\left(|N_{\hat{H}}(v)|\Delta^2 + (|N_{G^2}(v)| - |N_{\hat{H}}(v)|)(1-\epsilon)\Delta^2\right) \leq \frac{\Delta^2}{2}\left((1-\epsilon)\Delta^2 + \epsilon|N_{\hat{H}}(v)|\right) .$$

Combining the two bounds on the number of edges in $G^2[v]$,

$$\epsilon|N_{\hat{H}}(v)| \geq \Delta^2 - 1 - 2\zeta - (1-\epsilon)\Delta^2 = \epsilon\Delta^2 - 1 - 2\zeta .$$

Namely, the number of $\hat{H}$-neighbors of v is at least $\Delta^2 - (2\zeta+1)/\epsilon$.

2. By sparsity, there are at most $(2\zeta+1)\Delta^2$ edges of $\hat{H}$ with exactly one endpoint in $N_{G^2}(v)$. Nodes in $\hat{C} \setminus N_{G^2}(v)$ share at least $(1-10\epsilon)\Delta^2$ d2-neighbors with v, by ACD property 2(b). Thus, there are at most $2\zeta\Delta^2/((1-10)\epsilon\Delta^2) = 2\zeta/(1-10\epsilon) \leq 3\zeta$ nodes in $\hat{C}$ that are not d2-neighbors of v, using that $\epsilon \leq 1/30$.
3. By **1** of this lemma, v has degree at least $\Delta^2 - q$ in $\hat{H}$, where $q = (2\zeta+1)/\epsilon$. The at most q nodes in $N_{G^2}(v) \setminus N_{\hat{H}}(v)$ have degree sum at most $q(\Delta^2 - q)$. Thus, the number of edges in $\hat{H}[v]$ is at least $\binom{\Delta^2}{2} - \zeta\Delta^2 - q\Delta^2$. ◀

2.4 Correctness

We prove that D2-COLOR correctly d2-colors G with Δ^2+1 colors in $O(\log n)$ rounds. We assume that the almost-clique decomposition and the graphs H and $\hat{H}$ have been correctly constructed, in the sense of Def. 2.3. Also, that nodes of sparsity $\zeta \geq c_2 \log n$ have slack at least $\zeta/(4e^3)$ as promised by Prop. 2.2. All statements in this section are conditioned on these events.

We first give a high-level proof which encapsulates the core of the technical argument in the following lemma, which is then proven in the upcoming subsubsection.

▶ **Lemma 2.6.** *There is an absolute constant c' such that the following holds. For a live node v in given iteration of REDUCE-PHASE, there is a subset $S \subseteq \psi_v$ of size at least $|\psi_v|/2$ such that each color in S has probability at least $1/(c'|\psi_v|)$ of being validated by v.*

We then easily dispose of the sparse nodes. Since they have slack linear in their degree, they get colored with constant probability in each round, simply by contesting a uniformly random color.

▶ **Lemma 2.7.** *Every node in V_* is colored after Step 2 of D2-COLOR, w.h.p.*

Proof. Let $v \in V_*$. By Def. 2.3(1), v has sparsity at least $\zeta \geq \epsilon^2\Delta^2/4$, and by Prop. 2.2, it has slack at least $c_{13} \doteq \epsilon^2/(16e^3)$, w.h.p. Furthermore, the probability that no d2-neighbor of v tries the same color in the same round is at least $(1-1/(\Delta^2+1))^{\Delta^2} \geq 1/e$, using that $(1-1/x)^{x-1} \geq 1/e$ for any $x > 1$. Thus, with probability at least c_{13}/e, v becomes colored in that round. Hence, the probability that it is not colored in all $c_0 \log n$ rounds is at most $(1-c_{13}/e)^{c_0 \log n} \leq e^{-c_0 c_{13}/e \log n} \leq n^{-c_0 c_{13}/e} = n^{-3}$, since $c_0 = 48e^4/\epsilon^2 = 3e/c_{13}$. ◀

▶ **Theorem 1.1.** *There is a randomized CONGEST algorithm that d2-colors a graph with Δ^2+1 colors in $O(\log n)$ rounds, with high probability.*

Proof. By Lemma 2.7, it suffices to focus on the dense nodes. We first claim that in each iteration, each live node v with palette size $\Omega(\log n)$ becomes colored with a constant non-zero probability.

Consider a given iteration and a live node v. With probability $1/8$, v is active. It has at most $|\psi_v|$ live neighbors and expected at most $|\psi_v|/8$ are active. By Markov's inequality, at most $|\psi_v|/4$ are active, with probability at least $1/2$. By Lemma 2.6, there is a subset $S \subseteq \psi_v$ of size at least $|\psi_v|/2$ such that each color in S has probability at least $1/(c'|\psi_v|)$ of being validated. Independent of what these active neighbors choose, there is then a subset of at least $|\psi_v|/2 - |\psi_v|/4 = |\psi_v|/4$ colors that are available to v, i.e., are not contested by d2-neighbors of v in that iteration. The probability that one of them is validated, and leading to a valid coloring of v, is then at least $c_* = \frac{1}{8} \cdot \frac{1}{2} \cdot \frac{|\psi_v|/4}{c'|\psi_v|} = \frac{1}{64c'}$, establishing the claim.

Applying a Chernoff bound to the above claim, after $5/c_* \cdot \log n$ iterations of Reduce-Phase, it holds with probability at least $1 - 1/n^3$ that all nodes are either colored or have palette size $O(\log n)$ (in which case they have $O(\log n)$ uncolored d2-neighbors). The coloring is then completed by the two algorithms of [23], both running in $O(\log n)$ rounds. ◀

2.4.1 Proof of Lemma 2.6

We prove our main result in two parts, given in Lemmas 2.11 and 2.12, distinguishing between the two forms of making progress: based on Step 4 or Steps 5-7 of Reduce-Phase.

▶ **Definition 2.8.** *An inactive node is* v-decent *(or just* decent*) if it has at most* 4ϕ *2-paths in its almost-clique* C *to active nodes (in* C*)* and *has exactly one 2-path to* v*.*

The distinction between 2-paths and d2-neighbor relations is the rationale for the *decent* definition. Those nodes with lots of paths to active nodes can cause much congestion with poor proposals, while being of limited use to those H-neighbors to which they have few paths.

▶ **Lemma 2.9.** *Let* v *be a live node and* w *be a node, both in* $\hat{C}$*. Then,* v *and* w *have at least* $\Delta^2/4$ *common d2-neighbors in* C *that are* v*-decent.*

Proof. The two nodes v and w are 10ϵ-similar, by Def. 2.3(2b). Since v has at least $(1-10\epsilon)\Delta^2$ distinct d2-neighbors in C (by Def. 2.3(2d)), they share at least $(1-20\epsilon)\Delta^2 \geq 2\Delta^2/3$ d2-neighbors in C (using that $\epsilon \leq 1/60$). This also means that there are at most $10\epsilon\Delta^2 \leq \Delta^2/6$ nodes in C with multiple 2-paths to v. Also, since there are at most $\phi\Delta^2$ total number of 2-paths to the ϕ live nodes in C, there are at most $\Delta^2/4$ nodes with 4ϕ or more 2-paths to live nodes. Hence, there are at least $2\Delta^2/3 - \Delta^2/6 - \Delta^2/4 = \Delta^2/4$ common d2-neighbors of v and w in C that are decent, i.e., have at most 4ϕ 2-paths to active nodes and exactly one 2-path to v. ◀

A color proposed to an active node v is *bad* if it is already assigned to a d2-neighbor of v. Namely, it is bad if it is a "false positive".

▶ **Lemma 2.10.** *The expected number of bad proposals generated in Step 4 for* v *is at most* $(1/\epsilon + 1)\zeta/\phi$*.*

Proof. Let u be an inactive H-neighbor of v. Let Y_u be the event that u picks v in Step 2, and note that $\Pr[Y_P] \leq 1/(4\phi)$. Let X_u be the event that u generates a bad proposal for v in Step 4. That event occurs when u's randomly chosen color is used by a node in S_u, where $S_u = N_{G^2}(v) \setminus N_{\hat{H}}[u]$ is the set of d2-neighbors of v that are not $\hat{H}$-neighbors of u (nor u itself).

The number of such colors is at most $|S_u| = |N_{G^2}(v) \setminus N_{\hat{H}}[u]| \leq (\Delta^2 - 1) - |N_{\hat{H}}(u) \cap N_{\hat{H}}(v)|$. There are at most Δ^2 colors to choose from – all except the one on u – so

$$\Pr[X_u|Y_u] \leq \frac{|S_u|}{\Delta^2} \leq \frac{(\Delta^2-1) - |N_{\hat{H}}(u) \cap N_{\hat{H}}(v)|}{\Delta^2} .$$

By applying Lemma 2.5(3), we have that

$$\sum_{u \in N_{\hat{H}}(v)} |N_{\hat{H}}(u) \cap N_{\hat{H}}(v)| = 2|E(\hat{H}[v])| \geq \Delta^2(\Delta^2 - 1) - (4/\epsilon + 2)\zeta\Delta^2 .$$

Combining the two bounds, letting I denote the set of inactive H-neighbors of v, we get that

$$\sum_{u \in I} \Pr[X_u|Y_u] \leq \sum_{u \in N_{\hat{H}}(v)} \Pr[X_u|Y_u] \leq (4/\epsilon + 2)\zeta .$$

Hence, the expected number of bad proposals generated for v is

$$\sum_{u \in I} \Pr[X_u \cap Y_u] = \sum_{u \in I} \Pr[Y_u] \cdot \Pr[X_u|Y_u] \leq \frac{1}{4\phi} \sum_{u \in I} \Pr[X_u] \leq \frac{(4/\epsilon + 4)\zeta}{4\phi} .$$ ◀

Let ψ_v denote the set of colors in v's palette before a given round, i.e., the set of colors that have not already been taken by its d2-neighbors. Let $\overline{\psi_v}$ be the set of colors in v's palette that appear on nodes in $\hat{C}$. These colors must then appear only on non-$\hat{H}$-neighbors of v.

▶ **Lemma 2.11.** *Suppose $|\psi_v| \geq 2|\overline{\psi_v}|$ and $|\psi_v| = \Omega(\log n)$. Then, there is an absolute constant c such that each color in $\psi_v \setminus \overline{\psi_v}$ has probability at least $1/(c|\psi_v|)$ of being validated and contested by v in Step 4.*

Proof. Let $\hat{\psi} = \psi_v \setminus \overline{\psi_v}$. Any color from $\hat{\psi}$ that is guessed in Step 4 (by some H-neighbor u of v) becomes a *good* proposal to v (i.e., one that would pass validation). Let A (B) denote the expected number of good (bad) proposals to v, respectively. Let q be a color in $\hat{\psi}$ and let A_q be the expected number of proposals of q to v. We shall show that A_q is large, for colors in $\hat{\psi}$, and thus A is large in comparison to B. We then show that A_q is also large relative to the total number of proposals, $A + B$.

The probability that a decent H-neighbor u chooses to help v is $1/(4\phi)$, and the probability that it guesses q is $1/\Delta^2$. By Lemma 2.9, v has at least $\Delta^2/4$ decent H-neighbors. Summing up, $A_q \geq \sum_u 1/(4\phi) \cdot 1/\Delta^2 \geq 1/(16\phi)$, and $A \geq \sum_{q \in \hat{\psi}} A_q \geq |\hat{\psi}_v|/(16\phi) \geq |\psi_v|/(32\phi)$. By Lemma 2.10, $B \leq (1/\epsilon+1)\zeta/\phi$ and by Prop. 2.2, $|\psi_v| \geq \zeta/(4e^3)$. Thus, $B \leq (128e^3(1/\epsilon+1))A$. We can also bound A from above, summing over the at most Δ^2 H-neighbors and all the colors in v's palette:

$$A \leq \sum_{q' \in \psi_v} \sum_{u \in N_H(v)} \frac{1}{4\phi\Delta^2} = \frac{|\psi_v|}{4\phi} \leq 4|\psi_v|A_q .$$

By Markov's inequality, the probability that at most $2(A+B)$ proposals are generated for v is at least $1/2$. The probability that a proposal of q is chosen for validation is then at least

$$\frac{A_q}{4(A+B)} \geq \frac{A/(4|\psi_v|)}{4(1 + 128e^3(1/\epsilon + 1))A} = \frac{1}{16(1 + 128e^3(1/\epsilon + 1))|\psi_v|} .$$ ◀

▶ **Lemma 2.12.** *Suppose $|\psi_v| < 2|\overline{\psi_v}|$. Then, there is an absolute constant c such that each color in $\overline{\psi_v}$ has probability at least $1/(c|\psi_v|)$ of being validated and contested by v in Step 7.*

Proof sketch. For success in Step 7, only colors of nodes in $\hat{C}$ that are not d2-neighbors of v count (and by Lemma 2.5(2), there are at most 3ζ such nodes). The proof (in the full version) requires on one hand to lower bound the probability of a given such node is contacted on behalf of v. The technically more involved task is to show that such a query to w will survive the competition, both at w and at v. ◀

Lemma 2.6 follows from Lemmas 2.11 and 2.12.

3 Sub-Logarithmic Distance-2 Coloring

In this section, we extend the algorithm of Sec. 2 and combine it with the *graph shattering* technique [11, 12], which has been used extensively in recent years to get sub-logarithmic-time distributed algorithms for a large number of graph problems (mostly in the LOCAL model). By using this technique in our setting, we prove the following theorem.

Theorem 1.2 (restated). *There is a randomized CONGEST algorithm that d2-colors a graph with Δ^2+1 colors in $O(\log\Delta)+ND_2(\log n)\cdot \operatorname{poly}\log\log n$ rounds, with high probability.*

Here $ND_2(\log n)$ is the sum of $d\cdot c\cdot x$ and the time to that one needs to compute a distance-2 CONGEST-routable network decomposition (with weak cluster diameter d, c color classes and routing parameter x) on subgraphs of size $\operatorname{poly}\log n$ with node identifiers from a space of size $\operatorname{poly} n$ (see Definition 3.3 for the formal definition of such a decomposition).

▶ Remark 3.1. The current state of the art for $ND_2(\log n)$ is $2^{O(\sqrt{\log\log n})}$ [22]. However, the complexity for distance-1 network decompositions that can deal with a large identifier space was improved subsequent to the submission of this manuscript to $\operatorname{poly}\log\log n$ rounds [20]. Before the publication of [20] the complexity in Theorem 1.2 for distance-2 coloring matched the state of the art for distance-1 $(\Delta+1)$-coloring [19]. As the achievements of [20] improve the complexity for distance-1 coloring from $O(\log\Delta)+2^{O(\sqrt{\log\log n})}$ to $O(\log\Delta)+\operatorname{poly}\log\log n$ there currently is a gap between the complexities of distance-1 and distance-2 coloring. If [20] (or an alternative approach) extends to distance-2 decompositions, and such an extension is very likely, it will match again. In the remaining part of the writeup we use the best known upper bound of $ND_2(\log n)=2^{O(\sqrt{\log\log n})}$.

From a very high-level point of view, the rough idea of graph shattering applied to our problem is as follows. The algorithm of Sec. 2 consists of $O(\log n)$ individual $O(1)$-round steps, where in each step, each live node gets colored with constant probability. Thus, very roughly, if we just run the algorithm for $O(\log\Delta)$ steps, each node remains uncolored with probability at most $1/\operatorname{poly}(\Delta)$. Further, if nodes succeeded sufficiently independently, after $O(\log\Delta)$ rounds, each node would only have $O(\log n)$ uncolored neighbors. By combining these two properties, one can hope that after $O(\log\log n)$ more rounds, all the remaining live nodes induce components (in G^2) of size at most $\operatorname{polylog} n$. By adapting techniques developed in [11] to our G^2-coloring algorithm, we will show that this indeed (almost) is the case. We call this part of the algorithm, where we reduce the original problem to a problem on components of $\operatorname{polylog} n$ size, the *preshattering phase* of our algorithm.

The remaining problem that we need to solve on the components of size $\operatorname{polylog} n$ is a list coloring problem. Because these problems for each component are on much smaller graphs, they can be solved efficiently by using the best known deterministic algorithm. For the specific setting, where we have small components, but each node still has an ID from the original large ID space, the best known deterministic CONGEST algorithm (that can tolerate such a large ID space and works for G^2) can be obtained by combining a network decomposition

algorithm of Ghaffari and Portmann [22] with a recent deterministic CONGEST coloring algorithm of Bamberger, Kuhn, and Maus [4]. It requires $2^{O(\sqrt{\log N})} = 2^{O(\sqrt{\log\log n})}$ time, where $N = \text{polylog}\, n$ is the maximum component size. We call this second phase of solving the remaining list coloring instances on the components the *postshattering phase*.

While the general outline of the algorithm is relatively standard and largely follows the ideas of the distance-1 coloring algorithm for the LOCAL model in [11], there are various challenges that we have to cope with in order to apply the idea in the CONGEST model and to the d2-coloring problem. In [11, 19], the algorithm for the preshattering phase has every live node try a uniformly random color from its current list of available colors repeatedly, which we cannot do in the d2-coloring setting as it is not possible for a live node to learn its list of available colors. We would therefore like to show that the much more involved randomized algorithm of Sec. 2 also has the same shattering properties as the basic "choose-a-random-available-color" algorithm. Unfortunately, this is not obvious and we use a multi-stage algorithm to prove what we need. Greatly simplified, we do the following. We first show that $O(\log\Delta)$ rounds of an adaptation of the algorithm of Sec. 2 suffice to (essentially) reduce the maximum degree of the subgraph of G^2 induced by the live nodes to $O(\log n)$. At this point, it is possible for each live node to learn a sufficiently large list of available colors in $O(\log\Delta)$ rounds and we can now indeed run the basic preshattering algorithm of [11] to reduce the problem to a problem on $\text{polylog}\, n$-size components.

For the post-shattering phase, while we only have components of $\text{poly}\log n$ size, the input to the problem is still large because each node still has an ID of size $O(\log n)$ bits and because each node has a color list consisting of up to $O(\log n)$ colors from a range of size $O(\Delta^2)$. In order to have an efficient CONGEST algorithm for the problem, we have to reduce both the ID space and the color space of the remaining components. It is sufficient to obtain new node IDs that are unique up to distance $\text{poly}\log\log n$. We can obtain such IDs with $O(\log\log n)$ bits by first applying the network decomposition algorithm of [22] and then assigning unique labels in each cluster. For reducing the color space, we show that in each cluster of the network decomposition, we can efficiently (and deterministically) find a renaming of the colors such that for every node v, all colors in v's list are mapped to distinct new colors and such that the colors are from a space of size $\text{poly}\log n$. For each of the steps, the implementation in G^2 rather than in G adds some additional complications. In the following, we give a detailed overview over all the steps of our algorithm.

3.1 Preshattering: Algorithm Overview and Key Ideas

If $\log n = O(\log\Delta)$, we use the $O(\log n)$-time algorithm of Sec. 2. If $\Delta \leq \log n \cdot \text{poly}\log\log n$ we essentially simulate the preshattering algorithm of [11] for G on G^2 and combine it with our postshattering algorithm from Section 3.2. In all other cases, that is, if $\Delta = 2^{o(\log n)} \cap \tilde{\Omega}(\log n)$, we perform the following steps. They can be implemented in $O(\log\Delta) + \text{poly}\log\log n$ rounds (except for the postshattering phase which takes $2^{O(\sqrt{\log\log n})}$ rounds).

Almost Clique Decomposition

1. Compute the ACD exactly in $O(\log\Delta)$ rounds by hashing IDs to $O(\log\Delta)$ bits.
   Guarantee: Nodes know whether they are sparse/dense. Further each dense node knows an identifier of its almost clique.

Color Sparse Nodes

2. Every node (dense or sparse) tries a uniformly random color for $O(\log\Delta)$ rounds.
   Guarantee: All nodes have slack proportional to their sparsity and at most $O(\log n)$ live sparse neighbors.

3. Sparse nodes try $O(\log n)$ random colors simultaneously. In total, trying $O(\log n)$ colors requires sending/receiving $O(\log \Delta \cdot \log n)$ bits to immediate neighbors, which can be sent in $O(\log \Delta)$ rounds (by packing $O(\log n)$ bits in each message).
   Core idea: Each color you try has a constant probability to not be tried by anyone else nor adopted by a neighbor.
   Guarantee: All sparse nodes are colored, w.h.p.

Only dense (intermediate degree) nodes execute the remaining steps.

Degree Reduction of Uncolored Graph

4. Perform $O(\log \Delta)$ iterations of Reduce-Phase.
   Guarantee: Uncolored nodes either have low uncolored degree (at most $\tilde{\Delta}$), or are connected to at most $\tilde{\Delta}$ other high uncolored degree nodes, where $\tilde{\Delta} = O(\log n)$.
5. Estimate uncolored degree with $\Theta(\log n)$ precision.
   Guarantee: Uncolored nodes know whether they have low uncolored degree or not.

Let U^{lo} and U^{hi} be the sets of low and high uncolored degree vertices. All the steps afterwards first take place on U^{lo}, then on U^{hi}.

6. Try $\Theta(\log n)$ color proposals that arrive through parallel Reduce-Phases.
   Core idea: Compressing the messages communicated in a Reduce-Phase into $O(\log \Delta)$ bits. Argue a bound of $O(\log \Delta)$ on the congestion of each edge.
   Guarantee: Nodes with slack $\Omega(\log^2 n)$ become colored, w.h.p. All remaining live nodes then have sparsity $O(\log^2 n)$ (needed for Step 7 and 9).

Shattering Into Small Connected Uncolored Components

7. **Learn your list:** Expand on the method LearnPalette of [23] to have each live node learn a list of at least $d(v) + 1$ available colors from its palette. If the node has sparsity $O(\log n)$, we learn the exact list using LearnPalette as is. Otherwise, we randomly try colors not used in the almost-clique to learn enough available colors.
   Core idea: The bottleneck of the method is sending $O(\log n)$ colors over a single link, i.e., $O(\log n \log \Delta)$ bits. By compressing messages this can be done in $O(\log \Delta)$ rounds.
8. **Shattering:** Perform $O(\log \tilde{\Delta}) = O(\log \log n)$ informed color tries (OneShotColoring).
   Guarantee: Uncolored vertices induce $\text{poly}(\tilde{\Delta}) \log n = \text{poly} \log n$ sized components in G^2, and uncolored vertices know a palette that exceeds their degree.
9. **Add Steiner Nodes:** Add all vertices that link live nodes in different almost cliques. Inside each almost clique, learn all live neighbors IDs through ID-renaming, pick one intermediate node as Steiner node per pair of uncolored nodes in the almost clique.
   Guarantee: $G^2[U]$ connected components are G-connected and of size $N = \text{poly} \log n$.

