

# 32nd International Symposium on Distributed Computing

DISC 2018, October 15–19, New Orleans, Louisiana, USA

Edited by

Ulrich Schmid

Josef Widder



### *Editors*

Ulrich Schmid  
Embedded Computing Systems Group  
TU Wien  
Vienna, Austria  
s@ecs.tuwien.ac.at

Josef Widder  
Embedded Computing Systems Group  
TU Wien  
Vienna, Austria  
widder@ecs.tuwien.ac.at

### *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 978-3-95977-092-7**

### *Published online and open access by*

Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at <http://www.dagstuhl.de/dagpub/978-3-95977-092-7>.

### *Publication date*

October, 2018

### *Bibliographic information published by the Deutsche Nationalbibliothek*

The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at <http://dnb.d-nb.de>.

### *License*

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



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

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

The copyright is retained by the corresponding authors.

Digital Object Identifier: 10.4230/LIPIcs.DISC.2018.0

**ISBN 978-3-95977-092-7**

**ISSN 1868-8969**

**<http://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)
- Susanne Albers (TU München)
- Christel Baier (TU Dresden)
- Javier Esparza (TU München)
- Michael Mitzenmacher (Harvard University)
- Madhavan Mukund (Chennai Mathematical Institute)
- Anca Muscholl (University Bordeaux)
- Catuscia Palamidessi (INRIA)
- Raimund Seidel (Saarland University and Schloss Dagstuhl – Leibniz-Zentrum für Informatik)
- Thomas Schwentick (TU Dortmund)
- Reinhard Wilhelm (Saarland University)

**ISSN 1868-8969**

**<http://www.dagstuhl.de/lipics>**





## ■ Contents

Preface	
<i>Ulrich Schmid</i> .....	0:ix–0:x
Symposium Organization	
.....	0:xi–0:xiv
2018 Edsger W. Dijkstra Prize in Distributed Computing	
<i>Yehuda Afek, Idit Keidar, Boaz Patt-Shamir, Sergio Rajsbaum, Ulrich Schmid,</i> <i>Gadi Taubenfeld</i> .....	0:xv
2018 Principles of Distributed Computing Doctoral Dissertation Award	
<i>Lorenzo Alvisi, Idit Keidar, Andréa W. Richa, Alex Schwarzhmann</i> .....	0:xvii
Details of the DISC'18 Reviewing Process	
.....	0:xix–0:xx

### Invited Talks

Autonomous Vehicles: From Individual Navigation to Challenges of Distributed Swarms	
<i>Sándor P. Fekete</i> .....	1:1–1:1
Challenges for Machine Learning on Distributed Platforms	
<i>Tom Goldstein</i> .....	2:1–2:3
Logical Analysis of Distributed Systems: The Importance of Being Constructive	
<i>Michael Mendler</i> .....	3:1–3:1

### Regular Papers

Selecting a Leader in a Network of Finite State Machines	
<i>Yehuda Afek, Yuval Emek, and Noa Kolikant</i> .....	4:1–4:17
The Role of A-priori Information in Networks of Rational Agents	
<i>Yehuda Afek, Shaked Rafaeli, and Moshe Sulamy</i> .....	5:1–5:18
Distributed Approximate Maximum Matching in the CONGEST Model	
<i>Mohamad Ahmadi, Fabian Kuhn, and Rotem Oshman</i> .....	6:1–6:17
State Machine Replication Is More Expensive Than Consensus	
<i>Karolos Antoniadis, Rachid Guerraoui, Dahlia Malkhi, and</i> <i>Dragos-Adrian Seredinschi</i> .....	7:1–7:18
Allocate-On-Use Space Complexity of Shared-Memory Algorithms	
<i>James Aspnes, Bernhard Haeupler, Alexander Tong, and Philipp Woelfel</i> .....	8:1–8:17
Almost Global Problems in the LOCAL Model	
<i>Alkida Balliu, Sebastian Brandt, Dennis Olivetti, and Jukka Suomela</i> .....	9:1–9:16
A Population Protocol for Exact Majority with $O(\log^{5/3} n)$ Stabilization Time	

32nd International Symposium on Distributed Computing (DISC 2018).

Editors: Ulrich Schmid and Josef Widder



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

and $\Theta(\log n)$ States <i>Petra Berenbrink, Robert Elsässer, Tom Friedetzky, Dominik Kaaser, Peter Kling, and Tomasz Radzik</i> .....	10:1–10:18
Integrated Bounds for Disintegrated Storage <i>Alon Berger, Idit Keidar, and Alexander Spiegelman</i> .....	11:1–11:18
Distributed Recoloring <i>Marthe Bonamy, Paul Ouvrard, Mikaël Rabie, Jukka Suomela, and Jara Uitto</i> ...	12:1–12:17
A Tight Lower Bound for Semi-Synchronous Collaborative Grid Exploration <i>Sebastian Brandt, Jara Uitto, and Roger Wattenhofer</i> .....	13:1–13:17
Multi-Shot Distributed Transaction Commit <i>Gregory Chockler and Alexey Gotsman</i> .....	14:1–14:18
Deterministic Blind Radio Networks <i>Artur Czumaj and Peter Davies</i> .....	15:1–15:17
Detecting Cliques in CONGEST Networks <i>Artur Czumaj and Christian Konrad</i> .....	16:1–16:15
A Wealth of Sub-Consensus Deterministic Objects <i>Eli Daian, Giuliano Losa, Yehuda Afek, and Eli Gafni</i> .....	17:1–17:17
NUMASK: High Performance Scalable Skip List for NUMA <i>Henry Daly, Ahmed Hassan, Michael F. Spear, and Roberto Palmieri</i> .....	18:1–18:19
TuringMobile: A Turing Machine of Oblivious Mobile Robots with Limited Visibility and Its Applications <i>Giuseppe A. Di Luna, Paola Flocchini, Nicola Santoro, and Giovanni Viglietta</i> ...	19:1–19:18
Beeping a Deterministic Time-Optimal Leader Election <i>Fabien Dufoulon, Janna Burman, and Joffroy Beauquier</i> .....	20:1–20:17
An Almost Tight RMR Lower Bound for Abortable Test-And-Set <i>Aryaz Eghbali and Philipp Woelfel</i> .....	21:1–21:19
Distributed Set Cover Approximation: Primal-Dual with Optimal Locality <i>Guy Even, Mohsen Ghaffari, and Moti Medina</i> .....	22:1–22:14
Order out of Chaos: Proving Linearizability Using Local Views <i>Yotam M. Y. Feldman, Constantin Enea, Adam Morrison, Noam Rinetzky, and Sharon Shoham</i> .....	23:1–23:21
Redundancy in Distributed Proofs <i>Laurent Feuilloley, Pierre Fraigniaud, Juho Hirvonen, Ami Paz, and Mor Perry</i> .	24:1–24:18
Local Verification of Global Proofs <i>Laurent Feuilloley and Juho Hirvonen</i> .....	25:1–25:17
A Simple Parallel and Distributed Sampling Technique: Local Glauber Dynamics <i>Manuela Fischer and Mohsen Ghaffari</i> .....	26:1–26:11
Fast Multidimensional Asymptotic and Approximate Consensus <i>Matthias Függer and Thomas Nowak</i> .....	27:1–27:16