Postshattering: Before the process, uncolored dense nodes U form small connected components and each node has a palette of size that exceeds its degree. Further, with the Steiner nodes connected components of $G^2[U]$ are G-connected and have $N = \text{poly} \log n$ size. This is enough to apply Lemma 3.2 in Section 3.2 and list color the remaining components in $2^{O(\sqrt{\log N})} = 2^{O(\sqrt{\log \log n})}$ rounds.

Intuition for Correctness and Implementation of the Preshattering Phase: The shattering framework with informed color trials is well established, but we apply it here in an unusual setting where the nodes do not know their palette. Instead, we argue that each live node becomes colored in each iteration with constant probability (bounded away from 0). More strongly, we show that half of the colors of its palette have good probability of becoming the node's color in each round, and this holds independent of what its neighbors do (as long as

the unlikely event of too many of them are activated does not happen). Then we show that these conditions are sufficient to leave us with two disjoint subgraphs of live nodes, both of logarithmic degree, which we handle sequentially (we first execute all steps after Step 5 including the postshattering phase for the one subgraph and then for the other subgraph). After conducting additional $O(\log \Delta)$ informed color trials, the uncolored vertices induce polylogarithmic size components. The rest of the coloring can then be completed in the post-shattering phase. The idea of producing two subgraphs of small degree already appeared in [11], but it is significantly easier to show that they cover all uncolored vertices if one can perform informed color trials.

Several further technical complications arise that do not occur for ordinary graph coloring: determining which of the two subgraphs the live node should join; adding Steiner nodes to make the components connected in G (not just in G^2); and learning enough of the palette before the post-shattering phase, even when the palette might be large. All of these steps, however, are implementable within the $O(\log \Delta)$ time bound, with techniques of modest novelty. The key idea for their efficient implementation is to compress the communication so that multiple messages fit in a single CONGEST message. Color values use $\log \Delta$ bits, but we also compress node identifiers into $O(\log \Delta)$ bits, either through hashing or renumbering within a component. This allows us to speed up communication-heavy parts: $O(\log n \cdot \log \Delta)$ bits per edge can be sent in $O(\log \Delta)$ rounds.

All of the above is for dense nodes, for which we have the structure of the almost-clique decomposition to guide us. For sparse nodes, we can use simple uniformed color guessing, first with individual colors and then with parallel color guesses, to finish them off early.

3.2 Postshattering: Algorithm Overview and Key Ideas

The high level idea is to compute a network decomposition $\mathcal{D}$ on each connected component of uncolored vertices to split the components into small diameter clusters. Afterwards, we use the deterministic $(deg + 1)$-list coloring algorithm from [4] on each cluster (iterating through the clusters in an order that is given by $\mathcal{D}$). To obtain an efficient algorithm, we need a network decomposition with two features: a) it handles distance-2 relations, and b) it handles large node identifiers (in comparison with the component sizes). The latter is not handled by the new $\operatorname{poly}\log n$ result of Rozhoň and Ghaffari [33]. Hence, we cannot currently reduce the dependence on n in the time complexity to $\operatorname{poly}\log\log n$. Instead, the construction of Portmann and Ghaffari [22] handles both of these features. The downside is the resulting time complexity of $2^{O(\sqrt{\log\log n})}$. Further, the runtime of the list-coloring algorithm in [4] depends on the size of the colorspace, and we equip our algorithm with methods to reduce the colorspace before we apply [4].

Preconditions: We are given an n-vertex graph G with maximum degree Δ and a partial $d2$-coloring $\phi : V \to [\Delta^2] \cup \{\perp\}$. Let $U = \{\phi^{-1}(\perp)\} \subseteq V$ be the uncolored vertices. Further, we are given a subset $S \subseteq V$ and $\hat{\Delta} = O(\log n)$ such that:

- Each node $u \in U$ has at most $\hat{\Delta}$ $d2$-neighbors in U.
  This immediately implies that each node in V has at most $\hat{\Delta}$ U-neighbors in G.
- $d2$-connected components of $G^2[U]$ have size $\operatorname{poly}(\hat{\Delta}) \log n = \operatorname{poly}\log n$.
- For any $u \in U$ and any of its $d2$-neighbor $u' \in U$ there exists some $s \in S$ such that s is neighbor of u and u'.
- Let $K = G[U \cup S] \setminus E(G[S])$ the subgraph of G induced by $U \cup S$ without edges between vertices in S. The connected components of K have size at most $N = \operatorname{poly}\log n$. And,
- Each vertex $u \in U$ is equipped with a list L_u of colors that are not used in its $d2$-neighborhood. The size of $|L_u| \leq L \leq O(\log n) \leq N$.

▶ **Lemma 3.2** (Postshattering). *There is a deterministic CONGEST algorithm on communication network G that, under the above assumptions, list colors the nodes in U such that $d2$-neighbors pick distinct colors. The runtime of the algorithm is $2^{O(\sqrt{\log\log n})}$ rounds.*
Lemma 3.2 uses two subroutines from previous work. First, a network decomposition algorithm that works for G^k and does not rely on a small IDspace.

▶ **Definition 3.3** (Network Decomposition, x-CONGEST-routable [3]). *A weak $\big(d(n), c(n)\big)$-*network-decomposition *of an n-node graph $G = (V, E)$ is a partition of V into clusters such that each cluster has weak diameter at most $d(n)$ and the cluster graph is properly colored with colors $1, \ldots, c(n)$. If the decomposition is equipped with a routing backbone such that one can simulate one round of communication within clusters of G^k in $k \cdot x$ rounds of communication on G (if only clusters of one color class communicate at the same time) the decomposition is called x-*CONGEST-routable*.*

As we only use network decomposition in a blackbox manner we do not detail on the additional backbone structure for routing and refer to [22] which proves the next theorem.

▶ **Theorem 3.4** (Network Decomposition of G^k, [22]). *There is a deterministic distributed algorithm that in any N-node network G, which has S-bit identifiers and supports $O(S)$-bit messages for some arbitrary S, computes an x-CONGEST-routable $(g(N); g(N))$-network decomposition of G^k in $k \cdot g(N) \cdot \log^* S$ rounds, for any k and $g(N) = x = 2^{O(\sqrt{\log N})}$.*

Second, a CONGEST algorithm that can list-color graphs efficiently if their diameter, the maximum degree and the color space size are small.

▶ **Theorem 3.5** (Diameter List Coloring, [4]). *There is a deterministic CONGEST algorithm that given a list-coloring instance $G = (V, E)$ with color space $[C]$, lists $L(v) \subseteq [C]$ for which $|L(v)| \geq \deg(v) + 1$ holds for all $v \in V$ and an initial m-coloring of G, list-colors all nodes in $O\big(D \cdot \log N \cdot \log C \cdot (\log \Delta + \log m + \log\log C)\big)$ rounds.*
The message size of the algorithm is $O(\log C + \log m + \log \Delta)$.

We will need additional reasoning to execute the algorithm of Theorem 3.5 on parts of $G^2[U]$ while the communication network is G; for that it is essential that we reduce the color space. The core steps of the postshattering phase are as follows.

1. **Network decomposition:** Compute a distance-2 network decomposition $\mathcal{D}$ of connected components in graph K using the algorithm of Theorem 3.4 (or an alternative algorithm).
2. **ID space reduction:** Assign new IDs to vertices in U that are unique within each cluster of $\mathcal{D}$. The size of the IDspace is bounded by the cluster size and by N.
3. **Colorspace reduction:** Within each cluster $\mathcal{C}$ deterministically compute a colorspace reduction $f_C : [\Delta^2] \to \text{poly}\, N$. f is a *colorspace reduction* for the cluster $\mathcal{C}$ if it injectively maps each color list L_u for $u \in \mathcal{C}$.
   Core Idea: A random hash function (from a suitable space of hash functions), in expectation, fails for few vertices of the cluster. We derandomize the process of picking such a random hash function with the method of conditional expectation, similar to [13, 16, 4].
4. **Final $(deg + 1)$-list coloring:** Iterate through the color classes of the network decomposition $\mathcal{D}$ and run the $(deg + 1)$-list coloring algorithm of Theorem 3.5 on each cluster. Care is needed when refining the lists, i.e., when deleting colors of $d2$-neighbors of previously colored clusters.

Formal proofs about the correctness and an efficient implementation of Steps 1–4 are discussed in detail in the full version [24].

References

1. Noga Alon, László Babai, and Alon Itai. A fast and simple randomized parallel algorithm for the maximal independent set problem. *J. of Algorithms*, 7(4):567–583, 1986.
2. Sepehr Assadi, Yu Chen, and Sanjeev Khanna. Sublinear algorithms for $(\Delta+1)$ vertex coloring. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 767–786, 2019.
3. Baruch Awerbuch, Andrew V. Goldberg, Michael Luby, and Serge A. Plotkin. Network decomposition and locality in distributed computation. In *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*, pages 364–369, 1989.
4. Philipp Bamberger, Fabian Kuhn, and Yannic Maus. Efficient deterministic distributed coloring with small bandwidth. In *Proc. 39th ACM Symp. on Principles of Distributed Computing (PODC)*, page 243–252, 2020.
5. Reuven Bar-Yehuda, Keren Censor-Hillel, Yannic Maus, Shreyas Pai, and Sriram V. Pemmaraju. Distributed approximation on power graphs. In *Proc. 39th ACM Symp. on Principles of Distributed Computing (PODC)*, page 501–510, 2020.
6. Leonid Barenboim. Deterministic $(\Delta + 1)$-coloring in sublinear (in Δ) time in static, dynamic and faulty networks. In *Proc. 34th Symp. on Principles of Distributed Computing (PODC)*, pages 345–354, 2015.
7. Leonid Barenboim and Michael Elkin. Deterministic distributed vertex coloring in polylogarithmic time. In *Proc. 29th Symp. on Principles of Distributed Computing (PODC)*, 2010.
8. Leonid Barenboim and Michael Elkin. *Distributed Graph Coloring: Fundamentals and Recent Developments*. Morgan & Claypool Publishers, 2013.
9. Leonid Barenboim, Michael Elkin, and Uri Goldenberg. Locally-iterative distributed $(\Delta + 1)$-coloring below Szegedy-Vishwanathan barrier, and applications to self-stabilization and to restricted-bandwidth models. In *Proc. 37th ACM Symp. on Principles of Distributed Computing (PODC)*, pages 437–446, 2018.
10. Leonid Barenboim, Michael Elkin, and Fabian Kuhn. Distributed $(\Delta + 1)$-coloring in linear (in Δ) time. *SIAM J. on Computing*, 43(1):72–95, 2015.
11. Leonid Barenboim, Michael Elkin, Seth Pettie, and Johannes Schneider. The locality of distributed symmetry breaking. In *Proc. 53th Symp. on Foundations of Computer Science (FOCS)*, 2012.
12. József Beck. An algorithmic approach to the Lovaśz local lemma. *Random Structures & Algorithms*, 2:343–365, 1991.
13. Keren Censor-Hillel, Merav Parter, and Gregory Schwartzman. Derandomizing local distributed algorithms under bandwidth restrictions. In *Proc. 31st Symp. on Distributed Computing (DISC)*, pages 11:1–11:16, 2017.
14. Yi Jun Chang, Wenzheng Li, and Seth Pettie. An optimal distributed $(\Delta + 1)$-coloring algorithm? In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 445–456, 2018.
15. Imrich Chlamtac and Shay Kutten. A spatial-reuse TDMA/FDMA for mobile multi-hop radio networks. In *Proc. 4th IEEE Int. Conf. on Computer Communications (INFOCOM)*, pages 389–394, 1985.
16. Janosch Deurer, Fabian Kuhn, and Yannic Maus. Deterministic distributed dominating set approximation in the CONGEST model. In *Proc. 38th ACM Symp. on Principles of Distributed Computing (PODC)*, pages 94–103, 2019.
17. Michael Elkin, Seth Pettie, and Hsin-Hao Su. $(2\Delta - 1)$-edge-coloring is much easier than maximal matching in the distributed setting. In *Proc. of 26th ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 355–370, 2015.
18. Pierre Fraigniaud, Magnús M. Halldórsson, and Alexandre Nolin. Distributed testing of distance-k colorings. In *Proc. 27th Coll. on Structural Information and Communication Complexity (SIROCCO)*, pages 275–290, 2020.

19 Mohsen Ghaffari. Distributed maximal independent set using small messages. In *Proc. 30th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 805–820, 2019.

20 Mohsen Ghaffari, Christoph Grunau, and Václav Rozhoň. Improved deterministic network decomposition, 2020. `arXiv:2007.08253`.

21 Mohsen Ghaffari, David G. Harris, and Fabian Kuhn. On derandomizing local distributed algorithms. In *Proc. 59th Symp. on Foundations of Computer Science (FOCS)*, pages 662–673, 2018.

22 Mohsen Ghaffari and Julian Portmann. Improved network decompositions using small messages with applications on MIS, neighborhood covers, and beyond. In *Proc. 33rd Int. Symp. on Distributed Computing (DISC)*, pages 18:1–18:16, 2019.

23 Magnús M. Halldórsson, Fabian Kuhn, and Yannic Maus. Distance-2 coloring in the CONGEST model. In *Proc. 39th ACM Symp. on Principles of Distributed Computing (PODC)*, pages 233–242, 2020.

24 Magnus M. Halldorsson, Fabian Kuhn, Yannic Maus, and Alexandre Nolin. Coloring fast without learning your neighbors' colors, 2020. `arXiv:2008.04303`.

25 David G Harris, Johannes Schneider, and Hsin-Hao Su. Distributed $(\Delta + 1)$-coloring in sublogarithmic rounds. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 465–478, 2016.

26 Ö. Johansson. Simple distributed $\Delta + 1$-coloring of graphs. *Inf. Process. Lett.*, 70(5):229–232, 1999.

27 Fabian Kuhn. Faster deterministic distributed coloring through recursive list coloring. In *Proc. 31st ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 1244–1259, 2020.

28 Fabian Kuhn and Roger Wattenhofer. On the complexity of distributed graph coloring. In *Proc. 25th ACM Symp. on Principles of Distributed Computing (PODC)*, pages 7–15, 2006.

29 Nathan Linial. Locality in distributed graph algorithms. *SIAM Journal on Computing*, 21(1):193–201, 1992.

30 Michael Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM J. on Computing*, 15:1036–1053, 1986.

31 Steffen Mecke. MAC layer and coloring. In D. Wagner and R. Wattenhofer, editors, *Algorithms for Sensor and Ad Hoc Networks*, pages 63–80, 2007.

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

33 Václav Rozhon and Mohsen Ghaffari. Polylogarithmic-time deterministic network decomposition and distributed derandomization. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 350–363, 2020.

Brief Announcement: Efficient Load-Balancing Through Distributed Token Dropping

Sebastian Brandt
ETH Zürich, Switzerland
brandts@ethz.ch

Barbara Keller
Aalto University, Finland
barbara.keller@aalto.fi

Joel Rybicki
IST Austria, Klosterneuburg, Austria
joel.rybicki@ist.ac.at

Jukka Suomela
Aalto University, Finland
jukka.suomela@aalto.fi

Jara Uitto
Aalto University, Finland
jara.uitto@aalto.fi

Abstract

We introduce a new graph problem, the *token dropping game*, and we show how to solve it efficiently in a distributed setting. We use the token dropping game as a tool to design an efficient distributed algorithm for the *stable orientation* problem, which is a special case of the more general *locally optimal semi-matching* problem. The prior work by Czygrinow et al. (DISC 2012) finds a locally optimal semi-matching in $O(\Delta^5)$ rounds in graphs of maximum degree Δ, which directly implies an algorithm with the same runtime for stable orientations. We improve the runtime to $O(\Delta^4)$ for stable orientations and prove a lower bound of $\Omega(\Delta)$ rounds.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases distributed algorithms, graph problems, semi-matching

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.40

Related Version The full version of the paper is available at https://arxiv.org/abs/2005.07761.

1 Introduction

In this work, we study efficient distributed algorithms for assignment problems. The task is to assign each customer to one adjacent server, and the customers prefer servers with a low load, i.e., few other customers. We are interested in finding a *stable assignment*, that is, an assignment in which no customer has an incentive to unilaterally switch servers. The stable assignment problem that we study here is also known as a *locally optimal semi-matching*, and it was studied in the distributed setting by Czygrinow et al. [4].

Stable Orientations. We start by studying a restricted version of the problem, *stable orientation*, which is the special case in which all customers can choose between two possible servers – we will then see how the same ideas generalize also to the stable assignment problem.

We model this setting as the following graph problem: The task is to orient all edges. We say that an oriented edge $e = (u, v)$ is *happy* if $\text{indegree}(v) \leq \text{indegree}(u) + 1$, that is, turning the orientation of e from (u, v) to (v, u) would not lower the indegree of the head of edge e. An orientation is *stable* if all edges are happy, i.e., no customer has an incentive to change his or her server.

© Sebastian Brandt, Barbara Keller, Joel Rybicki, Jukka Suomela, and Jara Uitto;
licensed under Creative Commons License CC-BY

The intuition here is that if an edge $e = \{u, v\}$ is oriented from u to v, then customer e is using server v. The *load* of a server is the total number of customers using it, i.e., its indegree. Customers would like to use servers with a low load to maximize the quality of service. From this viewpoint, a stable orientation corresponds to a game-theoretic equilibrium.

Model of Computing. We study the stable orientation problem and its generalizations in the standard LOCAL [7] model of distributed computing.

State of the Art. By prior work [4], it is known that the problem can be solved in $O(\Delta^5)$ communication rounds in graphs of maximum degree Δ; note that the running time is independent of the number of nodes.

2 Contributions

The main technical contribution is that we improve the distributed round complexity of stable orientations from $O(\Delta^5)$ to $O(\Delta^4)$. We also prove a lower bound of $\Omega(\Delta)$ for any algorithm that finds a stable orientation.

New techniques. On a high level, we use the following new techniques to save time in comparison with the prior algorithm:

- In the prior work, one starts with an arbitrary orientation. This potentially creates a large amount of unhappiness and resolving it takes a lot of time.
- In our work, we iteratively orient edges more carefully, so that there is always at most one unit of excess load per node.
- We show that the carefully oriented edges have a nice structure and model this structure by a new problem called the *token dropping game*. We show how to solve this problem efficiently, that is, in $O(\Delta^3)$ rounds.
- We play token dropping with the excess load in order to resolve unhappiness. After $O(\Delta)$ such iterations, all edges are happily oriented.

Token Dropping Game. The key new idea that we use to solve the orientation problem is to introduce a new graph problem that we call the *token dropping game*. The input consists of a graph in which the nodes are organized in *layers*, numbered from 0 to L. Some of the nodes hold a token; a node can hold at most one token. The rules are simple:

> Any token can move downwards from layer ℓ to layer $\ell - 1$ along any edge to any node that does not currently hold a token. Each edge can be used at most once.

Put otherwise, once an edge has been used to move a token, it is deleted. The task is to find some possible sequence of token movements such that we reach a configuration in which no token can be moved anymore, i.e., the only goal of this single-player game is to get stuck.

Our algorithm for this problem follows a simple proposal strategy. In every round, a node that has no token, but is adjacent to a token in a higher layer, send a proposal to the node with a token. A node that has a token accepts one proposal, and passes its token to the proposing node. The main technical challenge is to show that this process terminates quickly.

Using Token Dropping to Find Stable Orientations. We show that the token dropping problem can be solved in $O(L \cdot \Delta^2)$ rounds with a distributed algorithm; we also prove a lower bound of $\Omega(L + \Delta)$ rounds. We show that any algorithm that solves token dropping in $T(L, \Delta)$ rounds can be used to find a stable orientation in $O(\Delta \cdot T(\Delta, \Delta))$ rounds. Plugging in our algorithm for token dropping, we obtain an algorithm for finding a stable orientation in $O(\Delta^4)$ rounds, a factor-Δ improvement over the previous algorithm by [4]. In the full version we also investigate generalizations and relaxations of stable orientations problem.

Discussion. This work is part of the ongoing effort of understanding the distributed computational complexity of *locally verifiable problems*. In brief, these are problems in which a solution is globally correct if it looks good in all constant-radius neighborhoods. Stable orientations are by definition locally verifiable: if all edges are happy, the orientation is stable, and the happiness of an edge only depends on the other edges adjacent to it.

Typically, the complexity of locally verifiable problems is studied as a function of two parameters, the number of nodes n and the maximum degree Δ. In essence, these capture two complementary notions of scalability: how does the complexity of finding a solution increase when the input graph gets larger vs. when the input graph gets denser.

To study dependency on Δ, it is helpful to identify natural examples of graph problems that can be solved in $T(\Delta)$ rounds for some function T, *independently of* n. Key examples of problems that can be solved in $T(\Delta)$ rounds include maximal matching on bipartite graphs [2,6], maximal fractional matching [1,5], and weak coloring in odd-degree graphs [3,8]. However, all of these problems have a complexity at most linear in Δ. Stable orientation is perhaps one of the simplest locally verifiable graph problems that is known to be solvable in $T(\Delta)$ rounds, but for which the current upper bound is *superlinear* in Δ.

In this work we take the first steps towards deriving tight bounds on the round complexity of the stable orientation problem, with the long-term goal of showing that it indeed requires $\omega(\Delta)$ rounds.

References

1. Matti Åstrand and Jukka Suomela. Fast distributed approximation algorithms for vertex cover and set cover in anonymous networks. In *SPAA*, 2010. doi:10.1145/1810479.1810533.
2. Alkida Balliu, Sebastian Brandt, Juho Hirvonen, Dennis Olivetti, Mikaël Rabie, and Jukka Suomela. Lower Bounds for Maximal Matchings and Maximal Independent Sets. In *FOCS*, 2019. doi:10.1109/FOCS.2019.00037.
3. Sebastian Brandt. An Automatic Speedup Theorem for Distributed Problems. In *PODC*, 2019. doi:10.1145/3293611.3331611.
4. Andrzej Czygrinow, Michal Hanćkowiak, Edyta Szymańska, and Wojciech Wawrzyniak. Distributed 2-Approximation Algorithm for the Semi-matching Problem. In *DISC*, 2012. doi:10.1007/978-3-642-33651-5_15.
5. Mika Göös, Juho Hirvonen, and Jukka Suomela. Linear-in-Δ lower bounds in the LOCAL model. *Distributed Computing*, 30(5), 2017. doi:10.1007/s00446-015-0245-8.
6. Michal Hanckowiak, Michal Karonski, and Alessandro Panconesi. On the Distributed Complexity of Computing Maximal Matchings. In *SODA*, 1998.
7. Nathan Linial. Locality in Distributed Graph Algorithms. *SIAM J. Comput.*, 21(1), 1992. doi:10.1137/0221015.
8. Moni Naor and Larry Stockmeyer. What Can be Computed Locally? *SIAM J. Comput.*, 24(6), 1995. doi:10.1137/S0097539793254571.

Brief Announcement: Distributed Graph Problems Through an Automata-Theoretic Lens

Yi-Jun Chang
ETH Zürich, Switzerland
yi-jun.chang@eth-its.ethz.ch

Jan Studený
Aalto University, Finland
jan.studeny@aalto.fi

Jukka Suomela
Aalto University, Finlad
jukka.suomela@aalto.fi

Abstract

We study the following algorithm synthesis question: given the description of a locally checkable graph problem Π for paths or cycles, determine in which instances Π is *solvable*, determine what is the *locality* of Π, and construct an asymptotically optimal *distributed algorithm* for solving Π (in the usual LOCAL model of distributed computing). To answer such questions, we represent Π as a nondeterministic finite automaton $\mathcal{M}$ over a unary alphabet, and identify polynomial-time-computable properties of automaton $\mathcal{M}$ that capture the locality and solvability of problem Π.

2012 ACM Subject Classification Theory of computation → Formal languages and automata theory; Theory of computation → Models of computation; Theory of computation → Distributed algorithms

Keywords and phrases Algorithm synthesis, locally checkable labeling problems, LOCAL model, locality, distributed computational complexity, nondeterministic finite automata

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.41

Related Version A full version of the paper is available at `https://arxiv.org/abs/2002.07659`.

1 Introduction

Key Questions: Solvability and Locality. When we encounter a new graph problem Π that we would like to solve in a distributed or parallel setting, there are three basic questions we would like to answer:

1. Is Π always *solvable*, or at least solvable in all but finitely many counterexamples?
2. Is Π solvable *locally*: if I am a node in the middle of a large graph, can I choose my own part of the solution based on the information in my own local neighborhood?
3. If Π is locally solvable, how do we find an efficient *algorithm* for solving it?

Locality is a powerful property: in many setting a problem can be solved efficiently if and only if it is highly local. For the sake of concreteness, in this work we will discuss deterministic distributed algorithms in the usual LOCAL model of computing, in which efficient solvability is equivalent to locality, but we emphasize that our work has direct implications also in other models of distributed computing (e.g. CONGEST) and also in models of parallel computing (e.g. PRAM and MPC), especially for the upper bounds.

Focus and Prior Work. We will focus on LCL problems. These are graph problems in which solutions are labelings of nodes and/or edges that can be *verified locally*: if a solution looks feasible in all constant-radius neighborhoods, then it is also globally feasible [4]. Simple examples of LCL problem are finding a proper 3-coloring and finding maximal independent set of a given graph.

© Yi-Jun Chang, Jan Studený, and Jukka Suomela;
licensed under Creative Commons License CC-BY
34th International Symposium on Distributed Computing (DISC 2020).
Editor: Hagit Attiya; Article No. 41; pp. 41:1–41:3
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Given the description of an LCL problem, ideally we would like to understand its solvability and locality *in a fully automatic fashion*, and *synthesize* efficient distributed and parallel algorithms for the problem, in the broadest possible setting. Unfortunately, in the case of general graphs this is an undecidable problem [3, 4], and even if we only consider labeled paths and cycles, the task is known to be at least PSPACE-hard [1]. On the other hand, the distributed complexity of LCLs on *unlabeled directed cycles* (consistently oriented cycles) has a simple graph-theoretic characterization [3]. In this work we seek to find the broadest possible setting in which solvability and locality can be decided efficiently.

Contributions. We show how to *automatically answer questions related to both solvability and locality* of any given LCL problem both in directed cycles and paths and in undirected cycles and paths. We show that all such questions are not only decidable but they are in NP or co-NP, and almost all such questions are in P, with the exception of a couple of specific questions that are NP-complete or co-NP-complete. We give a uniform automata-theoretic formalism that enables us to study such questions, leveraging prior work on automata theory.

2 Automata Representation

Node-Edge-Checkable Formalism. There are many equivalent ways to represent LCL problems (e.g., listing all feasible constant-radius neighborhoods [4]), but the following formalism [2] is convenient for us: we interpret each edge as a pair of *ports*, and the task is to label the ports, subject to constraints on nodes and edges. Note that one can easily use port labels and node and edge constraints to encode e.g. vertex coloring and edge coloring, and indeed the formalism is expressive enough to encode *any* LCL problem on paths and cycles, modulo local preprocessing and postprocessing.

Formally, an LCL problem Π in the node-edge-checkable formalism is a tuple $\Pi = (\Gamma, \mathcal{C}_{\mathsf{edge}}, \mathcal{C}_{\mathsf{node}}, \mathcal{C}_{\mathsf{start}}, \mathcal{C}_{\mathsf{end}})$ consisting of a finite set Γ of output labels, an edge constraint $\mathcal{C}_{\mathsf{edge}} \subseteq \Gamma \times \Gamma$ (that captures feasible labelings of edges), a node constraint $\mathcal{C}_{\mathsf{node}} \subseteq \Gamma \times \Gamma$ (that captures feasible labelings of the internal nodes of cycles and paths), and start and end constraints $\mathcal{C}_{\mathsf{start}} \subseteq \Gamma$ and $\mathcal{C}_{\mathsf{end}} \subseteq \Gamma$ (that capture feasible labelings at the endpoints of paths). We say that Π is *symmetric* if $\mathcal{C}_{\mathsf{edge}}$ and $\mathcal{C}_{\mathsf{node}}$ are symmetric relations and $\mathcal{C}_{\mathsf{start}} = \mathcal{C}_{\mathsf{end}}$; such a problem is well-defined also for undirected paths and cycles.

Automata Representation. We will represent a node-edge-checkable problem Π as a *nondeterministic finite automaton* $\mathcal{M}_\Pi$ over unary alphabet $\Sigma = \{o\}$. We identify the states of an automaton with the edge constraints and the transitions with the node constraints. More precisely, the set of states is $\mathcal{C}_{\mathsf{edge}}$, there is a transition from state ab to state cd whenever $bc \in \mathcal{C}_{\mathsf{node}}$, state $ab \in \mathcal{C}_{\mathsf{edge}}$ is a starting state whenever $a \in \mathcal{C}_{\mathsf{start}}$, and state $ab \in \mathcal{C}_{\mathsf{edge}}$ is an accepting state whenever $b \in \mathcal{C}_{\mathsf{end}}$. Note that there can be multiple starting states; the automaton can choose the starting state nondeterministically.

Classification of States. We classify the states as follows; we say that state $ab \in \mathcal{C}_{\mathsf{edge}}$ is:

- *Repeatable* if there is a walk $ab \rightsquigarrow ab$ in $\mathcal{M}_\Pi$.
- *Flexible* if for all sufficiently large k there is a walk $ab \rightsquigarrow ab$ of length exactly k in $\mathcal{M}_\Pi$.
- *Mirror-flexible* if for all sufficiently large k there are walks $ab \rightsquigarrow ab$, $ab \rightsquigarrow ba$, $ba \rightsquigarrow ab$, and $ba \rightsquigarrow ba$ of length exactly k in $\mathcal{M}_\Pi$.
- A *loop* if there is a state transition $ab \to ab$ in $\mathcal{M}_\Pi$.
- A *mirror-flexible loop* if ab is both a mirror-flexible state and a loop.

Table 1 Classification of LCL problems in cycles and paths into types A–K, and the implied characteristics (solvability and locality) for each type.

Type definition:	**A**	**B**	**C/D**	**E**	**F/G**	**H/I**	**J/K**
· symmetric problem	yes	yes	yes/no	yes	yes/no	yes/no	yes/no
· repeatable state	yes	yes	yes	yes	yes	yes	no
· flexible state	yes	yes	yes	yes	yes	no	no
· loop	yes	yes	yes	no	no	no	no
· mirror-flexible state	yes	yes	no	yes	no	no	no
· mirror-flexible loop	yes	no	no	no	no	no	no
Number of instances:							
· solvable cycles	∞	∞	∞	∞	∞	∞	0
· solvable paths	∞	∞	∞	∞	∞	∞	$< \infty$
· unsolvable cycles	0	0	0	$< \infty$	$< \infty$	∞	∞
· unsolvable paths	$< \infty$	$< \infty$	$< \infty$	$< \infty$	$< \infty$	hard	∞
Locality:							
· directed cycles	$O(1)$	$O(1)$	$O(1)$	$\Theta(\log^* n)$	$\Theta(\log^* n)$	$\Theta(n)$	—
· directed paths	$O(1)$	$O(1)$	$O(1)$	$\Theta(\log^* n)$	$\Theta(\log^* n)$	$\Theta(n)$	$O(1)$
· undirected cycles	$O(1)$	$\Theta(\log^* n)$	$\Theta(n)^\dagger$	$\Theta(\log^* n)$	$\Theta(n)^\dagger$	$\Theta(n)^\dagger$	—
· undirected paths	$O(1)$	$\Theta(\log^* n)$	$\Theta(n)^\dagger$	$\Theta(\log^* n)$	$\Theta(n)^\dagger$	$\Theta(n)^\dagger$	$O(1)^\dagger$

† = assuming a symmetric problem (otherwise it is not well-defined on undirected cycles and paths)

3 Results

Our main result is the classification presented in Table 1. Given any LCL problem Π, we can first construct the automaton $\mathcal{M}_\Pi$, and determine if the problem is symmetric and if it contains loops or repeatable, flexible, or mirror-flexible states. Based on these properties, we can then use the table to classify the problem into types A–K. In the full version of this work, we show that we can *determine the type of any given problem in polynomial time.*

Now once we know the type of the problem, we can use Table 1 to directly answer all questions related to solvability and locality of Π in both undirected and directed cycles and paths, with only one exception: the only case which cannot be efficiently answered is deciding the number of unsolvable instances for problems of types H–I; we show that this question is NP-complete. In the full version, we prove that the classification is correct, and we show also how our work connects to prior work on the existence of *synchronizing words* for nondeterministic automata.

References

1. Alkida Balliu, Sebastian Brandt, Yi-Jun Chang, Dennis Olivetti, Mikaël Rabie, and Jukka Suomela. The Distributed Complexity of Locally Checkable Problems on Paths is Decidable. In *Proc. PODC*, 2019. `doi:10.1145/3293611.3331606`.
2. Alkida Balliu, Sebastian Brandt, Yuval Efron, Juho Hirvonen, Yannic Maus, Dennis Olivetti, and Jukka Suomela. Classification of distributed binary labeling problems. In *Proc. DISC*, 2020.
3. Sebastian Brandt, Juho Hirvonen, Janne H Korhonen, Tuomo Lempiäinen, Patric R J Östergård, Christopher Purcell, Joel Rybicki, Jukka Suomela, and Przemysław Uznański. LCL problems on grids. In *Proc. PODC*, 2017. `doi:10.1145/3087801.3087833`.
4. 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: Phase Transitions of the k-Majority Dynamics in a Biased Communication Model

Emilio Cruciani
Inria, I3S Lab, Université Côte d'Azur, CNRS, Valbonne, France
https://sites.google.com/view/emiliocruciani
emilio.cruciani@inria.fr

Hlafo Alfie Mimun
Department of Economics and Finance, LUISS, Roma, Italy
hmimun@luiss.it

Matteo Quattropani
Department of Economics and Finance, LUISS, Roma, Italy
https://sites.google.com/view/matteo-quattropani
mquattropani@luiss.it

Sara Rizzo
Gran Sasso Science Institute, L'Aquila, Italy
https://sites.google.com/view/sararizzo
sara.rizzo@gssi.it

Abstract

We analyze the binary-state (either $\mathcal{R}$ or $\mathcal{B}$) k-MAJORITY dynamics in a biased communication model where nodes have some fixed probability p, independent of the dynamics, of being seen in state $\mathcal{B}$ by their neighbors. In this setting we study how p, as well as the initial unbalance between the two states, impact on the speed of convergence of the process, identifying sharp phase transitions.

2012 ACM Subject Classification Theory of computation → Random walks and Markov chains; Theory of computation → Distributed algorithms; Mathematics of computing → Probabilistic algorithms; Mathematics of computing → Markov processes

Keywords and phrases Biased Communication, Consensus, Majority Dynamics, Markov Chains, Metastability

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.42

Related Version https://arxiv.org/abs/2007.15306.

1 Introduction

Designing distributed algorithms that let the nodes of a graph reach a *consensus*, i.e., a configuration of states where all the nodes agree on the same state, is a fundamental problem in distributed computing and multi-agent systems. Consensus algorithms are used in protocols for other tasks, such as leader election and atomic broadcast, and in real-world applications such as clock synchronization tasks and blockchains.

Recently there has been a growing interest in the analysis of simple local *dynamics* as distributed algorithms for the consensus problem [2, 5], inspired by simple mechanisms studied in statistical mechanics for interacting particle systems. In this scenario, nodes are *anonymous* (i.e., they do not have distinct IDs) and they have a state that evolves over time according to some common local interaction with their neighbors. Many dynamics have been investigated in such a setting, for example VOTER, where nodes copy the state of a random neighbor, and 3-MAJORITY, where nodes sample 3 neighbors with replacement and update

© Emilio Cruciani, Hlafo Alfie Mimun, Matteo Quattropani, and Sara Rizzo;
licensed under Creative Commons License CC-BY

their states to the most frequent among the samples. The time needed by such dynamics to reach a consensus is very different: for example, on the complete graph, VOTER needs $\Omega(n)$ rounds, while 3-MAJORITY converges in $\mathcal{O}(\log n)$ [2, 5].

2 Communication Model

We focus on the scenario where each of the n nodes of the underlying communication graph G has a *binary state* (either $\mathcal{R}$ or $\mathcal{B}$) and the communication among the nodes proceeds in *synchronous* rounds. In this setting we analyze the k-MAJORITY dynamics, where nodes sample k neighbors from their neighborhood uniformly at random and with replacement, and then update their state to the majority of the sample; ties are broken uniformly at random.

Differently from most of the previous works, we consider a communication model which is *biased*, w.l.o.g., toward state $\mathcal{B}$, i.e., whenever nodes sample a neighbor they see state $\mathcal{B}$ with some probability p, regardless of the state of the sampled node, and its true state with probability $1-p$. This biased communication model has been introduced in [3], where it is used as a tool for the analysis of the 2-CHOICES dynamics on core-periphery networks.

3 Results

Most of the previous works rely on strong topological assumptions, e.g., considering complete graphs or expanders, to prove upper bounds on the consensus time of the dynamics. We move a step forward in this direction by removing all assumptions that depend on the topology of the graph; indeed we only require the underlying communication graph to be *sufficiently dense*, i.e., with minimum degree $\omega(\log n)$. Such a milder assumption, though, comes at the cost of a weaker notion of consensus, that we call *almost-consensus*.

▶ **Definition 1.** *A process reaches an $\mathcal{B}$-almost-consensus whenever a fraction $1-o(1)$ of the volume of the graph is in state $\mathcal{B}$.*

Pushing our techniques to their limit, we could prove a consensus on state $\mathcal{B}$ in $\mathcal{O}(\log n)$ rounds. We would still require no topological assumptions, but just a stronger condition on the minimum degree (from $\omega(\log n)$ to $\Omega(n)$), thus dramatically restricting the class of graphs taken into account to extremely dense ones. Note that, in the biased communication model we consider, the Markov Chain that models the process has a single absorbing state where all nodes are in state $\mathcal{B}$. For this reason we first consider an initial configuration where all the nodes are in state $\mathcal{R}$ and study the time needed by the process to reach a $\mathcal{B}$-almost-consensus. Trivially, if $p=0$ the process remains stuck in its initial configuration, while if $p=1$ the process reaches the absorbing state in one single round. More in general, it is intuitive that the process will converge slowly to the absorbing state if p is small and quickly if p is large. With such an intuition in mind, we prove a first *phase transition* phenomenon.

▶ **Theorem 2.** *Consider the k-MAJORITY dynamics in the communication model with bias p toward state $\mathcal{B}$, in any graph G with minimum degree $\omega(\log n)$, and where initially all nodes are in state $\mathcal{R}$. For every $k \geq 3$ there exists a constant $p_k^\star \in [\frac{1}{9}, \frac{1}{2}]$ such that:*

- Slow convergence: *If $p < p_k^\star$, a $\mathcal{B}$-almost-consensus is reached in $n^{\omega(1)}$ rounds, a.a.s.*[1]
- Fast convergence: *If $p > p_k^\star$, a $\mathcal{B}$-almost-consensus is reached in $\mathcal{O}(1)$ rounds, a.a.s.*

[1] We say that an event $\mathcal{E}_n$ happens *asymptotically almost surely* (in short, *a.a.s.*) if $\Pr(\mathcal{E}_n) = 1 - o(1)$.

The proof looks at the expected evolution of the fraction of neighbors in state $\mathcal{B}$ of every node and then applies concentration of probability arguments, akin to what has been done in [3]. However, compared to [3], we have a more comprehensive scenario (k-MAJORITY for any k vs. 2-CHOICES) with substantially more precise results.

Another important innovation with respect to [3] is the presence of a second phase transition on the initial unbalance between the states. Clearly in the *fast convergence regime*, i.e., when $p > p_k^\star$, the initial configuration in which none of the nodes is in state $\mathcal{B}$ is the hardest one for the process to reach a $\mathcal{B}$-almost-consensus. However, the scenario changes in the *slow convergence regime*, where a different initial configuration with some of the nodes already in state $\mathcal{B}$ could speed up the process.

▶ **Theorem 3.** *Consider the k-MAJORITY dynamics in the communication model with bias $p < p_k^\star$ toward state $\mathcal{B}$, in any graph G with minimum degree $\omega(\log n)$, and where initially every node is in state $\mathcal{B}$ with probability $1 - q$, independently of the others. For every $k \geq 3$ there exists a constant $q_{p,k}^\star$ such that:*

- Fast convergence: *If $q < q_{p,k}^\star$, a $\mathcal{B}$-almost-consensus is reached in $\mathcal{O}(1)$ rounds, a.a.s.*
- Slow convergence: *If $q > q_{p,k}^\star$, a $\mathcal{B}$-almost-consensus is reached in $n^{\omega(1)}$ rounds, a.a.s.*

4 Applications

Our results show that adding a bias to k-MAJORITY affects the dynamics in a non-trivial way. In particular, the arise of a metastable phase makes the framework suitable to design distributed algorithms to recover planted partitions in networks [1, 4, 6]. Consider a graph $G = ((V_1, V_2), E)$ with two clusters and such that, for every pair of nodes in each cluster, their fraction of neighborhood toward the other cluster is equal to some constant z, as in [1]. Let G run the k-MAJORITY and suppose that the two clusters reach a local almost-consensus on different states, say nodes in V_1 agrees on $\mathcal{R}$ and nodes in V_2 on $\mathcal{B}$. The evolution of such a process can be described by the biased k-MAJORITY dynamics. In fact, note that the k-MAJORITY dynamics with bias $p = z$ toward $\mathcal{B}$ performed by the nodes of the subgraph induced by V_1 describes the local evolution of the nodes in V_1, when considering a worst-case scenario in which the nodes in V_2 never change color. If G is such that $z = p < p_k^\star$, Theorem 2 implies that V_1 remains in an almost-consensus configuration, a.a.s. Since the same reasoning can be done for V_2, it follows that the graph would stay in a configuration that highlights its clustered structure for $n^{\omega(1)}$ rounds, hence making k-MAJORITY a suitable dynamics for the design a distributed community detection protocol.

References

1. Luca Becchetti, Emilio Cruciani, Francesco Pasquale, and Sara Rizzo. Step-by-step community detection in volume-regular graphs. In *ISAAC 2019*, pages 20:1–20:23, 2019.
2. Colin Cooper, Robert Elsässer, Hirotaka Ono, and Tomasz Radzik. Coalescing random walks and voting on connected graphs. *SIAM J. Discrete Math.*, 27(4):1748–1758, 2013.
3. Emilio Cruciani, Emanuele Natale, André Nusser, and Giacomo Scornavacca. Phase transition of the 2-choices dynamics on core-periphery networks. In *AAMAS 2018*, pages 777–785, 2018.
4. Emilio Cruciani, Emanuele Natale, and Giacomo Scornavacca. Distributed community detection via metastability of the 2-choices dynamics. In *AAAI 2019*, pages 6046–6053, 2019.
5. Mohsen Ghaffari and Johannes Lengler. Nearly-tight analysis for 2-choice and 3-majority consensus dynamics. In *PODC 2018*, pages 305–313, 2018.
6. Nobutaka Shimizu and Takeharu Shiraga. Phase transitions of best-of-two and best-of-three on stochastic block models. In *DISC 2019*, pages 32:1–32:17, 2019.

Brief Announcement: Distributed Quantum Proofs for Replicated Data

Pierre Fraigniaud
IRIF, CNRS and Université de Paris, France

François Le Gall
Graduate School of Mathematics, Nagoya University, Japan

Harumichi Nishimura
Graduate School of Informatics, Nagoya University, Japan

Ami Paz
Faculty of Computer Science, Universität Wien, Austria

Abstract

This paper tackles the issue of *checking* that all copies of a large data set replicated at several nodes of a network are identical. The fact that the replicas may be located at distant nodes prevents the system from verifying their equality locally, i.e., by having each node consult only nodes in its vicinity. On the other hand, it remains possible to assign *certificates* to the nodes, so that verifying the consistency of the replicas can be achieved locally. However, we show that, as the replicated data is large, classical certification mechanisms, including distributed Merlin-Arthur protocols, cannot guarantee good completeness and soundness simultaneously, unless they use very large certificates. The main result of this paper is a distributed *quantum* Merlin-Arthur protocol enabling the nodes to collectively check the consistency of the replicas, based on small certificates, and in a single round of message exchange between neighbors, with short messages. In particular, the certificate-size is logarithmic in the size of the data set, which gives an exponential advantage over classical certification mechanisms.

2012 ACM Subject Classification Theory of computation → Quantum communication complexity; Theory of computation → Distributed computing models

Keywords and phrases Quantum Computing, Distributed Network Computing, Algorithmic Aspects of Networks

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.43

Related Version A full version of the paper is available at `https://arxiv.org/abs/2002.10018`.

Funding *Pierre Fraigniaud*: Additional support from the ANR project *DESCARTES*.
François Le Gall: JSPS KAKENHI grants Nos. JP16H01705, JP19H04066, JP20H04139 and JP20H00579 and MEXT Quantum Leap Flagship Program Grant Number JPMXS0120319794.
Harumichi Nishimura: JSPS KAKENHI grants Nos. JP16H01705, JP19H04066 and MEXT Quantum Leap Flagship Program Grant Number JPMXS0120319794.
Ami Paz: Supported by the Austrian Science Fund (FWF): P 33775-N, Fast Algorithms for a Reactive Network Layer.