Local Queuing Under Contention <i>Paweł Garncarek, Tomasz Jurdziński, and Dariusz R. Kowalski</i> .....	28:1–28:18
Derandomizing Distributed Algorithms with Small Messages: Spanners and Dominating Set <i>Mohsen Ghaffari and Fabian Kuhn</i> .....	29:1–29:17
Distributed MST and Broadcast with Fewer Messages, and Faster Gossiping <i>Mohsen Ghaffari and Fabian Kuhn</i> .....	30:1–30:12
New Distributed Algorithms in Almost Mixing Time via Transformations from Parallel Algorithms <i>Mohsen Ghaffari and Jason Li</i> .....	31:1–31:16
Time-Message Trade-Offs in Distributed Algorithms <i>Robert Gmyr and Gopal Pandurangan</i> .....	32:1–32:18
Faster Distributed Shortest Path Approximations via Shortcuts <i>Bernhard Haeupler and Jason Li</i> .....	33:1–33:14
A Lower Bound for Adaptively-Secure Collective Coin-Flipping Protocols <i>Yael Tauman Kalai, Ilan Komargodski, and Ran Raz</i> .....	34:1–34:16
Adapting Local Sequential Algorithms to the Distributed Setting <i>Ken-ichi Kawarabayashi and Gregory Schwartzman</i> .....	35:1–35:17
Strong Separations Between Broadcast and Authenticated Channels <i>Julian Loss, Ueli Maurer, and Daniel Tschudi</i> .....	36:1–36:17
Broadcast and Minimum Spanning Tree with $o(m)$ Messages in the Asynchronous CONGEST Model <i>Ali Mashreghi and Valerie King</i> .....	37:1–37:17
Fault-Tolerant Consensus with an Abstract MAC Layer <i>Calvin Newport and Peter Robinson</i> .....	38:1–38:20
Randomized $(\Delta + 1)$ -Coloring in $O(\log^* \Delta)$ Congested Clique Rounds <i>Merav Parter and Hsin-Hao Su</i> .....	39:1–39:18
Congested Clique Algorithms for Graph Spanners <i>Merav Parter and Eylon Yogev</i> .....	40:1–40:18
Lattice Agreement in Message Passing Systems <i>Xiong Zheng, Changyong Hu, and Vijay K. Garg</i> .....	41:1–41:17

## Brief Announcements

Brief Announcement: Local Distributed Algorithms in Highly Dynamic Networks <i>Philipp Bamberger, Fabian Kuhn, and Yannic Maus</i> .....	42:1–42:4
Brief Announcement: Randomized Blind Radio Networks <i>Artur Czumaj and Peter Davies</i> .....	43:1–43:3
Brief Announcement: Deterministic Contention Resolution on a Shared Channel <i>Gianluca De Marco, Dariusz R. Kowalski, and Grzegorz Stachowiak</i> .....	44:1–44:3

Brief Announcement: Generalising Concurrent Correctness to Weak Memory <i>Simon Doherty, Brijesh Dongol, Heike Wehrheim, and John Derrick</i> .....	45:1–45:3
Brief Announcement: Exact Size Counting in Uniform Population Protocols in Nearly Logarithmic Time <i>David Doty, Mahsa Eftekhari, Othon Michail, Paul G. Spirakis, and Michail Theofilatos</i> .....	46:1–46:3
Brief Announcement: A Tight Lower Bound for Clock Synchronization in Odd-Ary M-Toroids <i>Reginald Frank and Jennifer L. Welch</i> .....	47:1–47:3
Brief Announcement: On Simple Back-Off in Unreliable Radio Networks <i>Seth Gilbert, Nancy Lynch, Calvin Newport, and Dominik Pajak</i> .....	48:1–48:3
Brief Announcement: Fast and Scalable Group Mutual Exclusion <i>Shreyas Gokhale and Neeraj Mittal</i> .....	49:1–49:3
Brief Announcement: On the Impossibility of Detecting Concurrency <i>Éric Goubault, Jérémy Ledent, and Samuel Mimram</i> .....	50:1–50:4
Brief Announcement: Effects of Topology Knowledge and Relay Depth on Asynchronous Consensus <i>Dimitris Sakavalas, Lewis Tseng, and Nitin H. Vaidya</i> .....	51:1–51:4
Brief Announcement: Loosely-stabilizing Leader Election with Polylogarithmic Convergence Time <i>Yuichi Sudo, Fukuhito Ooshita, Hirotsugu Kakugawa, and Toshimitsu Masuzawa</i> .	52:1–52:3

## ■ 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 2018, the 32nd International Symposium on Distributed Computing, held on October 15–19, 2018 in New Orleans, USA. It includes the citation for the 2018 Edsger W. Dijkstra Prize in Distributed Computing, jointly sponsored by DISC and PODC (the ACM Symposium on Principles of Distributed Computing), that was presented at PODC 2018 to *Bowen Alpern* and *Fred B. Schneider* for their paper “*Defining Liveness*.” The volume also includes the citation for the 2018 Doctoral Dissertation Award, also jointly sponsored by DISC and PODC, that was presented at DISC 2018 to *Rati Gelashvili* for his PhD thesis titled “*On the Complexity of Synchronization*,” supervised by Nir Shavit at the Massachusetts Institute of Technology. DISC 2018 also featured three keynote lectures, presented by Sándor P. Fekete (TU Braunschweig, Germany) on “*Autonomous Vehicles: From Individual Navigation to Challenges of Distributed Swarms*,” Tom Goldstein (University of Maryland, USA) on “*Challenges for Machine Learning on Distributed Platforms*,” and Michael Mendler (Otto-Friedrich University of Bamberg, Germany) on “*Logical Analysis of Distributed Systems: The Importance of Being Constructive*.” An abstract of each keynote lecture is included in the proceedings.

Like DISC 2017, DISC 2018 received a very high number of submissions (161 regular papers and 4 brief announcements). Every submission was read and evaluated by at least three members of the PC, assisted by 172 external reviewers, using a refined reviewing process (outlined on page xix). The Program Committee finally selected 38 regular papers and 11 brief announcements for inclusion in the conference program and in the proceedings. Among the latter, 10 are the result of inviting the authors of rejected regular submissions to provide a brief announcement version of their work. Each of those summarizes ongoing work or recent results, which were considered interesting by the PC members and where it could be expected that these results will appear as full papers in later conferences or journals.

The Best Paper Award for DISC 2018 was shared by Gregory Chockler and Alexey Gotsman for their paper “*Multi-Shot Distributed Transaction Commit*,” and Ali Mashreghi and Valerie King for their paper “*Broadcast and Minimum Spanning Tree with  $o(m)$  Messages in the Asynchronous CONGEST Model*.” Unfortunately, the authors of the nominated best student paper had to withdraw their submission at the very last moment. Revised and expanded versions of several additional selected regular papers will be considered for publication in a special issue of the journal *Distributed Computing*.

Two workshops were co-located with DISC 2018: The *7th Workshop on Advances in Distributed Graph Algorithms (ADGA)*, chaired by Merav Parter, on October 15, 2018, and the *2nd Workshop on Storage, Control, Networking in Dynamic Systems (SCNDS)*, organized by Kishori Konwar and Lewis Tseng, on October 19, 2018.

We wish to thank the many contributors to DISC 2018: the authors of the submitted papers, the PC members and the reviewers, the three keynote speakers, the conference general chair and local organizer Costas Busch, the publicity chair Peter Robinson, the proceedings

32nd International Symposium on Distributed Computing (DISC 2018).

Editors: Ulrich Schmid and Josef Widder



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

**0:x Preface**

chair Josef Widder, the web chair Wyatt Clements, all the workshop organizers led by the workshop chair Gokarna Sharma, and the DISC Steering Committee, led by Yoram Moses, for its guidance. Special thanks go to Andréa W. Richa, the PC chair of DISC 2017, for her invaluable support, and to Roman Kuznets for providing EasyChair expertise.

October 2018

Ulrich Schmid  
DISC 2018 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

Ulrich Schmid

TU Wien, Austria

### Program Committee

Ittai Abraham

VMware Research Group, USA

Marcos K. Aguilera

VMware Research Group, USA

Dan Alistarh

IST, Austria

Hagit Attiya

Technion, Israel

Janna Burman

U. Paris-Sud, France

Christian Cachin

IBM Research Zurich, Switzerland

Gregory Chockler

Royal Holloway U. of London, UK

Guy Even

Tel Aviv U., Israel

Pierre Fraigniaud

CNRS & U. Paris-Diderot, France

Mohsen Ghaffari

ETH Zurich, Switzerland

Seth Gilbert

NUS, Singapore

Robert Gmyr

U. of Houston, USA

Emmanuel Godard

Aix-Marseille U., France

Bernhard Haeupler

CMU, USA

Petr Kuznetsov

Telecom ParisTech, France

Silvio Lattanzi

Google Research, Switzerland

Christoph Lenzen

MPI for Informatics, Germany

Marios Mavronicolas

U. of Cyprus, Cyprus

Sayan Mitra

U. of Illinois, USA

Yoram Moses

Technion, Israel

Achour Mostefaoui

U. of Nantes, France

Gopal Pandurangan

U. of Houston, USA

Rafael Pass

Cornell Tech, USA

Andrzej Pelc

U. of Quebec, Canada

Rajmohan Rajaraman

Northeastern U., USA

32nd International Symposium on Distributed Computing (DISC 2018).

Editors: Ulrich Schmid and Josef Widder



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

## 0:xii Symposium Organization

Sergio Rajsbaum	UNAM, Mexico
Binoy Ravindran	Virginia Tech, USA
Andréa W. Richa	Arizona State U., USA
Peter Robinson	McMaster U., Canada
Nicola Santoro	Carleton U., Canada
Stefan Schmid	U. of Vienna, Austria
Ulrich Schmid (chair)	TU Wien, Austria
Pierre Sens	Sorbonne U., France
Gokarna Sharma	Kent State U., USA
Jukka Suomela	Aalto University, Finland
Nitin Vaidya	Georgetown University, USA
Jennifer Welch	Texas A&M, USA
Josef Widder	TU Wien, Austria
Haifeng Yu	NUS, Singapore

## Steering Committee

Roberto Baldoni	Sapienza Università di Roma, Italy
Cyril Gavoille	Bordeaux University, France
Fabian Kuhn	U. Freiburg, Germany
Yoram Moses ( <i>chair</i> )	Technion, Israel
Andréa Richa	Arizona State University, USA
Jukka Suomela	Aalto University, Finland

## Local Organization

Costas Busch ( <i>general chair</i> )	Louisiana State U., USA
Gokarna Sharma ( <i>workshop chair</i> )	Kent State U., USA
Josef Widder ( <i>proceedings chair</i> )	TU Wien, Austria
Peter Robinson ( <i>publicity chair</i> )	McMaster U., Canada
Wyatt Clements ( <i>web chair</i> )	Louisiana State U., USA

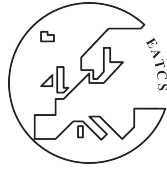
## External Reviewers

Saeed Akhoondian Amiri	Nathalie Bertrand	Bogdan Chlebus
Maya Arbel	Aditya Biradavolu	Richard Cleve
Balaji Arun	Lelia Blin	Pierluigi Crescenzi
John Augustine	Michael Blondin	Gianlorenzo D'Angelo
Alkida Balliu	Trevor Brown	Shantanu Das
Evangelos Bampas	Irina Calciu	Ajoy K. Datta
Joffroy Beauquier	Sarah Cannon	Peter Davies
Ohad Ben-Baruch	Armando Castañeda	Joshua Daymude
Ran Ben Basat	Jérémie Chalopin	Jean-Lou De Carufel
Petra Berenbrink	Bapi Chatterjee	Carole Delporte
Cédric Béranger	Soumyottam Chatterjee	Gianluca De Marco