1 Introduction

In distributed systems, the presence of faults potentially corrupting the individual states of the nodes creates a need to regularly check whether the system is in a global state that is legal with respect to its specification. A basic example is a system storing data, and using replicas in order to support crash failures. In this case, the application managing the data is in charge of regularly checking that the several replicas of the same data, stored at different nodes scattered in the network, are all identical. Another example is an application

© Pierre Fraigniaud, François Le Gall, Harumichi Nishimura, and Ami Paz;
licensed under Creative Commons License CC-BY

maintaining a tree spanning the nodes of a network, e.g., for multicast communication. In this case, every node stores a pointer to its parent in the tree, and the application must regularly check that the collection of pointers forms a spanning tree. This paper addresses the issue of checking the correctness of a distributed system configuration at low cost.

Several mechanisms have been designed for certifying the correctness of the global state of a system in a distributed manner. One popular mechanism is called *locally checkable proofs* [3], and it extends the seminal concept of *proof-labeling schemes* [4]. In these frameworks, the distributed application does not only construct or maintain some distributed data structure (e.g., a spanning tree), but also constructs a distributed *proof* that the data structure is correct. This proof has the form of a *certificate* assigned to each node (the certificates assigned to different nodes do not need to be the same). For collectively checking the legality of the current global system state, the nodes exchange their certificates with their neighbors in the network. Then, based on its own individual state, its certificate, and the certificates of its neighbors, every node accepts or rejects, according to the following specification. If the global state is legal, and if the certificates are assigned properly by the application, then all nodes accept. Conversely, if the global state is illegal, then at least one node rejects, *no matter which certificates are assigned to the nodes*. Such a rejecting node can raise an alarm, or launch a recovery procedure. The main aim of locally checkable proofs is to be *compact*, that is, to use certificates as small as possible, for two reasons: first, to limit the space complexity at each node, and, second, to limit the message complexity of the verification procedure involving communications between neighbors.

Unfortunately, not all boolean predicates on labeled graphs can be distributedly certified using certificates as small as for spanning tree. This is typically the case of the aforementioned scenario of a distributed data storage using replicas, for which one must certify equality. Let us for instance consider the case of two nodes Alice and Bob at the two extremities of a path, that is, the two players are separated by intermediate nodes. Alice and Bob respectively store two n-bit strings x and y, and the objective is to certify that $x = y$. That is, one wants to certify equality (EQ) between *distant* players. A direct reduction from the non-deterministic communication complexity of EQ shows that certifying EQ cannot be achieved with certificates smaller than $\Omega(n)$ bits.

Randomization may help circumventing the difficulty of certifying some boolean predicates on labeled graphs using small certificates. Hence, a weaker form of protocols has been considered, namely *distributed Merlin-Arthur* protocols (dMA), a.k.a. *randomized proof-labeling schemes* [2]. In this latter context, Merlin provides the nodes with a proof, just like in locally checkable proofs, and Arthur performs a *randomized* local verification at each node. Unfortunately, some predicates remain hard in this framework too. In particular, we show in the full version of our paper [1] that there is no classical dMA protocol for (distant) EQ using compact certificates. Recently, several extensions of dMA protocols were proposed, e.g., by allowing more interaction between the prover and the verifier. In this work, we add the quantum aspect, while considering only a single interaction, and only in the prescribed order: Merlin sends a proof to Arthur, and then there is no more interaction between them.

2 Our Results

We carry on the recent trend of research consisting of investigating the power of quantum resources in the context of distributed network computing (cf., e.g., see the references in the full version of our paper [1]) by designing a distributed Quantum Merlin-Arthur (dQMA) protocol for distant EQ, using compact certificates and small messages. While we use the

dQMA terminology in order to be consistent with prior work, we emphasize that the structure of the discussed protocols is rather simple: each node is given a quantum state as a certificate, the nodes exchange these states, perform a local computation, and finally accept or reject.

Our main result is the following. A collection of n-bit strings $x_1, \ldots, x_t$ are stored at t terminal nodes $u_1, \ldots, u_t$ in a network $G = (V, E)$. We denote EQ_n^t the problem of checking the equality $x_1 = \cdots = x_t$ between the t strings. Let us define the *radius* of a given instance of EQ_n^t as $r = \min_i \max_j \mathsf{dist}_G(u_i, u_j)$, where dist_G denotes the distance in the (unweighted) graph G. Our main result is the design of a dQMA protocol for EQ_n^t, using small certificate. This can be summarized by the following, informal statement.

▶ **Main Result.** *There is a distributed Quantum Merlin-Arthur (dQMA) protocol for certifying equality between t binary strings (EQ_n^t) of length n, located at a radius-r set of t terminals, in a single round of communication between neighboring nodes using certificates of size $O(tr^2 \log n)$ qubits, and messages of size $O(tr^2 \log(n + r))$ qubits.*

It is worth mentioning that, although the dependence in r and t is polynomial, the dependence in the actual size n of the instance remains logarithmic, which is our main concern. Indeed, for applications such as the aforementioned distributed data storage motivating the distant EQ_n^t problem, it is expected that both the number t of replicas, and the maximum distance between the nodes storing these replicas are of several orders of magnitude smaller than the size n of the stored replicated data.

It is also important to note that our protocol satisfies the basic requirement of *reusability*, as one aims for protocols enabling regular and frequent verifications that the data are not corrupted. Specifically, the quantum operations performed on the certificates during the local verification phase operated between neighboring nodes preserve the quantum nature of these certificates. That is, if EQ_n^t is satisfied, i.e., if all the replicas x_i's are equal, then, up to an elementary local relocation of the quantum certificates, these certificates are available for a next test. If EQ_n^t is not satisfied, i.e., if there exists a pair of replicas $x_i \neq x_j$, then the certificates do not need to be preserved as this scenario corresponds to the case where the correctness of the data structure is violated, requiring the activation of recovery procedures for fixing the bug, and reassigning certificates to the nodes.

Finally, observe that our logarithmic upper bound for dQMA protocols is in contrast to the linear lower bound that can be shown for classical dMA protocols even for $t = 2$ on a path of 4 nodes and even for the case where communication between the neighboring nodes is extended to multiple rounds (see precise statement and proof in the full version of our paper [1]). Our results thus show that quantum certification mechanism can provide an exponential advantage over classical certification mechanisms.

References

1. Pierre Fraigniaud, François Le Gall, Harumichi Nishimura, and Ami Paz. Distributed quantum proofs for replicated data. arXiv:2002.10018, 2020. `arXiv:2002.10018`.
2. Pierre Fraigniaud, Boaz Patt-Shamir, and Mor Perry. Randomized proof-labeling schemes. *Distributed Computing*, 32(3):217–234, 2019. `doi:10.1007/s00446-018-0340-8`.
3. Mika Göös and Jukka Suomela. Locally checkable proofs in distributed computing. *Theory of Computing*, 12:19:1–19:33, 2016. `doi:10.4086/toc.2016.v012a019`.
4. Amos Korman, Shay Kutten, and David Peleg. Proof labeling schemes. *Distributed Computing*, 22(4):215–233, 2010. `doi:10.1007/s00446-010-0095-3`.

Brief Announcement: Optimally-Resilient Unconditionally-Secure Asynchronous Multi-Party Computation Revisited

Ashish Choudhury
International Institute of Information Technology Bangalore, India
ashish.choudhury@iiitb.ac.in

Abstract

In this paper, we present an *optimally-resilient*, unconditionally-secure *asynchronous multi-party computation* (AMPC) protocol for n parties, tolerating a *computationally unbounded* adversary, capable of corrupting up to $t < \frac{n}{3}$ parties. Our protocol needs a communication of $\mathcal{O}(n^4)$ field elements per multiplication gate. This is to be compared with previous best AMPC protocol (Patra et al, ICITS 2009) in the same setting, which needs a communication of $\mathcal{O}(n^5)$ field elements per multiplication gate. To design our protocol, we present a simple and highly efficient *asynchronous verifiable secret-sharing* (AVSS) protocol, which is of independent interest.

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

Keywords and phrases Verifiable Secret-sharing, Secure MPC, Fault-tolerance, Byzantine faults, secret-sharing, unconditional-security, privacy

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.44

Related Version A full version of the paper is available at [4], `https://eprint.iacr.org/2020/906`.

Funding *Ashish Choudhury*: This research is an outcome of the R & D work undertaken in the project under the Visvesvaraya PhD Scheme of Ministry of Electronics & Information Technology, Government of India, being implemented by Digital India Corporation (formerly Media Lab Asia).

1 Introduction

Secure *multi-party computation* (MPC) [9, 6, 2] is a fundamental problem in secure distributed computing. Informally a MPC protocol allows a set of n mutually-distrusting parties to perform a joint computation on their inputs, while keeping their inputs as private as possible, even in the presence of an adversary Adv who can corrupt any t out of these n parties. While the MPC problem has been pre-dominantly studied in the *synchronous* communication model where the message delays are upper bounded by a public constant, the progress in the design of efficient asynchronous MPC (AMPC) protocols is rather slow. In the latter setting, the communication channels may have arbitrary but finite delays and deliver messages in any arbitrary order, with the only guarantee that all sent messages are *eventually* delivered.

In this work, we consider a setting where Adv is *computationally unbounded*. In this setting, we have two class of AMPC protocols. *Perfectly-secure* AMPC protocols give the security guarantees without any error, while *unconditionally-secure* AMPC protocols give the security guarantees with probability at least $1 - \epsilon_{\mathsf{AMPC}}$, where ϵ_{AMPC} is any given (non-zero) error parameter. While there are quite a few works which consider optimally-resilient perfectly-secure AMPC protocol, not too much attention has been paid to the design of efficient unconditionally-secure AMPC protocol with the optimal resilience of $t < \frac{n}{3}$ [3]. In this work, we make inroads in this direction, by presenting a simple and efficient unconditionally-secure AMPC protocol.

© Ashish Choudhury;
licensed under Creative Commons License CC-BY
34th International Symposium on Distributed Computing (DISC 2020).
Editor: Hagit Attiya; Article No. 44; pp. 44:1–44:3
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1.1 Our Results and Comparison with the Existing Works

In any unconditionally-secure AMPC protocol (including ours), the function to be computed is abstracted as a publicly-known ckt over some finite field $\mathbb{F}$, consisting of addition and multiplication gates over $\mathbb{F}$ and the goal is to let the parties jointly and"securely" evaluate ckt. The field $\mathbb{F}$ is typically the Galois field $\mathrm{GF}(2^\kappa)$, where κ depends upon ϵ_{AMPC}. The *communication complexity* of any AMPC protocol is dominated by the communication needed to evaluate the multiplication gates in ckt. Consequently, the focus of any generic AMPC protocol is to improve the communication required for evaluating the multiplication gates in ckt. The following table summarizes the communication complexity of the existing AMPC protocols with the optimal resilience of $t < \frac{n}{3}$ and our protocol.

Reference	Communication Complexity (in bits) for a Single Multiplication Gate
[3]	$\mathcal{O}(n^{11}\kappa^4)$
[7]	$\mathcal{O}(n^5\kappa)$
This paper	$\mathcal{O}(n^4\kappa)$

We follow the standard approach of shared circuit-evaluation, where each value during the evaluation of ckt is Shamir secret-shared [8] among the parties, with threshold t. Informally, a value s is said to be Shamir-shared with threshold t, if there exists some degree-t polynomial with s as its constant term and every party P_i holds a distinct evaluation of this polynomial as its share. In the AMPC protocol, each party P_i *verifiably* secret-shares its input for ckt. The verifiability here ensures that if the parties terminate this step, then some value is indeed Shamir secret-shared among the parties on the behalf of P_i. To verifiably secret-share its input, each party executes an instance of *asynchronous verifiable secret-sharing* (AVSS). Once the inputs of the parties are secret-shared, the parties then evaluate each gate in ckt, maintaining the following invariant: if the gate inputs are secret-shared, then the parties try to obtain a secret-sharing of the gate output. Due to the linearity of Shamir secret-sharing, maintaining the invariant for addition gates do not need any interaction among the parties. However, for maintaining the invariant for multiplication gates, the parties need to interact with each other and hence the onus is rightfully shifted to minimize this cost. For evaluating the multiplication gates, the parties actually deploy the standard Beaver's circuit-randomization technique [1]. The technique reduces the cost of evaluating a multiplication gate to that of publicly reconstructing two secret-shared values, provided the parties have access to a Shamir-shared random multiplication triple (a, b, c), where $c = a \cdot b$. The shared multiplication triples are generated in advance in a bulk in a circuit-independent pre-processing phase, using the efficient framework proposed in [5]. The framework allows to efficiently and verifiably generate Shamir-shared random multiplication triples, using any given AVSS protocol. Once all the gates in ckt are evaluated and the circuit-output is available in a secret-shared fashion, the parties publicly reconstruct this value. The privacy of the computation follows from the fact that during the shared circuit-evaluation, for each value in ckt, Adv learns at most t shares, which are independent of the actual shared value. While the AMPC protocols of [3, 7] also follow the above blue-print of shared circuit-evaluation, the difference is in the underlying AVSS protocol.

AVSS is a well-known and important primitive in secure distributed computing. On a very high level, an AVSS protocol enhances the security of Shamir secret-sharing against a *malicious* adversary (Shamir secret-sharing achieves its properties only in the *passive* adversarial model, where even the corrupt parties honestly follow protocol instructions). The existing unconditionally-secure AVSS protocols with $t < n/3$ [3, 7] need high communication. This is because there are significant number of obstacles in designing unconditionally-secure

AVSS with exactly $n = 3t + 1$ parties (which is the least value of n with $t < n/3$). The main challenge is to ensure that *all honest* parties obtain their shares of the secret. We call an AVSS protocol guaranteeing this "completeness" property as *complete* AVSS. However, in the asynchronous model, it is impossible to directly get the confirmation of the receipt of the share from each party, as corrupt parties may never respond. To get rid off this difficulty, [3] introduces a "weaker" form of AVSS which guarantees that the underlying secret is verifiably shared only among a set of $n - t$ parties and up to t parties may not have their shares. To distinguish this type of AVSS from complete AVSS, the latter category of AVSS is termed an *asynchronous complete secret-sharing* (ACSS) in [3], while the weaker version of AVSS is referred as just AVSS[1]. Given any AVSS protocol, [3] shows how to design an ACSS protocol using n instances of AVSS. An AVSS protocol with $t < n/3$ is also presented in [3]. With a communication complexity of $\Omega(n^9\kappa)$ bits, the protocol is highly expensive. This AVSS protocol when used in their ACSS protocol incurs a communication complexity of $\Omega(n^{10}\kappa)$bits. Apart from being communication expensive, the AVSS of [3] involves a lot of asynchronous primitives such as ICP, A-RS, AWSS and Two & Sum AWSS. In [7], a simplified AVSS protocol with communication complexity $\mathcal{O}(n^3\kappa)$ bits is presented, based on only few primitives, namely ICP and AWSS. This AVSS is then converted into an ACSS in the same way as [3], making the communication complexity of their ACSS $\mathcal{O}(n^4\kappa)$ bits.

In this work, we further improve upon the communication complexity of the ACSS of [7]. We first design a new AVSS protocol with a communication complexity $\mathcal{O}(n^2\kappa)$ bits. Then using the approach of [3], we obtain an ACSS protocol with communication complexity $\mathcal{O}(n^3\kappa)$ bits. Our AVSS protocol is conceptually simpler and is based on just the ICP primitive and hence easy to understand. Moreover, since we avoid the usage of AWSS in our AVSS, we get a saving of $\Theta(n)$ in the communication complexity, compared to [7] (the AVSS of [7] invokes n instances of AWSS, which is not required in our AVSS). We refer the readers to [4] for the complete details of our AVSS scheme and AMPC protocol.

References

1. D. Beaver. Efficient Multiparty Protocols Using Circuit Randomization. In *CRYPTO*, volume 576 of *LNCS*, pages 420–432. Springer, 1991.
2. M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation. In *STOC*, pages 1–10. ACM, 1988.
3. M. Ben-Or, B. Kelmer, and T. Rabin. Asynchronous Secure Computations with Optimal Resilience. In *PODC*, pages 183–192. ACM, 1994.
4. A. Choudhury. Optimally-resilient Unconditionally-secure Asynchronous Multi-party Computation Revisited. Cryptology ePrint Archive, Report 2020/906, 2020.
5. A. Choudhury and A. Patra. An Efficient Framework for Unconditionally Secure Multiparty Computation. *IEEE Trans. Information Theory*, 63(1):428–468, 2017.
6. O. Goldreich, S. Micali, and A. Wigderson. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In *STOC*, pages 218–229. ACM, 1987.
7. A. Patra, A. Choudhary, and C. Pandu Rangan. Efficient Statistical Asynchronous Verifiable Secret Sharing with Optimal Resilience. In *ICITS*, volume 5973 of *LNCS*, pages 74–92. Springer, 2009.
8. A. Shamir. How to Share a Secret. *Commun. ACM*, 22(11):612–613, 1979.
9. A. C. Yao. Protocols for Secure Computations. In *FOCS*, pages 160–164. IEEE, 1982.

[1] We stress that the weaker form of AVSS is not sufficient for the shared circuit-evaluation. This is because the set of $n - t$ share-holders might be different for different shared values.

Brief Announcement: Polygraph: Accountable Byzantine Agreement

Pierre Civit
UPMC, Paris 6, France
pierrecivit@gmail.com

Seth Gilbert
National University of Singapore, Singapore
seth.gilbert@comp.nus.edu.sg

Vincent Gramoli
The University of Sydney, Australia
EPFL, Lausanne, Switzerland
vincent.gramoli@sydney.edu.au

Abstract

In this paper, we introduce *Polygraph*, the first accountable Byzantine consensus algorithm. If among n users $f < n/3$ are malicious then it ensures consensus, otherwise it eventually detects malicious users that cause disagreement. Polygraph is appealing for blockchains as it allows to totally order blocks in a chain whenever possible, hence avoiding double spending and, otherwise, to punish at least $n/3$ malicious users when a fork occurs. This problem is more difficult than it first appears. Blockchains typically run in open networks whose delays are hard to predict, hence one cannot build upon synchronous techniques [5, 1]. One may exploit cryptographic evidence of PBFT-like consensus [2], however detecting equivocation would be insufficient. We show that it is impossible without extra logs of at least $\Omega(n)$ rounds [3]. Each round of Polygraph exchanges $O(n^2)$ messages.

2012 ACM Subject Classification Security and privacy → Distributed systems security

Keywords and phrases Fault detection, cryptography, equivocation, consensus

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.45

Funding Part of this work is funded by ARC projects DP180104030 and FT180100496.

The Accountable Byzantine Agreement problem. We consider n processes, $f < n$ are Byzantine. Let t_0 be $\lceil \frac{n}{3} \rceil - 1$. Processes are sequential and asynchronous. We assume a PKI and that the network is partially synchronous. A verification algorithm V takes as input the state of a process and returns a set G of undeniable *guilty* processes, that is, every process-id of G is tagged with an unforgeable proof of culpability. We define the Accountable Byzantine Agreement problem similarly to the Byzantine Agreement: each process begins with a binary *input* and outputs a *decision*, satisfying the three usual properties (agreement, validity, and termination), and that there exists a verification algorithm that can identify at least $t_0 + 1$ Byzantine users whenever there is disagreement.

▶ **Definition 1** (Accountable Byzantine Agreement (Acc)). *An algorithm solves Acc if each process takes an input value, possibly produces a decision, and satisfies the following properties:*

- Agreement: *If $f \leq t_0$, then every honest process decides the same value.*
- Validity: *If all processes are honest and have the same input value, then that is the only decision value.*
- Termination: *If $f \leq t_0$, every honest process eventually decides a value.*
- Accountability*: There exists a verification algorithm V such that: if two honest processes decide distinct values, then eventually for every honest process p_j, for every state s_j reached by p_j from that point onwards, the verification $V(s_j)$ outputs a set of size at least $t_0 + 1$, containing exclusively Byzantine processes.*

© Pierre Civit, Seth Gilbert, and Vincent Gramoli;
licensed under Creative Commons License CC-BY
34th International Symposium on Distributed Computing (DISC 2020).
Editor: Hagit Attiya; Article No. 45; pp. 45:1–45:3
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Polygraph. Polygraph is the first accountable Byzantine agreement protocol. It builds upon DBFT [4], an efficient consensus algorithm for blockchains. The notation broadcast(TAG,m)$\rightarrow$ $msgs$ denotes that p_i sends a message to every other process, with message type TAG, message content m and location $msgs$ to store received messages. We assume that every message is signed by the sender so the receiver can authenticate it. Finally, we write "receive k messages" to explain "receive messages from k distinct processes".

Algorithm 1 The Polygraph Protocol.

1: bin-propose(v_i):
2: $e_i = v_i$
3: $r_i = 0$
4: $\tau_i = 0$
5: $\ell_i[0] = \emptyset$
6: **repeat:**
7: $r_i \leftarrow r_i + 1$; ▷ *increment round*
8: $\tau_i \leftarrow \tau_i + 1$ ▷ *increment timer*
9: $c_i \leftarrow ((r_i - 1) \mod n) + 1$ ▷ *rotate coordinator*
▷ Phase 1:
10: bv-broadcast(EST$[r_i]$, e_i, $\ell_i[r_i - 1]$, i, bvs_i)
11: **if** $i = c_i$ **then** ▷ *coordinator rebroadcasts*
12: **wait until** $(bv_i[r_i] = \{w\})$ ▷ *bv-delivered bvs*
13: broadcast(CO$[r_i]$, w) $\rightarrow msgs_i$
14: start-timer(τ_i)
15: **wait until** $(bvs_i[r_i] \neq \emptyset \wedge$ timer expired)
▷ Phase 2:
16: **if** (CO$[r_i]$, w) $\in msgs_i$ from $p_{c_i} \wedge$
17: $w \in bvs_i[r_i]$) **then** $aux_i \leftarrow \{w\}$
18: **else** $aux_i \leftarrow bvs_i[r_i]$
19: sig_i = sign(aux_i, r_i, i) ▷ *sign the messages*
20: broadcast(ECHO$[r_i]$, $aux_i[r_i]$, sig_i) $\rightarrow msgs_i$
21: **wait until** $vals_i$ = values($msgs_i$, bvs_i, aux_i) $\neq \emptyset$
▷ Decision phase:
22: **if** $vals_i = \{v\}$ **then** ▷ *if one value, adopt it*
23: $e_i \leftarrow v$
24: **if** $v = (r_i \mod 2)$ **then** ▷ *if parity matches*
25: **if** no previous decision by p_i **then** decide(v)
26: **else**
27: $e_i \leftarrow (r_i \mod 2)$ ▷ *otherwise, adopt parity bit*
28: $\ell_i[r_i]$ = justify($vals_i$, e_i, r_i, bvs_i, $msgs_i$)

Rules:

1. Every message that is not properly signed by the sender is discarded.
2. Every message that is sent by bv-broadcast without a valid ledger after Round 1, except for messages containing value 1 in Round 2, are discarded.
3. On first discovering a ledger l that conflicts with a certificate, send ledger l to all processes.

29: bv-broadcast(MSG, val, l, i, bvs):
30: broadcast(BVAL, $\langle val, l, i\rangle$) $\rightarrow m$ ▷ *bcast message*
31: After round 2, and in round 1 if $val = 0$, discard
32: all messages received without a proper ledger.
33: **upon** receipt of (BVAL, $\langle v, \cdot, j\rangle$)
34: **if** received $(t_0 + 1)$ messages (BVAL, $\langle v, \cdot, \cdot\rangle$) and
35: (BVAL, $\langle v, \cdot, \cdot\rangle$) not yet broadcast **then**
36: Let $l \neq \emptyset$ be any ledger in these messages.
37: broadcast(BVAL, $\langle v, l, j\rangle$)
38: **if** received $(2t_0 + 1)$ times (BVAL, $\langle v, \cdot, \cdot\rangle$) **then**
39: Let $l \neq \emptyset$ be any ledger in these messages.
40: $bvs \leftarrow bvs \cup \{\langle v, l, j\rangle\}$

41: values($msgs$, b_set, aux_set): ▷ *check messages*
42: **if** $\exists S \subseteq msgs$ where the following conditions hold:
43: (i) $|S|$ contains $(n - t_0)$ distinct ECHO$[r_i]$ msgs
44: (ii) aux_set is equal to the set of values in S.
45: **then return**(aux_set)
46: **if** $\exists S \subseteq msgs$ where the following conditions hold:
47: (i) $|S|$ contains $(n - t_0)$ distinct ECHO$[r_i]$ msgs
48: (ii) Every value in S is in b_set.
49: **then return**(V = the set of values in S)
50: **else return**($\emptyset$)

51: justify($vals_i$, e_i, r_i, bvs_i, $msgs_i$): ▷ *compute ledger*
52: **if** $e_i = (r_i \mod 2)$ **then**
53: **if** $r_i > 1$ **then**
54: **return** $\ell[r_i]_i = l$ s.t. (EST$[r_i]$, $\langle v, l, \cdot\rangle$) $\in bvs_i$
55: **else return** $\ell[r_i]_i = \emptyset$
56: **else return** $\ell[r_i]_i = (n - t_0)$ signed messages
57: from $msgs_i$ containing only value e_i
58: **if** $vals_i = \{(r_i \mod 2)\} \wedge$ no previous decision
59: by p_i in previous round **then**
60: $cert_i = (n - t_0)$ signed messages from $msgs_i$
61: containing only value e_i
62: broadcast(e_i, r_i, i, $cert_i$) ▷ *broadcast certificate*

The protocol proceeds in asynchronous rounds where processes maintain an estimate. Each round proceeds in two phases, after which a possible decision is taken. In the first phase, each process bv-broadcasts its estimate using a reliable broadcast service that guarantees while $f < n/3$ that: (i) every message broadcast by $t_0 + 1$ honest processes is eventually delivered to every honest process; (ii) every message delivered to an honest process was broadcast by at least $t_0 + 1$ processes. All processes then wait until they receive at least one message, and until an increasing timer expires. A rotating coordinator for each round broadcasts its estimate with a special designation. In the second phase, if a process receives a message from the coordinator, then it chooses the coordinator's value to "echo" it to everyone. Otherwise, it simply echoes all the messages received in the first phase. At this point, each process p_i waits until it receives ECHO messages from enough $(n - t_0)$ distinct processes where every value in those messages was also received by p_i in the first phase. Finally, the processes try to decide. If process p_i has only one candidate value v, then p_i adopts that value v as its estimate. In that case, it can decide v if it matches the parity of the round, i.e., if $v = r_i \mod 2$. Otherwise, if p_i has more than one candidate value, then it adopts as its estimate $r_i \mod 2$, the parity of the round.

Ledgers and certificates. In order to ensure accountability, we need to record enough information during the execution to justify any decision that is made. To this end, we define two types of justifications: (i) a ledger designed to justify adopting a specific value and (ii) a certificate to justify a decision. We attach ledgers to certain messages and discard any message containing an invalid or malformed ledger. We define a ledger for round r and value v as follows. If $v \neq r \bmod 2$, then the ledger consists of the $(n - t_0)$ ECHO messages, each properly signed, received in Phase 2 of round r that contain only value v (and no other value). If $v = r \bmod 2$, then the ledger is simply a copy of *any* other ledger from the previous round $r - 1$ justifying value v. A certificate for a decision value v in round r is $(n - t_0)$ echo messages, each properly signed, received in Phase 2 of round r that contain only value v.

Accountability. We now explain how the ledgers and certificates are used. In every round, when a process uses bv-broadcast to send a message containing a value, it attaches a ledger from the previous round justifying why that value was adopted (except in Round 1 where no ledger is available to justify 1). The bv-broadcast ignores the ledger for the purpose of deciding when to echo a message. When it echoes a message m, it chooses any arbitrary non-empty ledger that was attached to a message containing m. However, every message that does not contain a valid ledger justifying its value is discarded, with the following exception: in Round 2, messages containing value 1 can be delivered without a ledger.

Whenever there is only one candidate value received in Phase 2, a process adopts that value and either (i) decides and constructs a certificate or (ii) does not decide and constructs a ledger. In both cases, this construction simply relies on the signed messages received in Phase 2 of that round. If a process decides a value v in round $r > 1$, or adopts v because it is the parity bit for round $r > 1$, then it also constructs a ledger justifying why it adopted that value v. It accomplishes this by examining all the bv-broadcast messages received for value v and copying a round $r - 1$ ledger. This is always possible since any message that is not accompanied by a valid ledger is ignored.

Proving culpability. When a process decides in round r, it sends its certificate to all the other processes. Any process that decides a different value in a round $r' > r$ can prove the culpability of at least $\lceil n/3 \rceil$ Byzantine processes by comparing this certificate to its logged ledgers. We will say that a certificate (e.g., from p_1) and a ledger (e.g., from p_2) *conflict* if they are constructed in the same round r, but for different values v and w. We now discuss how to find conflicting certificates and ledgers. Assume that process p_i decides value v in round r, and that process p_j decides a different value w in a round $> r$ after the network stabilizes. If p_j does not decide v, then, by looking at the messages received in round $r + 1$ and $r + 2$, it can identify a ledger that conflicts with the decision certificate of p_i and hence can prove the culpability of at least $t_0 + 1$ malicious processes. (Proofs are in the report [3].)

References

1 Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. Technical Report 1710.09437v4, arXiv, January 2019. arXiv:1710.09437v4.

2 Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4), 2002.

3 Pierre Civit, Seth Gilbert, and Vincent Gramoli. Polygraph: Accountable Byzantine consensus. In *Workshop on Verification of Distributed Systems (VDS)*, June 2019. Available at https://eprint.iacr.org/2019/587.pdf.

4 Tyler Crain, Vincent Gramoli, Mikel Larrea, and Michel Raynal. DBFT: Efficient leaderless Byzantine consensus and its applications to blockchains. In *IEEE NCA*, 2018. URL: http://gramoli.redbellyblockchain.io/web/doc/pubs/DBFT-preprint.pdf.

5 Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. PeerReview: Practical accountability for distributed systems. *SOSP*, 2007.

Brief Announcement: What Can(Not) Be Perfectly Rerouted Locally

Klaus-Tycho Foerster
Faculty of Computer Science, University of Vienna, Austria
klaus-tycho.foerster@univie.ac.at

Juho Hirvonen
Aalto University, Finland
juho.hirvonen@aalto.fi

Yvonne-Anne Pignolet
DFINITY, Zürich, Switzerland
yvonneanne@dfinity.org

Stefan Schmid
Faculty of Computer Science, University of Vienna, Austria
stefan_schmid@univie.ac.at

Gilles Tredan
LAAS-CNRS, Toulouse, France
gtredan@laas.fr

Abstract

In order to provide a high resilience and to react quickly to link failures, modern computer networks support fully decentralized flow rerouting, also known as local fast failover. In a nutshell, the task of a local fast failover algorithm is to *pre-define* fast failover rules for each node using locally available information only. Ideally, such a local fast failover algorithm provides a *perfect resilience* deterministically: a packet emitted from any source can reach any target, as long as the underlying network remains connected. Feigenbaum et al. showed [3] that it is not always possible to provide perfect resilience; on the positive side, the authors also presented an efficient algorithm which achieves at least 1-resilience, tolerating a single failure in any network.

Interestingly, not much more is known currently about the feasibility of perfect resilience. This brief announcement revisits perfect resilience with local fast failover, both in a model where the source can and cannot be used for forwarding decisions. By establishing a connection between graph minors and resilience, we prove that it is impossible to achieve perfect resilience on *any non-planar graph*; On the positive side, we can derive perfect resilience for outerplanar and some planar graphs.

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

Keywords and phrases Resilience, Local Failover

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.46

Related Version A full version of the paper is available at `https://arxiv.org/abs/2006.06513`.

Funding Research supported by WWTF project WHATIF, ICT19-045, 2020-2024.

Acknowledgements We would like to thank Jukka Suomela for several fruitful discussions.

1 Introduction

The dependability of distributed systems often critically depends on the underlying network, realized by a set of routers. To provide high availability, modern routers support local fast rerouting of flows: routers can be pre-configured with conditional failover rules which define,

© Klaus-Tycho Foerster, Juho Hirvonen, Yvonne-Anne Pignolet, Stefan Schmid, and Gilles Tredan; licensed under Creative Commons License CC-BY

Table 1 Summary of our results on perfect resilience for specific graph classes.

Graph class	Without source matching	With source matching
Outerplanar	Perfect resilience: Thm 1	Perfect resilience (see left)
K_4	Perfect resilience: Thm 2	Perfect resilience (see left)
Planar graphs	$\lvert V\rvert = 7$ counterexample: Thm 5	$\lvert V\rvert = 8$ counterexample: Thm 5
Non-planar graphs	Perfect res. impossible: Thm 4	Exact classification open

for each incoming port and desired target, to which port a packet arriving on this incoming port should be forwarded deterministically depending on the status of the *incident* links only: as routers need to react quickly, they do not have time to learn about remote failures.

This paper is motivated by the following fundamental question introduced by local fast rerouting mechanisms: *Is it possible to pre-define deterministic local failover rules which guarantee that packets reach their target, as long as the underlying network is connected?* This desired property is known as *perfect resilience*. The challenge of providing perfect resilience hence lies in the decentralized nature of the problem, where routers only have local information on failed links; achieving perfect resilience is straightforward with global knowledge.

Unfortunately, perfect resilience cannot be achieved in general: Feigenbaum et al. [3] presented an example with 12 nodes for which, after certain failures, no forwarding pattern on the original network allows each surviving node in the target's connected component to reach the target. Chiesa et al. [2] expanded on their result to require only two failures, but required over 30 nodes. On the positive side, Feigenbaum et al. showed that it is at least always possible to tolerate one link failure, i.e., to be 1-resilient. Interestingly, not much more is known today about when perfect resilience can be achieved, and when not. Our results are summarized in Table 1, with the full proofs and further results being available in [4].

2 Model

Let $G = (V, E)$ be a network represented by an undirected graph of nodes ("routers") V connected through undirected links E along which packets are exchanged. Initially, an arbitrary set $F \subset E$ of links fail (rendering them unusable in both directions). We study the class of local routing (forwarding) algorithms, in which every node $v \in V$ takes deterministic routing decisions based solely on the target t of the packet to route, the set of incident failed links $F \cap E(v)$, and the receiving or incoming port (*in-port*) of the packet at node i. Note that without knowledge of the in-port, already very simple failure scenarios prevent resilience. We also study the model where the forwarding may depend on a source node s. In particular, this implies that neither the state of the packet nor the state of the node can be changed, e.g., by header rewriting or using dynamic routing tables. We say that a forwarding pattern is *perfectly resilient* if it always succeeds in the connected component of the target.

3 Possibility of Perfect Resilience

For outerplanar graphs there exists a planar embedding such that all nodes are part of the outer face and this property holds also after arbitrary failures as long as the graph remains connected. Thus we can route along the links of the outer face of a planar graph using the well known right-hand rule [1] despite failures. Note that the face-routing pattern is even target-oblivious: starting on any node, it will visit every node. Moreover, this approach can also be applied for planar graphs if source and target are on the same face.

▶ **Theorem 1.** *Let $G = (V, E)$ be i) an outerplanar graph or ii) a planar graph where the packet starts on the same face as the destination. Then, there is a perfectly resilient forwarding pattern which does not require source matching.*

So far, we established that perfect resiliency is possible on outerplanar graphs as well as on the same face of planar graphs. This raises the question if perfect resilience is possible on some non-outerplanar planar graphs, which we answer in the affirmative for the clique K_4: we employ forwarding along a cyclic permutation, unless the destination is a neighbor.

▶ **Theorem 2.** *K_4 allows for perfectly resilient forwarding patterns without source matching.*

4 Impossibility of Perfect Resilience

We observe that a perfectly resilient algorithm is also perfectly resilient on subgraphs and contractions of its original graph, by a simulation argument. As subsetting and contracting are the two fundamental operations in the minor relationship, we deduce that the existence of a perfectly resilient algorithm on a graph G implies its existence on any minor of G.

▶ **Theorem 3.** *If G permits a perfectly resilient forwarding pattern, so do its minors, for both with and without source matching.*

We can hence prove that if a graph is not planar, it does not allow for perfect resilience without source matching, as both K_5 and $K_{3,3}$ do not allow perfect resilience: the latter can be shown by case distinction, where the routing cycles along nodes the three or four nodes no longer neighboring the destination. In contrast, for the K_4 after failures that leave the graph connected, at most two nodes are not direct neighbors of the destination.

▶ **Theorem 4.** *If G is not planar, then it is not perfectly resilient without source matching.*

We now show that there are planar graphs that do not permit perfect resiliency, using case distinction and further arguments in Figure 1. We can extend the result to include a source s, with slightly different argumentation, connected to the nodes 3 and 5:

▶ **Theorem 5.** *There exists a planar graph G on 7 nodes such that no forwarding pattern without source matching will succeed on G (8 nodes with source matching).*

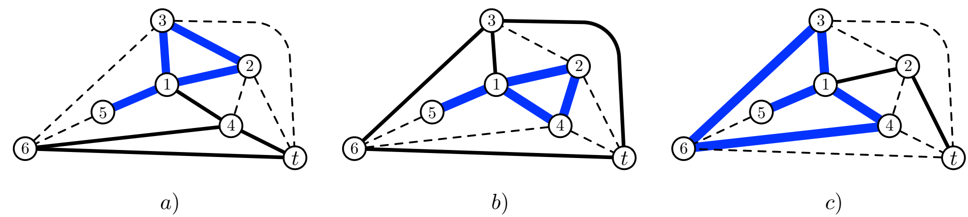


Figure 1 Planar graph with no perfectly resilient forwarding pattern. If the dashed links fail, a pattern that attempts to be perfectly resilient will be stuck in one of the loops shown in bold in the figures, depending on the routing at node 1, even though there is at least one remaining path.

References

1 J. A. Bondy and U. S. R. Murty. *Graph Theory with Applications*. Elsevier, New York, 1976.
2 M. Chiesa et al. On the resiliency of static forwarding tables. *Trans. Netw.*, 25(2), 2017.
3 J. Feigenbaum et al. BA: On the resilience of routing tables. In *Proc. PODC*, 2012.
4 K.-T. Foerster, J. Hirvonen, Y.-A. Pignolet, S. Schmid, and G. Trédan. On the feasibility of perfect resilience with local fast failover. *CoRR*, 2020. `arXiv:2006.06513`.

Brief Announcement: Byzantine Agreement, Broadcast and State Machine Replication with Optimal Good-Case Latency

Ittai Abraham
VMware Research, Herzliya, Israel
iabraham@vmware.com

Kartik Nayak
Duke University, Durham, NC, USA
kartik@cs.duke.edu

Ling Ren
University of Illinois at Urbana-Champaign, Champaign, IL, USA
renling@illinois.edu

Zhuolun Xiang
University of Illinois at Urbana-Champaign, Champaign, IL, USA
xiangzl@illinois.edu

Abstract

This paper investigates the problem *good-case latency* of Byzantine agreement, broadcast and state machine replication in the synchronous authenticated setting. The good-case latency measure captures the time it takes to reach agreement when all non-faulty parties have the same input (or in BB/SMR when the sender/leader is non-faulty) and all messages arrive instantaneously. Previous result implies a lower bound showing that any Byzantine agreement or broadcast protocol tolerating more than $n/3$ faults must have a good-case latency of at least Δ. Our first result is a matching tight upper bound for a family of protocols we call 1Δ. We propose a protocol 1Δ-BA that solves Byzantine agreement in the synchronous and authenticated setting with optimal good-case latency of Δ and optimal resilience $f < n/2$. We then extend our protocol and present 1Δ-BB and 1Δ-SMR for Byzantine fault tolerant broadcast and state machine replication, respectively, in the same setting and with the same optimal good-case latency of Δ and $f < n/2$ fault tolerance.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Security and privacy → Distributed systems security

Keywords and phrases Byzantine broadcast, synchrony, latency, state machine replication

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.47

Related Version A full version of the paper is available at https://arxiv.org/abs/2003.13155.

Funding This work is supported in part by a VMware early career faculty grant.

1 Introduction

Byzantine agreement (BA) and Byzantine broadcast (BB) are fundamental problems in distributed computing, and one of the most important practical applications of BA and BB is to implement Byzantine fault tolerant (BFT) state machine replication (SMR).

In this paper, we argue that a new model for BA and BB is needed due to the following practical considerations. First, lock-step execution where replicas start and end each round at the same time, should not be assumed. Most theoretical works assume lock-step execution, as a result, the latency of BA/BB protocols is typically measured by the round complexity. Such a lock-step assumption simplifies the protocol design but it is considered impractical because it not only is hard to enforce but also leads to poor performance. Secondly, BFT

© Ittai Abraham, Kartik Nayak, Ling Ren, and Zhuolun Xiang;
licensed under Creative Commons License CC-BY

SMR protocols typically care about the *good-case*, in which a stable honest leader stays in charge and drives consensus on many decisions. However, classical BA/BB protocols tend to optimize their *worst-case* latency, which is $f+1$ rounds for tolerating f faults [2]. Since f is typically assumed to be linear in n, any BA/BB protocol will inevitably have a poor worst-case latency as n increases. Furthermore, for a more accurate characterization of latency, we adopt the separation between the conservative bound Δ and the actual (unknown) bound δ [3]. Finally, instead of measuring only the traditional *worst-case* latency to terminate, practical SMR protocols are intended to run forever; replicas *commit* or *decide* on an ever-growing sequence of values. Motivated by the above considerations, we propose the *good-case latency* to commit as the main metric, which is defined as follows.

▶ **Definition 1** (good-case latency for Byzantine fault tolerant state machine replication). *The good-case latency is the maximal latency (over all adversarial strategies) until all honest replicas commit given:*

1. *(good leader) An honest leader is in charge.*
2. *(good network) All messages arrive instantaneously, i.e.,* $\delta = 0$.
3. *(good compute) All local computation is instantaneous.*

Of course, in practice, communication and computation are never instantaneous, but the good case definition captures the intuition that they are much smaller than the pessimistic upper bound Δ. From the good-case latency definition of BFT SMR above, it is natural to define the *good-case latency for BB* by replacing the good leader property with *good sender*: the designated sender is honest. Similarly, we define the *good-case latency for BA* by replacing the good leader property with *good input*: all honest replicas have the same input. The good input property also ties back to the good-case of BFT SMR because if the leader is honest, then all honest replicas receive the same input from the leader.

The main goal of our paper is to develop BA and BB protocols with optimal good-case latency and apply them to BFT SMR protocols under the synchronous and authenticated setting. Due to space constraints, in this brief announcement we only present a synchronous Byzantine agreement protocol 1Δ-BA with optimal good-case latency of Δ (Theorem 3 and 4) and optimal resilience of $f < n/2$.

2 Byzantine Agreement with Optimal Good-case Latency

We consider n replicas in a reliable, authenticated all-to-all network, where up to f replicas can be malicious and behave in a Byzantine fashion, and rest of the replicas are honest. Without loss of generality, we assume $n = 2f+1$. We assume standard digital signatures and public-key infrastructure (PKI). We use $\langle x \rangle_p$ to denote a signed message x by replica p.

▶ **Definition 2** (Byzantine Agreement). *A Byzantine agreement protocol provides the following three guarantees.*

- *Agreement. If two honest replicas commit value* b *and* b' *respectively, then* $b = b'$.
- *Termination. All honest replicas eventually commit and terminate.*
- *Validity. If all honest replicas have the same input value, then all honest replicas commit on the value.*

We first show a lower bound of Δ on the good-case latency, adapted from [1].

▶ **Theorem 3.** *Any Byzantine agreement or broadcast protocol that is resilient to* $f \geq n/3$ *faults must have a good-case latency at least* Δ.

Initially, every replica i has an input b_i, starts Step 1 at the same time $t = 0$, and sets $b_{lck} = \perp$.

1. **Propose.** Sign and send the input value b_i to all others. Once receiving $f + 1$ distinct signed messages of the same value b, form a proposal B with these messages as $B = \{\langle b \rangle_j\}_{f+1}$.
2. **Forward.** Upon forming or receiving a valid new proposal B containing $f + 1$ distinct signed messages of the same value b, forward B to all other replicas, set $\texttt{vote-timer}_B$ for proposal B to Δ and start counting down.
3. **Vote.** When $\texttt{vote-timer}_B$ for proposal B containing value b reaches 0, if the replica does not receive another valid proposal B' containing $f + 1$ distinct signed messages of a different value $b' \neq b$, it broadcasts a vote in the form of $\langle \texttt{vote}, B \rangle_i$.
4. **Commit.** Upon collecting $f+1$ distinct signed votes $\langle \texttt{vote}, B \rangle$ of a valid proposal B containing value b at time t, (i) if $t \leq 3\Delta$, it broadcasts these $f + 1$ votes, sets $b_{lck} = b$, and commits b, (ii) if $t > 3\Delta$, it sets $b_{lck} = b$.
5. **Byzantine agreement.** At time 4Δ, invoke an instance of Byzantine agreement with b_{lck} as the input. If not committed, commit on the output of the Byzantine agreement. Terminate.