Stéphane Devismes	Eleni Kanellou	Boaz Patt-Shamir
Dave Dice	Mohamed Karaoui	Ami Paz
Giuseppe A. Di Luna	Idit Keidar	Sriram Pemmaraju
Michael Dinitz	Maleq Khan	Eloi Perdereau
David Doty	Peter Kling	Matthieu Perrin
Swan Dubois	Marek Klonowski	Seth Pettie
Fabien Dufoulon	Kishori Konwar	Nguyen Dinh Pham
Chinmoy Dutta	Janne H. Korhonen	Pavan Poudel
Romarc Duvignau	Amos Korman	Mikaël Rabie
Faith Ellen	Eric Koskinen	Matthieu Rambaud
Ahmed Elsayed	Artur Kraska	Nicolas Rivera
Yuval Emek	Clyde Kruskal	Luis Rodrigues
Ittay Eyal	Fabian Kuhn	Will Rosenbaum
Chuchu Fan	Sandeep Kulkarni	Eric Ruppert
Reza Fathi	Saptarni Kumar	Joel Rybicki
Michael Feldmann	Shay Kutten	Laura Schmid
Klaus-Tycho Foerster	Marie Laveau	Robert Schweller
Tom Friedetzky	Douglas Lea	Michael Scott
Tobias Friedrich	Tuomo Lempiäinen	Elaine Shi
Matthias Függer	Mehraneh Liaee	Hussin Sibai
Juan A. Garay	Giuliano Losa	Devan Sohler
Leszek Gasieniec	Victor Luchangco	Ana Sokolova
Rati Gelashvili	Jan Marcinkowski	Alexander Spiegelman
Konstantinos Georgiou	Umang Mathur	Hsin-Hao Su
George Giakkoupis	Alex Matveev	Tigran Tonoyan
Brighten Godfrey	Alexandre Maurer	Jesper Larsson Träff
Wojciech Golab	Moti Medina	Jerry Trahan
Alexey Gotsman	Yuri Meshman	Amitabh Trehan
Ofer Grossman	Zarko Milosevic	Philippas Tsigas
Jan Hackfeld	Mohamed Mohamedin	Przemysław Uznański
Magnus M. Halldorsson	Anisur Rahaman Molla	Viktor Vafeiadis
David Harris	William K. Moses Jr.	Mario Valencia-Pabon
Danny Hendler	Cameron Musco	Kapil Vaswani
Maurice Herlihy	Danupon Nanongkai	Giovanni Viglietta
Ellis Hershkowitz	Emanuele Natale	Marko Vukolić
Eshcar Hillel	Ofer Neiman	Yuexuan Wang
Kristian Hinnenthal	Mikhail Nesterenko	Ben Wiederhake
Juho Hirvonen	Calvin Newport	Thomas Wies
Damien Imbs	Nicolas Nicolaou	Eric Winfree
Joseph Izraelevitz	Peter Niebert	Chuan Xu
Taisuke Izumi	Ruslan Nikolaev	Himank Yadav
Prasad Jayanti	Thomas Nowak	Yukiko Yamauchi
Siddhartha Jayanti	André Nusser	Maxwell Young
Denis Jeanneau	Dennis Olivetti	Ahad N. Zehmakan
Tomasz Jurdzinski	Eran Omri	Akka Zemmari
Hirotsugu Kakugawa	Rotem Oshman	
Nikolaos Kallimanis	Aurojit Panda	

## Sponsoring Organizations



European Association for  
Theoretical Computer Science



College of Engineering  
Louisiana State University



National Science Foundation



ORACLE



VMWARE

DISC 2018 acknowledges the use of the EasyChair system for handling submissions and managing the review process, and LIPIcs for producing and publishing the proceedings.

## ■ 2018 Edsger W. Dijkstra Prize in Distributed Computing

The Edsger W. Dijkstra Prize in Distributed Computing was created to acknowledge outstanding papers on the principles of distributed computing whose significance and impact on the theory or practice of distributed computing have been evident for at least a decade. The Prize is sponsored jointly by the ACM Symposium on Principles of Distributed Computing (PODC) and the EATCS Symposium on Distributed Computing (DISC). This award is presented annually, with the presentation taking place alternately at PODC and DISC. The 2018 Edsger W. Dijkstra Prize in Distributed Computing has been presented at PODC 2018 at the Royal Holloway University, London, UK.

The 2018 Award Committee, composed of Ulrich Schmid (Chair), Yehuda Afek, Idit Keidar, Boaz Patt-Shamir, Sergio Rajsbaum and Gadi Taubenfeld, has selected

**Bowen Alpern and Fred B. Schneider**

to receive the 2018 Edsger W. Dijkstra Prize in Distributed Computing for the outstanding paper:

Bowen Alpern and Fred B. Schneider:

**Defining liveness.**

Information Processing Letters 21(4),

October 1985, pages 181–185.

Concurrent and distributed algorithms today are characterized in terms of safety (“bad things” do not happen) and liveness (“good things” do happen). This seminal paper is what gave semantic legitimacy to that decomposition. Safety and liveness for concurrent programs had been suggested earlier by Lamport, but liveness was only formally defined for the first time in the winning paper, where it was accompanied by a compelling justification—that every (what we today call a) “trace property” is the conjunction of a safety and a liveness property. The liveness definition and accompanying decomposition theorem thus establish that safety and liveness are not only intuitively appealing but are also formally orthogonal. As a consequence, they constitute the basic building blocks of all (trace) properties and thus underlie a substantial number of papers that appeared at PODC and DISC so far.

Moreover, subsequent work has shown that invariants suffice for verifying safety properties and that variant functions on well-founded domains are suitable for verifying liveness properties. So, of the possible ways to decompose properties, the decomposition into safety and liveness provides the added value of also suggesting approaches for verifying each property. Further evidence of the importance of this work is that its topological characterizations and decomposition proof have since been scaled-up to safety and liveness hyperproperties, which express confidentiality and other important correctness concerns that trace properties cannot.

The 2018 Dijkstra Prize Committee:

- Yehuda Afek, Tel Aviv University
- Idit Keidar, Technion
- Boaz Patt-Shamir, Tel Aviv University
- Sergio Rajsbaum, UNAM
- Ulrich Schmid (chair), TU Wien
- Gadi Taubenfeld, IDC Herzliya

32nd International Symposium on Distributed Computing (DISC 2018).

Editors: Ulrich Schmid and Josef Widder



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



## ■ 2018 Principles of Distributed Computing Doctoral Dissertation Award

The winner of the 2018 Principles of Distributed Computing Doctoral Dissertation Award is **Dr. Rati Gelashvili**, for his dissertation titled “**On the Complexity of Synchronization,**” written under the supervision of Prof. Nir Shavit at the Massachusetts Institute of Technology.

The field of distributed algorithms revolves around efficiently solving synchronization tasks, such as leader election and consensus in different models. Gelashvili’s thesis provides an extraordinary study of the complexity of solving synchronization tasks, which is both deep and broad. It makes significant contributions towards understanding the complexity of solving synchronization tasks in various models. In particular, it pushes the boundary of our understanding of consensus, the algorithmic process by which asynchronous computation threads coordinate with each other, which has been the subject of extensive research for over 30 years.

In one part of his thesis, Gelashvili challenges the underpinnings of Herlihy’s consensus-based computability hierarchy, which has been the theoretical basis for classifying the computational power of concurrent data structures and synchronization primitives in multiprocessors and multicore machines for two and a half decades. He observes that Herlihy’s classical hierarchy treats synchronization instructions as distinct objects, an approach that is far from the real-world, where multiprocessors do let processes apply supported atomic instructions to arbitrary memory locations. Gelashvili shows that, contrary to common belief, solving consensus does not require multicore architectures to support “strong” synchronization instructions such as compare-and-swap. Rather, combinations of “weaker” instructions such as decrement and multiply suffice. He goes on to propose an alternative complexity-based hierarchy for concurrent objects. The dissertation further opens a new line of research by proving a linear-space bound for the anonymous case of randomized consensus, the first major progress on this problem in 15 years, which won the Best Paper Award at DISC 2015, and for which Gelashvili developed novel lower bound techniques. Apart from their great importance, these results are also technically complex and mathematically beautiful.

**The award.** The Principles of Distributed Computing Doctoral Dissertation 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. The 2018 award has been presented at DISC 2018, New Orleans, USA.

The 2018 Principles of Distributed Computing Doctoral Dissertation Award Committee:

- Lorenzo Alvisi, Cornell
- Idit Keidar (chair), Technion
- Andréa W. Richa, ASU
- Alex Schwarzmann, UConn





## ■ Details of the DISC'18 Reviewing Process

Since DISC 2018 was expected to get a similar number of submissions as DISC 2017, a large PC consisting of 39 distinguished members of the community was formed in an attempt to sufficiently cover all the 17 topics specifically addressed in the call for papers. In addition, stimulated by concerns with the reviewing process used at DISC and PODC in the past<sup>1</sup>, a number of quality-enhancing measures were foreseen for DISC 2018.

Besides enforcing the requirement for self-contained submissions (15 pages LIPIcs, without references) by disallowing appendices but encouraging full versions on publicly accessible archives like arXiv or HAL, which facilitates a fair comparison of submissions given the tight reviewing time constraints, the following measures were implemented:

- (i) To facilitate effective paper bidding, EasyChair's ability to match the selected topics of the submissions with the selected topics of expertise of the PC members was used to generate an initial bidding proposal for every PC member that could be modified during the actual paper bidding phase. The result of the bidding phase allowed EasyChair to find an optimal paper assignment (3 reviewers per submission) in a single assignment run, in negligible time.
- (ii) In order not to rule out the most competent reviewers for a submission by an overly restrictive conflict of interest policy, prohibitive CoI (like supervisor or personal relations, to be declared during bidding as usual) that forbid any access to the reviewing process, and milder forms of CoI (like occasional co-authorship, to be declared in the "comments to the PC section" of the reviews) were distinguished.
- (iii) A reviewing process with two intermediate reviews before the final review was enforced. The first intermediate review just asked for the reviewers' actual expertise for reviewing the assigned papers [1 week after paper assignment], the second intermediate review asked for an estimate of the overall merit figure (and optionally major strengths and weaknesses) [3 weeks after paper assignment]. The intermediate reviews were used to assign additional PC members/reviewers to submissions that either did not have at least 2 reviewers with expertise 3 ("knowledgable") or 4 ("expert"), or suffered from controversial merit figure estimates (a difference larger or equal to 3, from knowledgable reviewers). At the end, 50 (resp. 3) submissions ended up with 4 (resp. 5) reviewers.
- (iv) The full reviews were due 6 weeks after paper assignment, which allowed 3 weeks of discussion before the PC meeting. During paper discussion, the reviewers of a submission were supposed to either (i) resolve controversial merit figures or (ii) to determine both a proponent and an opponent is willing to make his/her case for/against the submission in the PC meeting. At the end, only 8 submissions did not fall under (i) and thus needed to be dealt with in the PC meeting.
- (v) The PC meeting (July 9–10, 2018) was set up as a virtual one using Adobe Connect. As there were only few submissions up for discussion, each of those was assigned a fixed time slot where all interested PC members could join. Depending on the outcome of the discussion, either the controversial scores were appropriately modified or additional reviews were provided.

As a result, 23 submissions ended up with an average expertise-weighted score of at least 1.7, which has been set as the threshold for a "safe accept" (at least two "accept")

---

<sup>1</sup> Also raised explicitly by a group of members of the community in the DISC 2017 business meeting.



## 0:xx **Details of the DISC'18 Reviewing Process**

and no reject), and 16 submissions with an average expertise-weighted score of at least 1.3, which has been set as the threshold for a “possible accept” (at least one “accept” and no reject). The PC eventually decided to accept all these submissions as full papers, and to invite all authors of 25 submissions with an average expertise-weighted score of at least 0.3 (at least two weak accepts) to submit a brief announcement version of their work. Ultimately, 11 accepted this invitation and submitted a brief announcement, all of which were finally accepted after a short round of additional reviewing.




# Autonomous Vehicles: From Individual Navigation to Challenges of Distributed Swarms

Sándor P. Fekete

Department of Computer Science, TU Braunschweig, 38106 Braunschweig, Germany

s.fekete@tu-bs.de

 <https://orcid.org/0000-0002-9062-4241>

---

## Abstract

Recent years have seen impressive advancements in the development of robots on four wheels: autonomous cars. While much of this progress is owed to a combination of breakthroughs in artificial intelligence and improved sensors, dealing with complex, non-ideal scenarios, where errors or failures can turn out to be catastrophic is still largely unsolved; this will require combining “fast”, heuristic approaches of machine learning with “slow”, more deliberate methods of discrete algorithms and mathematical optimization. However, many of the real challenges go beyond performance guarantees for individual vehicles and aim at the behavior of swarms: How can we control the complex interaction of a distributed swarm of vehicles, such that the overall behavior can measure up to and go beyond the capabilities of humans? Even though many of our engineering colleagues do not fully realize this yet, there is no doubt that this will have to be based to no small part on expertise in distributed algorithms.