Figure 1 1Δ-BA Protocol under the Synchronous Model.

We now briefly describe the 1Δ-BA presented in Figure 1. Initially, all replicas have an input, and set their locked value b_{lck} to be some default value $\perp$. When protocol starts, all replicas first sign and broadcast its input value, and try to form a proposal containing $f + 1$ signed messages of an identical value (Step 1). When any replica i has a proposal of value b, it forwards the proposal to detect conflict (Step 2). After the replica forwards the proposal, it locally starts a timer called `vote-timer` to wait for Δ time period. If the timer expires and the replica does not receive any conflicting proposal containing a different value $b' \neq b$, it broadcasts a `vote` message for the proposal (Step 3). Once the replica gathers $f + 1$ distinct `vote` messages for the proposal containing value b within time 3Δ, it broadcasts these `vote` messages, sets locked value $b_{lck} = b$ and commits the value b. If the $f + 1$ distinct `vote` messages for value b are received later than 3Δ, the replica only sets locked value $b_{lck} = b$ without committing the value (Step 4). At time 4Δ, the replica initiates an instance of Byzantine agreement with its locked value as the input, and any replica that has not committed yet will commit on the output of the agreement (Step 5).

▶ **Theorem 4.** *1Δ-BA protocol solves Byzantine agreement in the synchronous authenticated setting with optimal good-case latency of Δ and optimal resilience of $f < n/2$.*

3 Conclusion

We propose using the non-lock-step models and the good-case latency metric for Byzantine agreement and broadcast as they better capture what matters in practical BFT SMR. In the full version of this paper, we propose the first Byzantine agreement, broadcast, and state machine replication protocols with optimal good-case latency of Δ.

References

1. Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Maofan Yin. Sync hotstuff: Simple and practical synchronous state machine replication. *IEEE Symposium on Security and Privacy (SP)*, 2020.
2. Michael J Fischer and Nancy A Lynch. A lower bound for the time to assure interactive consistency. *Information Processing Letters*, 14(4):183–186, 1982.
3. A Ierzberg and S Kutten. Efficient detection of message forwarding faults. In *Proceeding of the 8th ACM Symposium on Principles of Distributed Computing*, pages 339–353, 1989.

Brief Announcement: Multi-Threshold Asynchronous Reliable Broadcast and Consensus

Martin Hirt
Department of Computer Science, ETH Zurich, Switzerland
hirt@inf.ethz.ch

Ard Kastrati
Department of Computer Science, ETH Zurich, Switzerland
ard.kastrati@ethz.ch

Chen-Da Liu-Zhang
Department of Computer Science, ETH Zurich, Switzerland
lichen@inf.ethz.ch

Abstract

Classical protocols for reliable broadcast and consensus provide security guarantees as long as the number of corrupted parties f is bounded by a single given threshold t. If $f > t$, these protocols are completely deemed insecure. We consider the relaxed notion of *multi-threshold* reliable broadcast and consensus where validity, consistency and termination are guaranteed as long as $f \leq t_v$, $f \leq t_c$ and $f \leq t_t$ respectively. For consensus, we consider both variants of $(1-\epsilon)$-consensus and *almost-surely terminating* consensus, where termination is guaranteed with probability $(1-\epsilon)$ and 1, respectively. We give a very complete characterization for these primitives in the asynchronous setting and with no signatures:

- Multi-threshold reliable broadcast is possible if and only if $\max\{t_c, t_v\} + 2t_t < n$.
- Multi-threshold almost-surely consensus is possible if $\max\{t_c, t_v\} + 2t_t < n$, $2t_v + t_t < n$ and $t_t < n/3$. Assuming a global coin, it is possible if and only if $\max\{t_c, t_v\} + 2t_t < n$ and $2t_v + t_t < n$.
- Multi-threshold $(1-\epsilon)$-consensus is possible if and only if $\max\{t_c, t_v\} + 2t_t < n$ and $2t_v + t_t < n$.

2012 ACM Subject Classification Theory of computation → Cryptographic protocols; Theory of computation → Distributed algorithms; Security and privacy → Cryptography

Keywords and phrases broadcast, byzantine agreement, multi-threshold

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.48

Related Version A full version of the paper is available at `https://eprint.iacr.org/2020/958`.

1 Extended Abstract

Consensus and reliable broadcast are fundamental building blocks in fault-tolerant distributed computing. Consensus allows a set of parties, each holding an input, to agree on a common value v', where, if all honest parties hold the same input v, $v' = v$. Reliable broadcast allows a designated party, called the sender, to consistently distribute a value v among a set of recipients such that all honest recipients output v in case the sender is honest. If the sender is dishonest, either all honest recipients output the same value or none of them terminates. Both primitives are used typically in the design of more complex applications, including multi-party computation, verifiable secret-sharing or voting, just to name a few.

The first consensus protocol was introduced in the seminal work of Lamport et al. [5] for the model where parties have access to a complete network of point-to-point authenticated channels, and where at most $t < n/3$ parties are corrupted. Reliable broadcast was first introduced by Bracha [2] as a useful primitive to construct building blocks in asynchronous environments. Since then, both primitives has been extensively studied in many different settings [3, 4, 2, 1, 6].

© Martin Hirt, Ard Kastrati, and Chen-Da Liu-Zhang;
licensed under Creative Commons License CC-BY
34th International Symposium on Distributed Computing (DISC 2020).
Editor: Hagit Attiya; Article No. 48; pp. 48:1–48:3
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Most known fault-tolerant distributed protocols provide security guarantees in an *all-or-nothing* fashion: if up to t parties are corrupted, all security guarantees remain. However, if more than t parties are corrupted, the protocols do not provide any security guarantees. Multi-threshold protocols (also known as hybrid security) provide different security guarantees depending on the amount of corruption, thereby allowing a graceful degradation of security.

In this work, we consider consensus and reliable broadcast protocols with separate thresholds t_v, t_c and t_t for *validity*, *consistency* and *termination*, respectively. For consensus, we consider both variants of $(1-\epsilon)$-consensus and *almost-surely terminating* consensus, where termination is guaranteed with probability $(1-\epsilon)$ and 1, respectively. Our protocols work without the use of signatures and in the purely asynchronous model without the need to make any timing assumptions. Our contributions give a very complete picture of feasibility and impossibility results: 1) Multi-threshold reliable broadcast is possible if and only if $\max\{t_c, t_v\} + 2t_t < n$; 2) Multi-threshold almost-surely consensus is possible if $\max\{t_c, t_v\} + 2t_t < n$, $2t_v + t_t < n$ and $t_t < n/3$. Assuming a global coin, we further show that the condition $t_t < n/3$ can be dropped; 3) Multi-threshold $(1-\epsilon)$-consensus is possible if and only if $\max\{t_c, t_v\} + 2t_t < n$ and $2t_v + t_t < n$.

Multi-Threshold Reliable Broadcast. Reliable broadcast is a fundamental primitive in distributed computing which allows a sender to consistently distribute a message towards a set of recipients. We consider a setting with $n+1$ parties, one sender S and n recipients $\mathcal{R} = \{R_1, ..., R_n\}$. Let us denote the number of corrupted recipients (not including the sender) at the end of the protocol execution by f.

▶ **Definition 1** (Reliable Broadcast). *Let $\mathcal{M}$ be a finite message space and f be the number of corrupted recipients at the end of the execution. A protocol π where initially the sender S has an input $m \in \mathcal{M}$ and every recipient R_i upon termination outputs $m_i \in \mathcal{M}$, is a reliable broadcast protocol, with respect to thresholds t_c, t_v, and t_t, if it satisfies the following:*

- **Consistency**. *If $f \leq t_c$, then every honest recipient that terminates outputs the same message. That is, $\exists m' \in \mathcal{M} : \forall$ honest R_i that terminate $m_i = m'$.*
- **Validity**. *If $f \leq t_v$ and the sender is honest, then every honest recipient R_i that terminates outputs the sender's message. That is, $\forall$ honest R_i that terminate $m_i = m$.*
- **Termination**.
  1. *An honest sender always terminates.*
  2. *If $f \leq t_t$ and an honest recipient terminates, then every honest recipient eventually terminates.*
  3. *If $f \leq t_t$ and the sender is honest, then eventually every honest recipient terminates.*

▶ **Theorem 2.** *Multi-threshold reliable broadcast protocol is possible if and only if* $\max\{t_c, t_v\} + 2t_t < n$.

Multi-Threshold Consensus. Stated in simple terms, consensus allows a set of parties to agree on a common value. More formally, the protocol starts with every party having an input and ends with every party having a consistent output. Moreover, if every honest party starts with the same input, they keep it. Due to the FLP impossibility proof, non-terminating executions are inevitable for every consensus protocol. Hence, we require the parties to terminate only with probability 1, termed in the literature as almost-surely terminating consensus.

▶ **Definition 3** (Almost-Surely Terminating Consensus). *Let $\mathcal{M}$ be a finite message space and f be the number of corrupted parties at the end of the execution. A protocol π where initially each party has an input $x_i \in \mathcal{M}$ and finally every party P_i upon termination has an output $y_i \in \mathcal{M}$, is a consensus protocol, with respect to thresholds t_c, t_v, t_t, if it satisfies the following:*

- **Consistency**. *If $f \leq t_c$, then the output of every honest party is the same value. That is, $\exists y \in \mathcal{M} : \forall$ honest P_i that output $y_i = y$.*
- **Validity**. *If $f \leq t_v$ and every honest party has the same input value $x \in \mathcal{M}$, then the output of every honest party P_i is x. That is, $\forall$ honest P_i that output $y_i = x$.*
- **Termination**. *If $f \leq t_t$, then with probability 1 eventually every honest party outputs and terminates.*

▶ **Theorem 4.** *Multi-threshold almost-surely consensus is possible if* $\max\{t_c, t_v\} + 2t_t < n$, $2t_v + t_t < n$ *and* $t_t < n/3$.

In the full paper, we show that the bounds $\max\{t_c, t_v\} + 2t_t < n$, $2t_v + t_t < n$ are required. We leave the feasibility of almost-surely multi-threshold consensus with $t_t \geq n/3$ as an open question. However, we provide a construction that overcomes the $n/3$ bound for the case where parties have access to a global coin.

In contrast to almost-surely terminating consensus, we show that it is possible to overcome the $n/3$ bound also if we further relax the termination guarantee as follows.

▶ **Definition 5** ($(1-\epsilon)$-Consensus). *The consistency and validity property of $(1-\epsilon)$-consensus are the same as in Definition 3. We only change the termination property.*

- **Termination**. *If $f \leq t_t$, then with probability $1 - \epsilon$ eventually every honest party outputs and terminates.*

▶ **Theorem 6.** *Multi-threshold $(1-\epsilon)$-consensus is possible if and only if* $\max\{t_c, t_v\} + 2t_t < n$ *and* $2t_v + t_t < n$.

References

1. Michael Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols (extended abstract). In *Proceedings of the Second Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 27–30. ACM, 1983. `doi:10.1145/800221.806707`.
2. Gabriel Bracha. Asynchronous Byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987. `doi:10.1016/0890-5401(87)90054-X`.
3. Danny Dolev and H. Raymond Strong. Authenticated algorithms for byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983.
4. Pesech Feldman and Silvio Micali. An optimal probabilistic protocol for synchronous byzantine agreement. *SIAM Journal on Computing*, 26(4):873–933, 1997.
5. Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.
6. Achour Mostéfaoui, Moumen Hamouma, and Michel Raynal. Signature-free asynchronous Byzantine consensus with $t < n/3$ and $O(n^2)$ messages. In *ACM Symposium on Principles of Distributed Computing*, pages 2–9. ACM, 2014. `doi:10.1145/2611462.2611468`.

Brief Announcement: Game Theoretical Framework for Analyzing Blockchains Robustness

Paolo Zappalà
Cedric, Cnam, 75003 Paris, France
paolo.zappala@cnam.fr

Marianna Belotti
BDTD, Caisse des Dépôts et Consignations, 75013 Paris, France
Cedric, Cnam, 75003 Paris, France
marianna.belotti@caissedesdepots.fr

Maria Potop-Butucaru
Lip6, Sorbonne Université, 75005 Paris, France
maria.potop-butucaru@lip6.fr

Stefano Secci
Cedric, Cnam, 75003 Paris, France
stefano.secci@cnam.fr

Abstract

Blockchains systems evolve in complex environments that mix classical patterns of faults (e.g crash faults, transient faults, Byzantine faults, churn) with selfish, rational or irrational behaviors typical to economical systems. In this paper we propose a game theoretical framework in order to formally characterize the robustness of blockchains systems in terms of resilience to rational deviations and immunity to Byzantine behaviors. Our framework includes necessary and sufficient conditions for checking the immunity and resilience of games and a new technique for composing games that preserves the robustness of individual games. We prove the practical interest of our formal framework by characterizing the robustness of three different protocols popular in blockchain systems: a HTLC-based payment scheme (a.k.a. Lightning Network), a side-chain protocol and a cross-chain swap protocol.

2012 ACM Subject Classification Networks

Keywords and phrases Blockchains, Game Theory, Byzantine-Altruistic-Rational behaviours

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.49

1 Introduction

Distributed Ledger Technologies (DLTs) allow sharing a ledger of transactions among multiple users forming a peer-to-peer (P2P) network. DLTs characterized by a block architecture are called "Blockchains"; transactions are stored in blocks that are chained to each other by means of cryptographic methods, namely hash functions. Blockchain systems are the composition of various protocolar building blocks enabling its users to transfer cryptoassets in a decentralized manner. Beyond the traditional blockchain protocols that exist today [7, 12, 17], the literature proposes other protocols that respectively define and regulate interactions outside the blockchain (*layer-2 protocols* [10]) and between different blockchains (*cross-chain protocols* [9]). Each of these protocols establishes the instructions that a user must follow in order to interact with or through a blockchain.

In a Blockchain system, agents can be classified in three different categories accordingly to [3]: (i) players who follow the prescribed protocol are called *altruistic*, (ii) those who act in order to maximise their own benefit are said to be *rational* and, (iii) players who may arbitrarily deviate from the prescribed protocol are defined as *Byzantine*. Moreover, protocols can be classified in: *Byzantine Altruistic Rational Tolerant* (BART) protocols that guarantee the safety and liveness properties in the presence of rational deviations and

© Paolo Zappalà, Marianna Belotti, Maria Potop-Butucaru, and Stefano Secci;
licensed under Creative Commons License CC-BY
34th International Symposium on Distributed Computing (DISC 2020).
Editor: Hagit Attiya; Article No. 49; pp. 49:1–49:3
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Incentive-Compatible Byzantine Fault Tolerant (IC-BFT) that incentivize rational agents to follow the prescribed protocol, also in presence of Byzantine players.

In this context, game theory helps in designing IC-BFT protocols guaranteeing that rational players follow the prescribed protocol's instructions. Concerning layer-2 and cross-chain protocols, game theoretical analysis are carried out by [4, 5, 6, 8]. More precisely, authors in [4, 5] design IC-BFT off-chain channels. In [6, 8] authors adopt the Nash equilibrium solution concept to respectively evaluate the stability of various network structures and the stability of existing cross-chain swap protocols.

Our contribution. This paper presents a game theoretical framework to analyze the robustness of blockchains systems, in terms of resilience to rational deviations and immunity to Byzantine behaviors; it is the first one, as of our knowledge, with respect to the current state of the art. The closest work to ours was proposed in [2] where the authors introduce the concept of *mechanism* (a pair game-prescribed strategy). In order to characterize the robustness of a distributed system authors in [2] introduce the notions of k-resiliency and t-immunity. In a k-resilient equilibrium there is no coalition of k players having an incentive to simultaneously change strategy to get a better outcome. On the other hand, the concept of t-immunity evaluates the risk of a set of t players to have a Byzantine behavior. The property of t-immunity is often impossible to be satisfied by practical systems [1]. We thus introduce the concept of *t-weak-immunity*. A mechanism is t-weak-immune if any altruistic player receives no worse payoff than the initial state, no matter how any set of t players deviate from the prescribed protocol. We further extend the framework in [2] by proving the necessary and sufficient conditions for a mechanism to be optimal resilient and t-weak-immune. Moreover, we define a new operator for mechanism composition and prove that it preserves the robustness properties of the individual games. In this way, we characterize the robustness of complex protocols via the composition of simpler robust building blocks.

The effectiveness of our framework is demonstrated by its capability to capture the robustness of various blockchain protocols. We studied (k,t)-robustness and (k,t)-weak-robustness (i.e., optimal k-resilience and t-weak-immunity) of Lightning Network protocol [14], a side-chain protocol [15] and the very first implementation of a cross-chain swap protocol proposed in [13] and formalized in [11]. Our analysis spotted the weakness of the Lightning Network protocol [14] to Byzantine behaviour. More precisely, the protocol's strategy to close a channel (i.e., an off-chain payment line) is neither weak immune nor immune. Thus, we provide an alternative protocol (i.e., an alternative closing module) satisfying weak immunity. The protocols regulating transactions in side-chains (i.e., secondary independent blockchain)

Table 1 Immunity and resilience properties for Lightning Network [14], the modified version with a different closing module, a side-chain protocol [15] and a cross-chain swap protocol [11, 13].

Protocol	Optimal Resilience	Weak Immunity	Immunity
Lightning Network [14]	**Yes**	**No**	**No**
Closing module	Yes	No	No
Other modules	Yes	Yes	No
Modified Lightning Network	**Yes**	**Yes**	**No**
Alternative closing module	Yes	Yes	No
Other modules	Yes	Yes	No
Side-chain (Platypus [15])	**Yes**	**Yes**	**No**
Cross-chain Swap [11, 13]	**Yes**	**Yes**	**No**

proposed in [15] is weak immune since players following the protocol never get negative utility when all the other players act as adversaries. Composition of games is used to prove the weak immunity of the cross-chain swap protocol proposed in [11, 13]. Each blockchain corresponds to a game that is weak immune, thus the composition preserves the weak immunity. Our results are reported in Table 1. The details can be found in [16].

References

1 Ittai Abraham, Lorenzo Alvisi, and Joseph Y. Halpern. Distributed computing meets game theory: combining insights from two fields. *Acm Sigact News*, 42(2):69–76, 2011.

2 Ittai Abraham, Danny Dolev, Rica Gonen, and Joe Halpern. Distributed computing meets game theory: Robust mechanisms for rational secret sharing and multiparty computation. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Principles of Distributed Computing*, PODC '06, page 53–62, 2006.

3 Amitanand S. Aiyer, Lorenzo Alvisi, Allen Clement, Michael Dahlin, Jean-Philippe Martin, and Carl Porth. Bar fault tolerance for cooperative services. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 45–58, 2005.

4 Georgia Avarikioti, Eleftherios Kokoris Kogias, and Roger Wattenhofer. Brick: Asynchronous state channels. *arXiv preprint*, 2019. `arXiv:1905.11360`.

5 Georgia Avarikioti, Felix Laufenberg, Jakub Sliwinski, Yuyi Wang, and Roger Wattenhofer. Towards secure and efficient payment channels. *arXiv preprint*, 2018. `arXiv:1811.12740`.

6 Georgia Avarikioti, Rolf Scheuner, and Roger Wattenhofer. Payment networks as creation games. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, pages 195–210. Springer, 2019.

7 Marianna Belotti, Nikola Božić, Guy Pujolle, and Stefano Secci. A vademecum on blockchain technologies: When, which, and how. *IEEE Communications Surveys & Tutorials*, 21(4):3796–3838, 2019.

8 Marianna Belotti, Stefano Moretti, Maria Potop-Butucaru, and Stefano Secci. Game theoretical analysis of Atomic Cross-Chain Swaps. In *40th IEEE International Conference on Distributed Computing Systems, ICDCS*, 2020. URL: `https://hal.archives-ouvertes.fr/hal-02414356`.

9 Michael Borkowski, Daniel McDonald, Christoph Ritzer, and Stefan Schulte. Towards atomic cross-chain token transfers: State of the art and open questions within tast. *Distributed Systems Group TU Wien, Report*, 2018.

10 Lewis Gudgeon, Pedro Moreno-Sanchez, Stefanie Roos, Patrick McCorry, and Arthur Gervais. Sok: Off the chain transactions. *IACR Cryptology ePrint Archive*, 2019:360, 2019.

11 Maurice Herlihy. Atomic cross-chain swaps. In *Proceedings of the 2018 ACM symposium on principles of distributed computing*, pages 245–254, 2018.

12 Christopher Natoli, Jiangshan Yu, Vincent Gramoli, and Paulo Esteves-Verissimo. Deconstructing blockchains: A comprehensive survey on consensus, membership and structure. *arXiv preprint*, 2019. `arXiv:1908.08316`.

13 Tier Nolan. Re: Alt chains and atomic transfers, 2013. Accessed on July 30, 2020. `https://bitcointalk.org/index.php?topic=193281.msg2224949#msg2224949`.

14 Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments, 2016. `https://lightning.network/lightning-network-paper.pdf`.

15 Alejandro Ranchal-Pedrosa and Vincent Gramoli. Platypus: Offchain protocol without synchrony. In *2019 IEEE 18th International Symposium on Network Computing and Applications, NCA*, pages 1–8, 2019.

16 Paolo Zappalà, Marianna Belotti, Maria Potop-Butucaru, and Stefano Secci. Game Theoretical Framework for Analyzing Blockchains Robustness, May 2020. URL: `https://hal.archives-ouvertes.fr/hal-02634752`.

17 Zibin Zheng, Shaoan Xie, Hong-Ning Dai, Xiangping Chen, and Huaimin Wang. Blockchain challenges and opportunities: A survey. *International Journal of Web and Grid Services*, 14(4):352–375, 2018.

Brief Announcement: Jiffy: A Fast, Memory Efficient, Wait-Free Multi-Producers Single-Consumer Queue

Dolev Adas
Technion – Israel Institute of Technology, Haifa, Israel
sdolevfe@cs.technion.ac.il

Roy Friedman
Technion – Israel Institute of Technology, Haifa, Israel
roy@cs.technion.ac.il

Abstract

In applications such as sharded data processing systems, data flow programming and load sharing applications, multiple concurrent data producers are feeding requests into the same data consumer. This can be naturally realized through concurrent queues, where each consumer pulls its tasks from its dedicated queue. For scalability, wait-free queues are often preferred over lock based structures.

The vast majority of wait-free queue implementations, and even lock-free ones, support the multi-producer multi-consumer model. Yet, this comes at a premium, since implementing wait-free multi-producer multi-consumer queues requires utilizing complex helper data structures. The latter increases the memory consumption of such queues and limits their performance and scalability. Additionally, many such designs employ (hardware) cache unfriendly memory access patterns.