I will present a multi-level overview of results and challenges, ranging from information exchanges of small groups all the way to game-theoretic mechanisms for large-scale control. Application scenarios do not just arise from road traffic (where short response times, large numbers of vehicles and individual interests give rise to many difficulties), but also from swarms of autonomous space vehicles (where huge distances, times and energies make distributed methods indispensable).

**2012 ACM Subject Classification** Theory of computation → Distributed algorithms, Theory of computation → Algorithmic game theory and mechanism design

**Keywords and phrases** Autonomous vehicles, interaction, robot swarms, game theory

**Digital Object Identifier** 10.4230/LIPIcs.DISC.2018.1

**Category** Invited Talk



© Sándor P. Fekete;

licensed under Creative Commons License CC-BY

32nd International Symposium on Distributed Computing (DISC 2018).

Editors: Ulrich Schmid and Josef Widder; Article No. 1; pp. 1:1–1:1

Leibniz International Proceedings in Informatics



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



# Challenges for Machine Learning on Distributed Platforms

Tom Goldstein<sup>1</sup>

University of Maryland, College Park, MD, USA  
tomg@cs.umd.edu

---

## Abstract

Deep neural networks are trained by solving huge optimization problems with large datasets and millions of variables. On the surface, it seems that the size of these problems makes them a natural target for distributed computing. Despite this, most deep learning research still takes place on a single compute node with a small number of GPUs, and only recently have researchers succeeded in unlocking the power of HPC. In this talk, we'll give a brief overview of how deep networks are trained, and use HPC tools to explore and explain deep network behaviors. Then, we'll explain the problems and challenges that arise when scaling deep nets over large system, and highlight reasons why naive distributed training methods fail. Finally, we'll discuss recent algorithmic innovations that have overcome these limitations, including "big batch" training for tightly coupled clusters and supercomputers, and "variance reduction" strategies to reduce communication in high latency settings.

**2012 ACM Subject Classification** Computing methodologies → Machine learning

**Keywords and phrases** Machine learning, distributed optimization

**Digital Object Identifier** 10.4230/LIPIcs.DISC.2018.2

**Category** Invited Talk

## 1 How do we train neural nets

Deep neural networks are one of the most flexible and powerful tools in machine learning. Neural networks are complex models that are "trained" by solving a large optimization that minimizes an objective function, called the "loss," that measures how well the neural net fits to training data. Computing this loss function is expensive because it requires summing over every element in a large training dataset. To avoid this expense, neural optimization problems are commonly solved using stochastic gradient descent (SGD). This algorithm works by randomly sampling a small batch of data on each iteration, forming an approximate loss function using only this small data sample, and then doing an approximate gradient descent step using this approximate loss function. This SGD algorithm was originally adopted because computers in the 1980s didn't have the computing power to evaluate the exact loss function (which requires the full dataset). SGD only uses a small batch of data on each iteration, and this makes each gradient descent update cheap, but "noisy" (inexact).

---

<sup>1</sup> Support for this work was provided by DARPA Lifelong Learning Machines (FA8650-18-2-7833), the US Office of Naval Research (N00014-17-1-2078), the US National Science Foundation (CCF-1535902), and the Sloan Foundation.



## 2 Why is training on HPC platforms hard?

It is commonly said that SGD is an “inherently serial” algorithm; the  $(k + 1)$ th iteration of SGD uses the result of the  $k$ th iteration as a starting point, and so iterations need to take place one at a time. Furthermore, each iteration is cheap when a small batch size is used. This makes the algorithm hard to scale – when the batch size is small, iterations are cheap and don’t require enough work to spread over a large number of workers. For example, with a minibatch size of 128 data samples (which is fairly standard for many imaging problems) and 128 workers, each worker would be processing only 1 image/sample per iteration, and the workers would have to communicate after each computation. In this case, the communication costs would far outweigh the compute costs, and training would be inefficient. This is largely what motivated the recent emergence of GPUs for machine learning. On a GPU, a single iteration of SGD can be split over 1000s of small cores in a shared memory architecture. In this case, we’re still doing the same old serial SGD algorithm, but using lots of parallelism to get each serial step done faster. This only works on a GPU because all of the cores are synchronized and memory is shared, which means there is little or no communication overhead.

## 3 Can’t we just use bigger batch sizes?

There’s an obvious (but naive) solution to the scalability problem described above: increase the batch size. This gives us more accurate (i.e., less “noisy”) gradient computations that should make the algorithm converge faster. With bigger batch sizes, there’s lots more work to do per iteration, and this work can be spread over many workers. Furthermore, if convergence happens in fewer iterations, then this could speed things up and enable training with lower wall-clock time.

But big batch training poses a problem: the argument above assumes that more accurate big-batch gradients work better than less accurate small-batch gradients. Shockingly, this is the opposite of what happens in practice; larger batches and more accurate gradients are *worse* for neural optimization. You need the noise to find good minimizers. Big-batch algorithms sometimes get stuck in local minimizers of the non-convex loss functions, or else find global minimizers that perform poorly on new data samples that weren’t used for training. In contrast, noisy methods “bounce out” of these local minimizer traps, and tend to find global minimizers that perform well on new data points that aren’t used for training. This good behavior of small batch SGD is known as “implicit regularization”; while there are many minimizers to neural loss functions, small-batch SGD creates a bias towards minimizers that avoid “over-fitting,” and perform well on test data. The difference between small batch and large batch optimization, and the cause of implicit regularization is still not well understood. The qualitative differences between large and small batch training were recently explored in [3].

## 4 So what can be done to scale up SGD in distributed environments?

There are three main approaches to scaling up SGD.

- *Find a way to use bigger batches without finding bad minimizers:* While using big batches in a naive way results in poor models, it is possible to use big batches in a more sophisticated way that still performs well. In one approach [2], we start with a small batch size at the early stages of optimization, and quickly expand the batch size to be

very large. We show that this approach helps mitigate the loss in performance that comes from starting with a big batch, while simultaneously making it easier to automate the training process.

- *Find a way to reduce communication overhead so that SGD can tolerate small batches:* This usually requires an algorithm that can do multiple iterations on a worker before communicating back to a central server. By using delayed asynchronous communication, algorithms avoid being communication bound because they keep working while they wait for communication to happen in a separate (often asynchronous) thread. Special variants of SGD can be developed in which workers share information that enables them to “stay on the same page” and search for similar solutions even when communication is infrequent. This direction was explored in [1].
- *Find problem domains where iterations are so expensive that HPC is needed, even for small batch sizes:* One such problem domain is the processing 3D datasets (as opposed to 2D images). Processing videos and 3D volumes requires a large amount of memory and far more FLOPS per byte than 2D processing. This is a new frontier domain where HPC is likely to be dominant.

---

### References

---

- 1 Soham De and Tom Goldstein. Efficient distributed SGD with variance reduction. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pages 111–120. IEEE, 2016. doi:10.1109/ICDM.2016.0022.
- 2 Soham De, Abhay Yadav, David Jacobs, and Tom Goldstein. Automated inference with adaptive batches. In Aarti Singh and Jerry Zhu, editors, *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*, volume 54 of *Proceedings of Machine Learning Research*, pages 1504–1513. PMLR, 2017. URL: <http://proceedings.mlr.press/v54/de17a.html>.
- 3 Hao Li, Zheng Xu, Gavin Taylor, and Tom Goldstein. Visualizing the loss landscape of neural nets, 2017. arXiv:1712.09913v1.



# Logical Analysis of Distributed Systems: The Importance of Being Constructive

Michael Mendler

The Otto-Friedrich University of Bamberg, Bamberg, Germany

---

## Abstract

The design and analysis of complex distributed systems proceeds along numerous levels of abstractions. One key abstraction step for reducing complexity is the passage from analog transistor electronics to synchronously clocked digital circuits. This significantly simplifies the modelling from continuous differential equations over the real numbers to discrete Mealy automata over two-valued Boolean algebra. Although typically taken for granted, this step is magic. How do we obtain clock synchronization from asynchronous communication of continuous values? How do we decide on the discrete meaning of continuous signals without a synchronization clock? From a logical perspective, the possibility of synchronization is paradoxical and appears “out of thin air.” The chicken-or-egg paradox persists at higher levels abstraction for distributed software. We cannot achieve globally consistent state from local communications without synchronization. At the same time we cannot synchronize without access to globally consistent state. From this perspective, distributed algorithms such as for leader election, consensus or mutual exclusion do not strictly solve their task but merely reduce one synchronization problem to another.

This talk revisits the logical justification of the synchronous abstraction claiming that correctness arguments, in so far as they are not merely reductions, must intrinsically depend on reasoning in classical logic. This is studied at the circuit level, where all software reductions must end. The well-known result that some synchronization elements cannot be implemented in delay-insensitive circuits is related to Berry’s Thesis according to which digital circuits are delay-insensitive if and only if they are provably correct in constructive logic. More technically, the talk will show how non-inertial delays give rise to a constructive modal logic while inertial delays are inherently non-constructive. This gives a logical explanation for why inertial delays can be used to build arbiters, memory-cells and other synchronization elements, while non-inertial delays are not powerful enough. Though these results are tentative, they indicate the importance of logical constructiveness for metastable-free discrete abstractions of physical behavior. This also indicates that metastability is an unavoidable artifact of the digital abstraction in classical logic.

**2012 ACM Subject Classification** Theory of computation → Modal and temporal logics, Theory of computation → Constructive mathematics, Computing methodologies → Concurrent algorithms, Hardware → Hardware validation

**Keywords and phrases** Hardware synchronisation, inertial delays, delay-insensitive circuits, constructive circuits, metastability, constructive modal logic

**Digital Object Identifier** 10.4230/LIPIcs.DISC.2018.3

**Category** Invited Talk

**Funding** This work is partially supported by the German Research Council (DFG) under grant number ME-1427/6-2.

**Acknowledgements** This work is based on joint work with Tom Shiple and Gérard Berry.



© Michael Mendler;

licensed under Creative Commons License CC-BY

32nd International Symposium on Distributed Computing (DISC 2018).

Editors: Ulrich Schmid and Josef Widder; Article No. 3; pp. 3:1–3:1

Leibniz International Proceedings in Informatics



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





# Selecting a Leader in a Network of Finite State Machines

Yehuda Afek<sup>1</sup>

Tel Aviv University, Tel Aviv, Israel  
afek@cs.tau.ac.il

Yuval Emek<sup>2</sup>

Technion - Israel Institute of Technology, Haifa, Israel  
yemek@technion.ac.il

Noa Kolikant

Tel Aviv University, Tel Aviv, Israel  
noakolikant@mail.tau.ac.il

---

## Abstract

This paper studies a variant of the *leader election* problem under the *stone age* model (Emek and Wattenhofer, PODC 2013) that considers a network of  $n$  randomized finite automata with very weak communication capabilities (a multi-frequency asynchronous generalization of the *beeping* model's communication scheme). Since solving the classic leader election problem is impossible even in more powerful models, we consider a relaxed variant, referred to as *k-leader selection*, in which a leader should be selected out of at most  $k$  initial candidates. Our main contribution is an algorithm that solves  $k$ -leader selection for bounded  $k$  in the aforementioned stone age model. On (general topology) graphs of diameter  $D$ , this algorithm runs in  $\tilde{O}(D)$  time and succeeds with high probability. The assumption that  $k$  is bounded turns out to be unavoidable: we prove that if  $k = \omega(1)$ , then no algorithm in this model can solve  $k$ -leader selection with a (positive) constant probability.

**2012 ACM Subject Classification** Theory of computation → Distributed computing models

**Keywords and phrases** stone age model, beeping communication scheme, leader election,  $k$ -leader selection, randomized finite state machines, asynchronous scheduler

**Digital Object Identifier** 10.4230/LIPIcs.DISC.2018.4

## 1 Introduction

Many distributed systems rely on the existence of one distinguishable node, often referred to as a *leader*. Indeed, the *leader election* problem is among the most extensively studied problems in distributed computing [23, 9, 29, 3]. Leader election is not confined to digital computer systems though as the dependency on a unique distinguishable node is omnipresent in *biological systems* as well [27, 34, 28]. A similar type of dependency exists also in networks of man-made micro- and even nano-scale sub-microprocessor devices [16].

The current paper investigates the task of electing a leader in networks operating under the *stone age (SA)* model [20] that provides an abstraction for distributed computing by nodes that are significantly inferior to modern computers in their computation and communication capabilities. In this model, the nodes are controlled by randomized finite automata and

---

<sup>1</sup> The work of Y. Afek was partially supported by a grant from the Blavatnik Cyber Security Council and the Blavatnik Computer Science Research Fund.