In this work we study the implementation of wait-free multi-producer single-consumer queues. Specifically, we propose Jiffy, an efficient memory frugal novel wait-free multi-producer single-consumer queue and formally prove its correctness. We then compare the performance and memory requirements of Jiffy with other state of the art lock-free and wait-free queues. We show that indeed Jiffy can maintain good performance with up to 128 threads, delivers better throughput than other constructions we compared against, and consumes less memory.

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

Keywords and phrases Wait-freedom, MPSC Queues, Concurrent data-structures

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.50

Funding Israel Science Foundation grant #1505/16.

1 Background

Concurrent queues are a fundamental data-exchange mechanism in multi-threaded applications. A queue enables one thread to pass a data item to another thread in a decoupled manner, while preserving ordering between operations. The thread inserting a data item is often referred to as the *producer* or *enqueuer* of the data, while the thread that fetches and removes the data item from the queue is often referred to as the *consumer* or *dequeuer* of the data. In particular, queues can be used to pass data from multiple threads to a single thread - known as *multi-producer single-consumer queue* (MPSC), from a single thread to multiple threads - known as *single-producer multi-consumer queue* (SPMC), or from multiple threads to multiple threads - known as *multi-producer multi-consumer queue* (MPMC).

MPSC is useful in sharded software architectures, resource allocation and data-flow computation schemes. In many sharded architectures, a single thread is responsible for each shard, in order to avoid costly synchronization while accessing a specific shard. In this case, multiple feeder threads (e.g., that communicate with external clients) insert requests into the

© Dolev Adas and Roy Friedman;
licensed under Creative Commons License CC-BY

queues according to the shards. Each thread that is responsible for a given shard repeatedly dequeues the next request for the shard, executes it, dequeues the next request, etc. Similarly, in a data flow graph, multiple events may feed the same computational entity, e.g., a reducer that reduces the outcome of multiple mappers. Here too each computational entity can be served by a single thread while multiple threads are sending it items (requests) to be handled.

MPMC is the most general form of a queue and can be used in any scenario. Therefore, MPMC is also the most widely studied data structure [5, 6, 7, 9, 10, 11, 12]. Yet, this may come at a premium compared to using a more specific queue implementation.

Specifically, concurrent accesses to the same data structure require adequate concurrency control to ensure correctness. The simplest option is to lock the entire data structure on each access, but this usually dramatically reduces performance due to the sequentiality and contention it imposes [4]. A more promising approach is to reduce, or even avoid, the use of locks and replace them with *lock-free* and *wait-free* protocols that only rely on atomic operations such as *fetch-and-add* (FAA) and *compare-and-swap* (CAS), which are supported in most modern hardware architectures [3]. Wait-free implementations are particularly appealing since they ensure that each operation always terminates in a finite number of steps.

Alas, known MPMC wait-free queues suffer from large memory overheads, intricate code complexity, and low scalability. In particular, it was shown that wait-free MPMC queues require the use of a helper mechanism [1]. On the other hand, as discussed above, there are important classes of applications for which MPSC queues are adequate. Such applications could therefore benefit if a more efficient MPSC queue construction was found. This motivates studying wait-free MPSC queues, which is the topic of this paper.

2 Contributions

In this work we present Jiffy, a fast memory efficient wait-free MPSC queue. Jiffy is unbounded in the the number of elements that can be enqueued without being dequeued (up to the memory limitations of the machine). Yet the amount of memory Jiffy consumes at any given time is proportional to the number of such items and Jiffy minimizes the use of pointers, to reduce its memory footprint.

To obtain these good properties, Jiffy stores elements in a linked list of arrays, and only allocates a new array when the last array is being filled. Also, as soon as all elements in a given array are dequeued, the array is released. This way, a typical enqueue operation requires little more than a simple FAA and setting the corresponding entry to the enqueued value and changing its status from `empty` to `set`. Hence, operations are very fast and the number of pointers is a multiple of the allocated arrays rather than the number of queued elements.

To satisfy linearizability and wait-freedom, a dequeue operation in Jiffy may return a value that is already past the head of the queue, if the enqueue operation working on the head is still on-going. To ensure correctness, we devised a novel mechanism to handle such entries both during their immediate dequeue as well as during subsequent dequeues.

Another novel idea in Jiffy is related to its buffer allocation policy. Naively, when the last buffer is full, any enqueuer at that point should allocate a new buffer and try adding it to the queue with a CAS. When multiple enqueuers try this concurrently, only one succeeds and the others need to free their allocated buffer. However, this both creates contention on the end of the queue and wastes CPU time in allocating and freeing multiple buffers each time. To alleviate these, in Jiffy the enqueuer of the second entry in the last buffer already allocates the next buffer and tries to add it using CAS. This way, almost always, when enqueuers reach the end of a buffer, the next buffer is already available for them with no contention.

We have implemented Jiffy and evaluated its performance in comparison with three other leading lock-free and wait-free implementations, namely WFqueue [13], CCqueue [2], and MSqueue [7]. We also examined the memory requirements for the data and code of all measured implementations using valgrind [8]. The results indicate that Jiffy is up to 50% faster than WFqueue and roughly 10 times faster than CCqueue and MSqueue in a multiple enqueuers single dequeuer workload. Jiffy is also more scalable than the other queue structures we tested, enabling more than 20 million operations per second even with 128 threads. Finally, the memory footprint of Jiffy is roughly 90% better than its competitors in the tested workloads, and provides similar benefits in terms of number of cache and heap accesses. Jiffy obtains better performance since the size of each queue node is much smaller and there are no auxiliary data structures. For example, in WFqueue, which also employs a linked list of arrays approach, each node maintains two pointers, there is some per-thread meta-data, the basic slow-path structure (even when empty), etc. Further, WFqueue employs a lazy reclamation policy, which according to its authors is significant for its performance. Hence, arrays are kept around for some time even after they are no longer useful. In contrast, the per-node meta-data in Jiffy is just a 2-bit flag, and arrays are being freed as soon as they become empty. This translates to a more (hardware) cache friendly access pattern. Also, in Jiffy dequeue operations do not invoke any atomic (e.g., FAA & CAS) operations at all.

References

1 Keren Censor-Hillel, Erez Petrank, and Shahar Timnat. Help! In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 241–250, 2015.

2 Panagiota Fatourou and Nikolaos D Kallimanis. Revisiting the Combining Synchronization Technique. In *Proc. of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, volume 47 (8), pages 257–266, 2012.

3 Maurice Herlihy. Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.

4 Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann, 2011.

5 Edya Ladan-Mozes and Nir Shavit. An Optimistic Approach to Lock-Free FIFO Queues. In *Distributed Computing, 18th International Conference (DISC)*, pages 117–131, 2004.

6 Nhat Minh Lê, Adrien Guatto, Albert Cohen, and Antoniu Pop. Correct and Efficient Bounded FIFO Queues. In *25th International Symposium on Computer Architecture and High Performance Computing*, pages 144–151. IEEE, 2013.

7 Maged M. Michael and Michael L. Scott. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 267–275, 1996.

8 Nicholas Nethercote and Julian Seward. Valgrind: A Program Supervision Framework. *Electronic notes in theoretical computer science*, 89(2):44–66, 2003.

9 William N. Scherer, Doug Lea, and Michael L. Scott. Scalable Synchronous Queues. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 147–156, 2006.

10 Michael L. Scott. Non-Blocking Timeout in Scalable Queue-Based Spin Locks. In *Proc. of the 21st ACM Symposium on Principles of Distributed Computing, (PODC)*, pages 31–40, 2002.

11 Michael L. Scott and William N. Scherer. Scalable Queue-Based Spin Locks with Timeout. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 44–52, 2001.

12 Nir Shavit and Asaph Zemach. Scalable Concurrent Priority Queue Algorithms. In *Proc. of the 18th ACM Symp. on Principles of Distributed Computing (PODC)*, pages 113–122, 1999.

13 Chaoran Yang and John Mellor-Crummey. A Wait-Free Queue as Fast as Fetch-and-Add. *Proc. of the ACM Symp. on Principles and Practice of Parallel Programming (PPoPP)*, 2016.

Brief Announcement: Concurrent Fixed-Size Allocation and Free in Constant Time

Guy E. Blelloch

Carnegie Mellon University, Pittsburgh, PA, USA
guyb@cs.cmu.edu

Yuanhao Wei

Carnegie Mellon University, Pittsburgh, PA, USA
yuanhao1@cs.cmu.edu

Abstract

We describe an algorithm for supporting allocation and free for *fixed-sized* blocks, for p asynchronous processors, with $O(1)$ worst-case time per operation, $\Theta(p^2)$ *additive* space overhead, and using only single-word read, write, and CAS. While many algorithms rely on having constant-time fixed-size `allocate` and `free`, we present the first implementation of these two operations that is constant time with reasonable space overhead.

2012 ACM Subject Classification Computing methodologies → Concurrent algorithms

Keywords and phrases malloc, free, fixed-size, concurrent, constant time

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.51

Related Version A full version of the paper is available at `https://arxiv.org/abs/2008.04296`.

Funding This work was supported in part by NSF grants CCF-1901381, CCF-1910030, and CCF-1919223. Yuanhao Wei was also supported by a NSERC PGS-D Scholarship.

1 Introduction

Dynamic memory allocation is an important problem that plays a role in many data structures. Although some data structures require variable sized memory blocks, many are built on fixed sized blocks, including linked lists and trees.

Our goal is to efficiently solve the dynamic memory allocation problem for fixed sized memory blocks in a concurrent setting. An `allocate`() returns a reference to a block, and a `free`(p) takes a reference p to an block. We say a block is *live* at some point in the execution history if the last operation that used it (either a `free` that took it, or `allocate` that returned it) was an `allocate`. Otherwise the block is *available*. As would be expected, an `allocate` must transition the returned block from available to live (i.e., the operation cannot return a block that is already live), and a `free`(p) must be applied to a live block p (i.e., the operation cannot free a block that is already available). In our setting, processes run asynchronously and may be delayed arbitrarily.

We describe a concurrent linearizable implementation of the fixed-sized memory allocation problem with the following properties.

▶ **Result 1** (Fixed-sized Allocate/Free). *Given m as the maximum number of live blocks, on p processes, we can support linearizable* **allocate** *and* **free** *for fixed sized blocks of $k \geq 2$ words, with*

1. *references that are just pointers to the block (i.e., memory addresses),*
2. *$O(1)$ worst-case time for each operation,*
3. *$k(m + \Theta(p^2))$ space,*
4. *single-word (at least pointer-width) read, write, and CAS.*

© Guy E. Blelloch and Yuanhao Wei;
licensed under Creative Commons License CC-BY

Our notion of time/space complexity includes both local and shared instructions/words. Limiting ourselves to pointer-width atomic instructions means that we do not use unbounded sequence numbers or hide bits in pointers. Achieving the last property of Result 1 requires using a recent result on implementing LL/SC from pointer-width CAS objects [3].

We are not addressing the problem of supporting an unbounded number of allocated cells. Indeed this would seem to be impossible without some assumption of how the operating system gives out large chunks of memory to allocate from. In our algorithm, the memory allocator could potentially read from a memory block that is live, but it works correctly regardless of what the user writes into the memory blocks. We believe these kinds of accesses are reasonable.

There has been recent progress in designing memory reclamation algorithms that are wait-free [9, 2, 7, 1]. With wait-free memory reclamation, the memory allocation part becomes the new limiting factor. While there has been work on lock-free memory allocation [6, 5, 8], the only work that we know of that is wait-free is by Aghazadeh and Woelfel [1]. Aghazadeh and Woelfel's `GetFree` and `Release` operations can be used to implement `allocate` and `free` in constant time, but their algorithm requires $\Omega(mp^3)$ space, making it impractical in many applications. We guarantee the same time complexity and wait-freedom while using significantly less space. While our algorithm is mostly a theoretical contribution, the constants are small and we believe it will be fast in practice as well.

2 Algorithm Overview

Our algorithm is fairly simple. For some constant $l \in \Theta(p)$, our data structure consists of local private pools that each hold $\Theta(l)$ blocks and a shared pool that maintains a stack of *batches*, each containing l blocks. The high level idea of maintaining separate private and shared pools is widely used and has been shown to be fast in practice [6, 5, 8]. In the common case, most calls to `allocate` and `free` will be handled directly by the private pools. Batches are transferred between shared and private pools occasionally to make sure there are not too many or too few free blocks in each private pool. Having too many private blocks weakens the bound in Item 3 of Result 1 because these blocks cannot be allocated by other processes. Pushing to and popping from the shared stack takes $O(p)$ time which is fairly expensive. To amortize this cost, push and pop can be broken up into p steps of $O(1)$ time each and performed across multiple calls to `allocate` or `free`. In the full version of our paper, we show how to manage the private pools so that there is at most one ongoing push or pop per process. For our shared pool, we start with Fatourou and Kallimanis's P-SIM stack [4] and modify it to achieve the following bounds:

▶ **Result 2** (Shared Stack). *Given a concurrent allocator satisfying Result 1 with parameter $k \geq 2$, assuming that reading from a memory block that has already been freed returns an arbitrary value, on p processes, we can support linearizable* **push** *and* **pop** *with*

1. *$O(p)$ worst-case time for each operation,*
2. *at most $2p$ calls to* **allocate** *and $2p$ calls to* **free** *in each operation,*
3. *$Mk + \Theta(p^2k)$ space (where M is the number of nodes in the stack),*
4. *single-word (at least pointer-width) read, write, and CAS.*

At first glance, it may seem circular that Results 1 and 2 are used to implement each other. However, this is the key trick in our algorithm. We observe that it is safe for the data structures in the shared pool to allocate memory from the same private pools as the user. Special care is needed to ensure that the private pools always have enough blocks to service both the user and the shared pool. For this to work, Property 2 of Result 2 is crucial.

In the P-SIM stack, push and pop operations help each other to ensure that each operation completes within $O(p)$ time (Property 1). The P-SIM stack can be modified to satisfy Property 4 using a recent simulation of LL/SC from CAS [3]. While there has been a lot of work on simulating LL/SC from CAS, [3] is the only one we are aware of that maintains the other properties in Result 2. To satisfy Properties 2 and 3, we add memory management while ensuring that each push or pop calls `free` at most $2p$ times. In the P-SIM stack, each process performs push and pop operations on its local copy of the shared stack, and then tries to set its local copy as the new shared stack using an SC. We modify this algorithm so that after each successful SC, the process frees all the nodes that it locally popped, and after each unsuccessful SC, the process frees all the nodes that it locally pushed. This modification sounds straightforward but it has a complication. It is possible for a process working on a outdated copy of a shared stack to read a node that is already freed by this approach. In most settings, accessing freed memory is not allowed, however, these accesses are reasonable in our setting because they are internal to our memory allocator. Whenever a node is popped off the shared stack and freed, the memory is not freed back to the operating system, instead it is made available to the user. Accessing memory that has been allocated to the user may return an arbitrary value, so such accesses are still dangerous. We protect against this by performing VL after every potentially dangerous access. If the VL returns true, then there has not been an SC on the shared stack since the process's last LL, so the process is working on an up-to-date view of the shared stack. This means that the earlier memory access was safe. If the VL returns false, then the process's subsequent SC is guaranteed to fail. In this case, the process restarts its operation and frees any nodes that it locally pushed. With these modifications, the P-SIM stack satisfies all the properties in Result 2.

References

1. Zahra Aghazadeh and Philipp Woelfel. Upper bounds for boundless tagging with bounded objects. In *International Symposium on Distributed Computing (DISC)*, pages 442–457, 2016.
2. Guy E. Blelloch and Yuanhao Wei. Concurrent reference counting and resource management in wait-free constant time, 2020. `arXiv:2002.07053`.
3. Guy E Blelloch and Yuanhao Wei. LL/SC and atomic copy: Constant time, space efficient implementations using only pointer-width CAS. In *International Symposium on Distributed Computing (DISC)*, 2020.
4. Panagiota Fatourou and Nikolaos D Kallimanis. A highly-efficient wait-free universal construction. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 325–334, 2011.
5. Anders Gidenstam, Marina Papatriantafilou, and Philippas Tsigas. NBmalloc: Allocating memory in a lock-free manner. *Algorithmica*, 58(2):304–338, 2010.
6. Maged M Michael. Scalable lock-free dynamic memory allocation. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 35–46, 2004.
7. Ruslan Nikolaev and Binoy Ravindran. Universal wait-free memory reclamation. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, page 130–143, 2020.
8. Sangmin Seo, Junghyun Kim, and Jaejin Lee. SFMalloc: A lock-free and mostly synchronization-free dynamic memory allocator for manycores. In *International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 253–263, 2011.
9. Håkan Sundell. Wait-free reference counting and memory management. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2005.

Brief Announcement: Building Fast Recoverable Persistent Data Structures with Montage

Haosen Wen[1]
University of Rochester, NY, USA
hwen5@cs.rochester.edu

Wentao Cai[1]
University of Rochester, NY, USA
wcai5@cs.rochester.edu

Mingzhe Du
University of Rochester, NY, USA
mdu5@cs.rochester.edu

Benjamin Valpey
University of Rochester, NY, USA
bvalpey@cs.rochester.edu

Michael L. Scott
University of Rochester, NY, USA
scott@cs.rochester.edu

Abstract

The recent emergence of fast, dense, nonvolatile main memory suggests that certain long-lived data structures might remain in their natural, pointer-rich format across program runs and hardware reboots. Operations on such structures must be instrumented with explicit write-back and fence instructions to ensure consistency in the wake of a crash. Techniques to minimize the cost of this instrumentation are an active topic of current research.

We present what we believe to be the first general-purpose approach to building *buffered durably linearizable* persistent data structures, and a system, Montage, to support that approach. Montage is built on top of the Ralloc nonblocking persistent allocator. It employs a slow-ticking *epoch clock*, and ensures that no operation appears to span an epoch boundary. If a crash occurs in epoch e, all work performed in epochs e and $e - 1$ is lost, but all work from prior epochs is preserved.

We describe the implementation of Montage, argue its correctness, and report on experiments confirming excellent performance for operations on queues, sets/mappings, and general graphs.

2012 ACM Subject Classification Theory of computation → Parallel computing models; Computing methodologies → Concurrent algorithms; Computer systems organization → Reliability

Keywords and phrases Durable linearizability, consistency, persistence, fault tolerance

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.52

Related Version `https://arxiv.org/abs/2009.13701`

Supplementary Material `https://github.com/urcs-sync/Montage`

Funding This work was supported in part by NSF grants CCF-1717712 and CNS-1900803, and by a Google Faculty Research award.

1 Background and Overview

A data structure that is intended to survive system crashes is generally expected to be *durably linearizable* [3], meaning (1) it is linearizable during crash-free operation, (2) each operation persists (reaches a state that will survive a crash) between its invocation and response, and (3) the order of persists matches the linearization order. At the cost of locality (trivial composability), a *buffered* durably linearizable data structure may preserve only some consistent prefix of its happens-before history on a crash.

Publications over the past few years have described some two dozen general-purpose systems to provide failure atomicity for outermost critical sections or speculative transactions on a concurrent data structure. The need for operations to persist before returning is a significant source of overhead in these systems. As an example of how one might reduce this overhead, Nawab et al. presented a buffered durably linearizable hash table (Dalí [5])

[1] The first two authors (Haosen Wen and Wentao Cai) contributed equally to this paper.

© Haosen Wen, Wentao Cai, Mingzhe Du, Benjamin Valpey, and Michael L. Scott;
licensed under Creative Commons License CC-BY
34th International Symposium on Distributed Computing (DISC 2020).
Editor: Hagit Attiya; Article No. 52; pp. 52:1–52:3
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

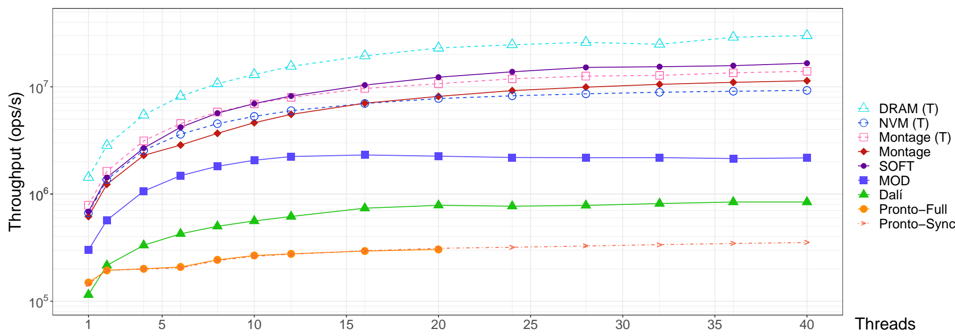


Figure 1 Persistent mapping throughput – 2:1:1 get:insert:remove; T = transient only.

that persists on a *periodic* (as opposed to incremental) basis. More recently, Haria et al.'s MOD project [2] proposed that in history-preserving ("functional") tree structures, each update can be persisted by updating a single root pointer, eliminating the need for logging. Concurrently, Memaripour et al.'s Pronto project [4] proposed that concurrent objects log their *high level* (abstract) operations (rather than low-level updates), together with occasional checkpoints; on a crash, they can then replay the suffix of the log past the most recent checkpoint. Similarly, Zuriel et al.'s SOFT [6] keeps the abstract state of a set – its keys and values – in both DRAM and NVM; the NVM copy is used only for recovery.

Inspired in part by these previous projects, we present what we believe to be the first general-purpose approach to *buffered* durably linearizable structures. Our system, Montage, preserves the *abstract* state of the concurrent object without necessarily persisting its *concrete* state. In a mapping, for example, it persists only a bag of key-value pairs; the look-up structure (hash table, tree, skip list) lives entirely in transient DRAM. During normal (crash-free) execution, Montage employs a slow-running *epoch clock*, and ensures that no operation appears to span an epoch boundary. If a crash occurs in epoch e, Montage recovers the state of the abstraction from the end of epoch $e-2$ and rebuilds the concrete structure.

Our implementation of Montage is built on top of Ralloc [1], a lock-free allocator for persistent memory. Montage itself is also lock-free during normal operation, except for epoch advances: if the data structure is itself nonblocking, a stalled thread will not impede operations of its peers; it can, however, indefinitely prevent those operations from persisting.

2 Implementation and Correctness

A typical concurrent object in Montage consists of a collection of *payload* blocks in nonvolatile memory, together with an index or other supporting structure in transient memory (DRAM). A global *epoch clock* is also kept in persistent memory. Each payload block indicates the epoch in which it was created. During recovery from a crash in epoch e, all blocks created in epochs e and $e-1$ are discarded in Ralloc. A block that was created in epoch e can be modified *in place* in epoch e (under protection of whatever synchronization would normally be used for the concurrent object). A block created in epoch $d < e$ is not modified in place; rather, a new block is created to replace it. The old block is reclaimed once the epoch counter has advanced to $e+2$, at which point we know that the new block will survive a crash. An old block can be *deleted* by replacing it with an "anti-block."

An operation that has updated a block in epoch e must abort and start over if it accesses any block that was created in epoch $e+1$; this ensures that the epoch boundary represents a consistent cut across the happens-before relationship. Epoch advance from e to $e+1$ must wait until (1) all blocks that were to be reclaimed in epoch $e-2$ have been reclaimed, and (2) all operations that began in epoch $e-1$ have completed and have persisted their updates.

Many details can be tuned for better performance. Buffered durable linearizability requires that updates to persistent data be written back and fenced by the end of the subsequent

epoch, but not necessarily before. To reduce the likelihood of cache misses induced by writing back (and, as a side effect, evicting) data that may still be useful, a thread can keep the addresses of its updates in a (transient) buffer for delayed write-back. To move the overhead of persistence off the critical path, buffers may be emptied by a background thread. To ensure nonblocking progress in application threads, updates to the epoch clock can also be confined to a background thread. Epoch length can be adjusted to trade run-time overhead against the amount of data that may be lost on a crash.

In the wake of a crash, Ralloc helps Montage iterate through all potentially in-use blocks in the heap, keeping those that are not from the two most recent epochs. The application then re-creates any needed transient structures.

3 Concrete examples

We have implemented several data structures in Montage, including a nonblocking queue, blocking and nonblocking mappings based on hash tables and trees, and general graphs with dynamic creation and deletion of vertices and edges. As a general rule, we avoid unbounded-length chains of pointers in persistent memory, since an update at the end of the chain could recursively necessitate new versions of all the predecessor blocks. Rather, we keep such chains in transient memory when needed, and arrange to rebuild them after a crash. In a queue, for example, each payload block contains a *sequence number* that indicates its place in line. In a graph, each edge is represented by a *relationship* block in persistent memory that contains the unique IDs of its endpoint vertices. Pointers from vertices to edges appear only in transient memory. Anecdotally, adapting a structure to Montage requires relatively modest programmer effort beyond the creation of the original concurrent object.

Figure 1 reports throughput for five persistent mappings: Dalí [5], synchronously logged and (on ≤ 20 threads) "full" (asynchronous) versions of Pronto [4], MOD [2], SOFT [6], and Montage. The Montage implementation uses a background thread to perform writes-back and advance the epoch every 100 ms. All implementations use a lock-based hash table as index. For comparison, we also measure a transient hash table (with data in DRAM or in NVM) and Montage without persistence (T). Only SOFT outperforms Montage, with the significant disadvantage of being limited by the size of DRAM. We interpret these results as a strong endorsement of buffered durable linearizability, and of Montage in particular.

References

1. W. Cai, H. Wen, H. A. Beadle, C. Kjellqvist, M. Hedayati, and M. L. Scott. Understanding and optimizing persistent memory allocation. In *19th Intl. Symp. on Memory Mgmt.*, June 2020.
2. S. Haria, M. D. Hill, and M. M. Swift. MOD: Minimally ordered durable datastructures for persistent memory. In *25th Intl. Conf. on Arch. Support for Prog. Lang. and Op. Sys.*, pages 775–788, March 2020.
3. J. Izraelevitz, H. Mendes, and M. L. Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In *Intl. Symp. on Dist. Comp.*, pages 313–327, September 2016.
4. A. Memaripour, J. Izraelevitz, and S. Swanson. Pronto: Easy and fast persistence for volatile data structures. In *25th Intl. Conf. on Arch. Support for Prog. Lang. and Op. Sys.*, pages 789–806, March 2020.
5. F. Nawab, J. Izraelevitz, T. Kelly, C. B. Morrey III, D. R. Chakrabarti, and M. L. Scott. Dalí: A periodically persistent hash map. In *Intl. Symp. on Dist. Comp.*, pages 37:1–37:16, October 2017.
6. Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen, and Erez Petrank. Efficient lock-free durable sets. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019.

Brief Announcement: Reaching Approximate Consensus When Everyone May Crash

Lewis Tseng
Boston College, Chestnut Hill, MA, USA
lewis.tseng@bc.edu

Qinzi Zhang
Boston College, Chestnut Hill, MA, USA
zhangbcu@bc.edu

Yifan Zhang
Boston College, Chestnut Hill, MA, USA
zhangbbq@bc.edu

Abstract

Fault-tolerant consensus is of great importance in distributed systems. This paper studies the *asynchronous* approximate consensus problem in the *crash-recovery model with fair-loss links*. In our model, up to f nodes may crash forever, while the rest may crash intermittently. Each node is equipped with a limited-size persistent storage that does *not* lose data when crashed. We present an algorithm that only stores three values in persistent storage – state, phase index, and a counter.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Approximate Consensus, Fair-loss Channel, Crash-recovery

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.53

1 Introduction

Fault-tolerant distributed consensus is an important problem for large-scale distributed systems. We study the asynchronous approximate consensus problem in a very weak model, crash-recovery model [5, 4, 2]. We present an algorithm for crash faults along with a proof sketch in this paper. Our technical report [7] contains the full proof, and an extension to Byzantine faults. Our algorithms are appropriate for systems with small and fragile devices such as sensor networks because of weak assumptions on devices and communication, and small space complexity. References [4, 2, 5] mainly use a failure-detector approach to achieve consensus. We are not aware of any approximate consensus algorithm in this model.

System Model. We consider a message-passing system consisting of n nodes, and at most f of which may crash forever. Once a faulty node crashes, it *cannot* recover nor send/receive messages. The rest of the nodes are called *crash-prone nodes*, which have up time and down time, and may crash and recover for infinitely many times. During the up time, the nodes can send and receive messages, but not during the down time. Each node can choose to store some data in a persistent storage so that it can retrieve the data after recovery. All the other data is lost during the down time.

The network forms a clique, i.e., each pair of nodes can directly communicate with each other. The link is assumed to be an asynchronous fair-lossy channel, which may lose message infinitely often. Following prior work [2, 5], we assume *eventual communication* – if a crash-prone node i repeatedly sends messages to another crash-prone node j, then j can receive *at least* one message during Δ units of time, where Δ is unknown a priori.

Prior solutions in traditional crash fault model [6, 3, 1] can trivially work in our model given an *unbounded* persistent storage, since each crash-prone can send the whole history of received values along with sequence numbers to emulate reliable channel. However, with a

© Lewis Tseng, Qinzi Zhang, and Yifan Zhang;
licensed under Creative Commons License CC-BY

34th International Symposium on Distributed Computing (DISC 2020).
Editor: Hagit Attiya; Article No. 53; pp. 53:1–53:3
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

bounded size persistent storage, it is impossible to emulate a reliable channel. Our algorithm only stores two values (state and phase index) and an n-bit counter.

Approximate Consensus. Approximate consensus algorithms need to satisfy the following three conditions [3]: (i) ϵ-*agreement*: The outputs of all crash-prone nodes are within ϵ; (ii) *Validity*: The output of all crash-prone nodes are within the range of the inputs; and (iii) *Termination*: Each crash-prone node decides an output value within finite time.

2 Approximate Consensus Algorithm in Crash-Recovery Model

We present a simple algorithm that solves approximate consensus for $n \geq 2f + 1$ under our crash-recovery model with fair-loss links. We use an n-bit counter, R, represented in a vector form. Each bit is either 0 or 1. Define $|R|$ as the number of 1's in vector R. We also use the function RESET(R): $R \leftarrow$ zero vector of length n, and $R[i] \leftarrow 1$. Algorithm 1 presents the code for each node i with input x_i in range $[0, K]$, where K is known a priori.

Algorithm 1 Steps at each node i; v_i, p_i, R_i stored in persistent storage.

```
1:  Initialization:  v_i ← x_i;  p_i ← 0;  RESET(R_i)
2:  broadcast (i, v_i, p_i) periodically
3:  repeat
4:      upon receive (j, v_j, p_j) do
5:          if p_j > p_i then
6:              copy state and jump to future phase:  v_i ← v_j;  p_i ← p_j;  RESET(R_i)
7:          else if p_j = p_i and R_i[j] =⊥ then
8:              R_i[j] ← 1;  v_i ← v_i + v_j
9:              if |R_i| ≥ n − f then
10:                 update state and go to next phase: v_i ← v_i/|R_i|;  p_i ← p_i + 1;  RESET(R_i)
11: until p_i ≥ p_end                                      ▷ p_end defined in Equation (1)
12: output v_i
```

Our algorithm has two differences from prior solutions [6, 3, 1]: (i) each node can "jump" to a future phase (line 6); (ii) each node directly adds a received value to its local state (line 8). (i) allows us to process incoming messages without the reliable and FIFO channel. (Prior algorithms process messages in the increasing order of phases). (ii) reduces the space usage.

The proofs for termination and validity are straightforward, and presented in [7]. ϵ-agreement is more difficult, since prior proofs [6, 3, 1] rely on the fact that a pair of nodes receive at least *one common value* for each phase (via a typical quorum intersection argument). In our case, nodes may not receive any message for a certain phase. The key challenge is to device the setup so that we can use an induction to prove a key claim. Especially, the way we define and use $V(p)$ is different from prior proofs [6, 3, 1]. Our split- and induction-based proof is useful for handling the case when nodes may not receive common values.

ϵ-Agreement Proof Sketch. Define $V(p)$ as a multi-set of phase-p states of all nodes i that has set $p_i = p$ at some point of time. For convenience, the elements in $V(p)$ are ordered *chronologically*. That is, the i-th element in $V(p)$ is the state of the i-th node that reached phase p. Define $V^k(p)$ as a multi-set of the first k elements in $V(p)$. For a finite set $S \subset \mathbb{R}$, define the range of S as $\delta(S) = \max(S) - \min(S)$. Also define the interval of S as $\rho(S) = [\min(S), \max(S)]$.

First observe that for all p, there must be at least $n - f$ states in $V(p)$. Since otherwise, no node can update to phase $p + 1$, which contradicts the termination property (proved in [7]). Next, we prove the following key induction statement $P(k)$.

▷ Claim 1. For each phase p, for all $1 \leq k \leq |V(p)|$, we have $\delta(V^k(p+1)) < r \cdot \delta(V(p))$, where r is some decrease rate to be determined later in Equation (1).

Proof Sketch of Claim 1. *Base case* $k = 1$: Each node has two ways to proceed to phase $p+1$, either by copying a phase-$(p+1)$ state or by taking average of $n - f$ phase-p states. Let node i be the first node to phase $p + 1$. Then, it must update by taking average.

For brevity, scale $\rho(V(p))$ to $[0, 1]$. Let $0 < \lambda \leq |V(p)|/2$, and note that $n-f \leq |V(p)| \leq n$. WLOG, assume that λ states in $V(p)$ are in the interval $[0, \frac{1}{2})$ and $|V(p)| - \lambda$ states are in $[\frac{1}{2}, 1]$. In this case, by simple algebra, we can show that the local state of node i in phase $p+1$ is in the interval $[\frac{n-2f}{4(n-f)}, 1]$. The other case is symmetric.

Induction step: Now suppose $P(k)$ is true. Consider the $(k + 1)$-th node in $V(p + 1)$, say node j. Node j updates to phase $p+1$ either by taking average of $n - f$ phase-p messages or by copying an existing state in phase $p + 1$. The first case is similar to the base case.

In the second case, j could be in a much lower phase. Let v_j be the state of j in $V(p+1)$. Since v_j is copied from an existing state in $V^k(p + 1)$, $V^k(p + 1)$ and $V^{k+1}(p + 1)$ contain identical values (except that $V^{k+1}(p + 1)$ has a value appearing one more time); hence, $\delta(V^k(p + 1)) = \delta(V^{k+1}(p + 1))$. Then by induction hypothesis, $P(k + 1)$ holds. ◁

The induction statement implies that $\delta(V(p + 1)) \leq r\delta(V(p))$ for all p, and the proof above also shows that $r = 1 - \frac{n-2f}{4(n-f)} = \frac{3n-2f}{4(n-f)}$. Note that $0 < r < 1$ for $n \geq 2f + 1$. Also recall the assumption that the initial range is bounded above by K, i.e., $\delta(V(0)) \leq K$. Hence, we can define the termination phase, p_{end}, as the following so that Algorithm 1 satisfies the ϵ-agreement property.

$$p_{end} = \frac{\log \epsilon - \log K}{\log(3n - 2f) - \log(4n - 4f)} = \log_r \left(\frac{\epsilon}{K}\right) \quad (1)$$

References

1 I. Abraham, Y. Amit, and D. Dolev. Optimal resilience asynchronous approximate agreement. In *International Conference On Principles Of Distributed Systems*, volume 3544, pages 229–239, December 2004. doi:10.1007/11516798_17.

2 M. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash-recovery model. *Distributed Computing*, 13:99–125, April 2000. doi:10.1007/s004460050070.

3 D. Dolev, N. A. Lynch, S. S. Pinter, E. W. Stark, and W. E. Weihl. Reaching approximate agreement in the presence of faults. *J. ACM*, 33:499–516, May 1986.

4 M. Hurfi, A. Mostéfaoui, and M. Raynal. Consensus in asynchronous systems where processes can crash and recover. In *Proceedings of the The 17th IEEE Symposium on Reliable Distributed Systems*, SRDS '98, page 280, USA, 1998. IEEE Computer Society.

5 R. Oliveira, R. Guerraoui, and A. Schiper. Consensus in the crash-recover model. In *Technical Report*. École Polytechnique Fédérale de Lausanne, 1997.

6 D. Sakavalas and L. Tseng. *Network Topology and Fault-Tolerant Consensus*, volume 9. Morgan & Claypool, May 2019. doi:10.2200/S00918ED1V01Y201904DCT016.

7 L. Tseng, Q. Zhang, and Y. Zhang. Reach approximate consensus when everyone may crash. In *Technical Report*. Boston College, 2020.

Brief Announcement: On Decidability of 2-Process Affine Models

Petr Kuznetsov
LTCI, Télécom Paris, Institut Polytechnique Paris, France
petr.kuznetsov@telecom-paris.fr

Thibault Rieutord
CEA LIST, PC 174, Gif-sur-Yvette, France
thibault.rieutord@cea.fr

Abstract

Affine models of computation, defined as subsets of iterated immediate-snapshot runs, capture a wide variety of shared-memory systems: wait-freedom, t-resilience, k-concurrency, and fair shared-memory adversaries. The question of whether a given task is solvable in a given affine model is, in general, undecidable.

In this paper, we focus on affine models defined for a system of two processes. We show that task computability of 2-process affine models is decidable and presents a complete hierarchy of five equivalence classes of 2-process affine models.

2012 ACM Subject Classification Theory of computation → Distributed computing models

Keywords and phrases Affine tasks, Decidability

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.54

Related Version A full version of the paper is available at `https://arxiv.org/abs/2008.02099`.

1 Introduction

The question of whether a task is solvable in a *wait-free* manner, i.e., in the asynchronous read-write shared-memory model with no restrictions on who and when can fail, is known to be undecidable for systems with more than 3 processes [3, 5]. We can still, however, study the *relative* computability of models of computation. The framework of *affine models* was introduced to capture task computability of various restrictions of the wait-free model [4].

More precisely, an *affine task* A on $n+1$ processes can be represented as a pure n-dimensional sub-complex of a finite number of iterations of the *standard chromatic subdivision*, i.e., $A \subseteq \mathrm{Chr}^k \mathbf{s}, k \in \mathbb{N}$, where all facets of A are of dimension n. Many shared-memory models such as t-resilience [8], k-concurrency [2] or fair adversaries [7] are characterized via affine tasks. Task computability of an *affine model* of A, denoted by A^*, is defined as follows: A^* solves a task $(\mathcal{I}, \Delta, \mathcal{O})$ if and only if there is a natural integer $b \in \mathbb{N}$ and a simplicial map $\delta\colon A^b(\mathcal{I}) \to \mathcal{O}$ such that δ is carried by Δ, i.e., $\forall s \in I, \delta(A^b(\mathcal{I})) \subseteq \Delta(s)$. A natural challenge is therefore to compare relative task computability of affine models:

A^* is *stronger* than B^*, i.e., $A^* \succeq_{\mathcal{A}} B^*$, if all tasks solvable in B^* can be solved in A^*.

Hence, we can state our problem as follows:

Given two affine tasks, A and B, is the question of whether $A^* \succeq_{\mathcal{A}} B^*$ decidable?

Equivalently, we can study decidability of the question whether A^* *solves* B, i.e., whether A solves the *simplex agreement* task on B [1]. Indeed, suppose that A^* solves B, inductively, for any $b \in \mathbb{N}$, A^* solves B^b. Thus, any task solvable in B^* can be solved in A^*.

© Petr Kuznetsov and Thibault Rieutord;
licensed under Creative Commons License CC-BY
34th International Symposium on Distributed Computing (DISC 2020).
Editor: Hagit Attiya; Article No. 54; pp. 54:1–54:3
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Figure 1 Relations between canonical affine tasks and corresponding models.

In this paper, we first present a framework for studying the decidability question above in 2-process affine models. We provide a complete hierarchy of 2-process affine models, including most, if not all, 2-process shared-memory models. We show that these models break down into five equivalence classes, where each class is equipped with a *representative* defined as a subset of a single iteration of the standard chromatic subdivision. The order depicted in Figure 1 defines the complete hierarchy of relative task computability of these five equivalence classes.

An intriguing question is whether this approach can be applied to higher-dimensional systems. One approach could be to focus on models defined using *link-connected* affine tasks.

2 Equivalence classes

Property selection. We define equivalence classes of 2-process affine tasks via a simple predicate on a set of properties. The power a 2-process system relies on the properties of *solo executions*, i.e., the endpoints of the corresponding affine task. Assuming a fixed input state, there is only one such endpoint v_0 (resp., v_1) of process p_0 (resp., p_1). We identify the following (obviously disjoint and forming a partition) classes of 2-process affine tasks:

1. There is a path from v_0 to v_1.
2. Both v_0 and v_1 belong to the task, but there is no path between v_0 to v_1.
3. Only v_0 belongs to the task.
4. Only v_1 belongs to the task.
5. Neither v_0 nor v_1 belongs to the task.

Equivalence for solving tasks. We show that for any tasks A and B in the same class, A^* solves B.

- If v_0 and v_1 are connected, then A and B are both iterations of the standard chromatic subdivision. By the simplicial approximation theorem, there exists a simplicial map that maps an iteration of A onto B. Thus $A^* \succeq_{\mathcal{A}} B^*$.
- Suppose now that there is no path between v_0 and v_1. Then simplices of A can be split into connected components. Let A_0 (resp., A_1) be the (possibly empty) connected component including v_0 (resp., v_1). We can then simply map every facets of A_0 (resp. A_1) to the facet of B containing v_0 (resp. v_1), and every facets of remaining connected components to any facet of B. This allows us to solve B using simply one iteration of A and, thus, $A^* \succeq_{\mathcal{A}} B^*$.

Canonical tasks. In each equivalence class, we can then select a characterizing representative, which we call a *canonical task*. We will show that the partial order on these canonical tasks (depicted in Figure 1) captures the relative power of the corresponding equivalence classes. Figure 2 depicts the five canonical affine tasks. The affine model of a canonical task is also called canonical.

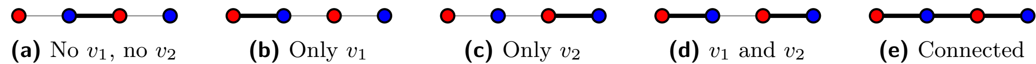


Figure 2 Representative affine tasks for the five classes.

3 Comparing equivalence classes

To prove that the partial order in Figure 1 indeed corresponds to relative task computability power of affine models, we need to show that: (1) iterations of an affine task cannot belong to a higher equivalence class; (2) carrier-preserving simplicial maps can only send tasks to those in smaller or equal classes; and (3) canonical affine models follow this order. It is easy to check that (1) and (2) imply that equivalent affine models belong to the same class. As all models in a class are equivalent, comparing canonical tasks (3) is, hence, sufficient to compare all models. Moreover, (1) and (2) also imply that models in a class cannot solve tasks in higher classes; consequently, (3) reduces to showing that a higher canonical model is stronger than a smaller one.

- **Iterating affine tasks.** An iteration of an affine task replaces each simplex with a set of simplices defined by the task for the corresponding processes. Hence, a vertex with a carrier of dimension 0 is replaced by a vertex with a carrier of dimension 0. A path is also stable under iteration in this setting as the existence of a path is equivalent to having a complete subdivision.
- **Simplicial maps.** A carrier-preserving simplicial map must send the simplex with carrier p_0 (resp., p_1) to the simplex with carrier p_0 (resp.,p_1). Therefore, models from all classes but the smallest one can only be mapped to models in smaller classes. It is also easy to check that a path between the two endpoints must be mapped to a path between them.
- **Comparing canonical models.** If neither v_0 nor v_1 belongs to the task, we can map all facets to any other task facet. Hence, this class is the strongest one. For other canonical tasks, the order follows a direct task inclusion, which implies the solvability of canonical tasks in smaller classes (the solution being the identity map).

Formal statements and proofs of the claims listed above can be found in [6].

References

1 Elizabeth Borowsky and Eli Gafni. A simple algorithmically reasoned characterization of wait-free computation (extended abstract). In *PODC*, pages 189–198, 1997.
2 Eli Gafni, Yuan He, Petr Kuznetsov, and Thibault Rieutord. Read-write memory and k-set consensus as an affine task. In *OPODIS*, pages 6:1–6:17, 2016.
3 Eli Gafni and Elias Koutsoupias. Three-processor tasks are undecidable. *SIAM J. Comput.*, 28(3):970–983, 1999.
4 Eli Gafni, Petr Kuznetsov, and Ciprian Manolescu. A generalized asynchronous computability theorem. In *PODC*, pages 222–231, 2014.
5 Maurice Herlihy and Sergio Rajsbaum. The decidability of distributed decision tasks (extended abstract). In *STOC*, pages 589–598, 1997.
6 Petr Kuznetsov and Thibault Rieutord. On decidability of 2-process affine models, 2020. `arXiv:2008.02099`.
7 Petr Kuznetsov, Thibault Rieutord, and Yuan He. An asynchronous computability theorem for fair adversaries. In *PODC*, pages 387–396, 2018.
8 Vikram Saraph, Maurice Herlihy, and Eli Gafni. Asynchronous computability theorems for t-resilient systems. In *DISC*, pages 428–441, 2016.