<sup>2</sup> The work of Y. Emek was supported in part by an Israeli Science Foundation grant number 1016/17.



can communicate with their network neighbors using a fixed message alphabet based on a weak communication scheme that can be viewed as an asynchronous extension of the *set broadcast (SB)* communication model of [25] (a formal definition of our model is provided in Section 1.1).

Since the state space of a node in the SA model is fixed and does not grow with the size of the network, SA algorithms are inherently *uniform*, namely, the nodes are anonymous and lack any knowledge of the network size. Unfortunately, classic impossibility results state that leader election is hopeless in these circumstances (even under stronger computational models): Angluin [4] proved that uniform algorithms cannot solve leader election in a network with success probability 1; Itai and Rodeh [26] extended this result to algorithms that are allowed to fail with a bounded probability.

Thus, in the distributed systems that interest us, leader election cannot be solved by the nodes themselves and some “external help” is necessary. This can be thought of as an external *symmetry breaking signal* that only one node is supposed to receive. Symmetry breaking signals are actually quite common in reality and can come in different shape and form. A prominent example for such external signaling occurs during the development process of multicellular organisms, when ligand molecules flow through a cellular network in a certain direction, hitting one cell before the others and triggering its differentiation [35].

But what if the symmetry breaking signal is *noisy* and might be received by a handful of nodes? Is it possible to detect that several nodes received this signal? Can the system recover from such an event or is it doomed to operate with multiple leaders instead of one?

In this paper, we study the *k-leader selection* problem, where at most  $k$  (and at least 1) nodes are initially marked as *candidates*, out of which exactly one should be selected. On top of the relevance of this problem to the aforementioned questions, it is also motivated by the following application. Consider scenarios where certain nodes, including the leader, may get lost during the network deployment process, e.g., a sensor network whose nodes are dropped from an airplane. In such scenarios, one may wish to produce  $k > 1$  candidate leaders with the purpose of increasing the probability that at least one of them survives; a *k-leader selection* algorithm should then be invoked to ensure that the network has exactly one leader when it becomes operational.

The rest of the paper is organized as follows. In Section 1.1, we provide a formal definition of the distributed computing model used in the paper. Our results are summarized in Section 1.2 and some additional related literature is discussed in Section 1.3. A *k-leader selection* algorithm that constitutes our main technical contribution, is presented in Section 2, whereas Section 3 provides some negative results.

## 1.1 Model

The distributed computing model considered in this paper follows the *stone age (SA)* model of Emek and Wattenhofer [20]. Under this model, the communication network is represented by a finite connected undirected graph  $G = (V, E)$  whose nodes are controlled by *randomized finite automata* with state space  $Q$ , message alphabet  $\Sigma$ , and transition function  $\tau$  whose role is explained soon.

Each node  $v \in V$  of degree  $d_v$  is associated with  $d_v$  *input ports* (or simply *ports*), one port  $\psi_v(u)$  for each neighbor  $u$  of  $v$  in  $G$ , holding the last message  $\sigma \in \Sigma$  received from  $u$  at  $v$ . The communication model is defined so that when node  $u$  sends a message, the same message is delivered to all its neighbors  $v$ ; when (a copy of) this message reaches  $v$ , it is written into port  $\psi_v(u)$ , overwriting the previous message in this port. Node  $v$ 's (read-only) access to its own ports  $\psi_v(\cdot)$  is very limited: for each message type  $\sigma \in \Sigma$ , it can only distinguish between the case where  $\sigma$  is not written in any port  $\psi_v(\cdot)$  and the case where it is written in at least one port.

The execution is event driven with an asynchronous scheduler that schedules the aforementioned message delivery events as well as node activation events.<sup>3</sup> When node  $v \in V$  is activated, the transition function  $\tau : Q \times \{0, 1\}^\Sigma \rightarrow 2^{Q \times \Sigma}$  determines (in a probabilistic fashion) its next state  $q' \in Q$  and the next message  $\sigma' \in \Sigma$  to be sent based on its current state  $q \in Q$  and the current content of its ports. Formally, the pair  $(q', \sigma')$  is chosen uniformly at random from  $\tau(q, \chi_v)$ , where  $\chi_v \in \{0, 1\}^\Sigma$  is defined so that  $\chi_v(\sigma) = 1$  if and only if  $\sigma$  is written in at least one port  $\psi_v(\cdot)$ .

To complete the definition of the randomized finite automata, one has to specify the set  $Q_{in} \subseteq Q$  of initial states that encode the node's input, the set  $Q_{out} \subseteq Q$  of output states that encode the node's output, and the initial message  $\sigma_0 \in \Sigma$  written in the ports when the execution begins. SA algorithms are required to have *termination detection*, namely, every node must eventually decide on its output and this decision is irrevocable.

Following the convention in message passing distributed computing (cf. [32]), the *run-time* of an asynchronous SA algorithm is measured in terms of *time units* scaled to the maximum of the time it takes to deliver any message and the time between any two consecutive activations of a node. Refer to [20] for a more detailed description of the SA model.

The crux of the SA model is that the number of states in  $Q$  and the size of the message alphabet  $\Sigma$  are constants independent of the size (and any parameter) of the graph  $G$ . Moreover, node  $v$  cannot distinguish between its ports and in general, its degree may be larger than  $|Q|$  (and  $|\Sigma|$ ).

**Weakening the Communication Assumptions.** The model defined in the current paper is a restriction of the model of [20], where the algorithm designer could choose an additional constant *bounding parameter*  $b \in \mathbb{Z}_{>0}$ , providing the nodes with the capability to count the number of ports holding message  $\sigma \in \Sigma$  up to  $b$ . In the current paper, the bounding parameter is set to  $b = 1$ . This model choice can be viewed as an asynchronous multi-frequency variant of the *beeping* communication model [11, 2].

Moreover, in contrast to the existing SA literature, the communication graph  $G = (V, E)$  assumed in the current paper may include *self-loops* of the form  $(v, v) \in E$  which means, in accordance with the definition of the SA model, that node  $v$  admits port  $\psi_v(v)$  that holds the last message received from itself. Using the terminology of the beeping model literature (see, e.g., [2]), the assumption that the communication graph is free of self-loops corresponds to a *sender collision detection*, whereas lifting this assumption means that node  $v$  may not necessarily distinguish its own transmitted message from those of its neighbors.

It turns out that self-loops have a significant effect on the power of SA algorithms. Indeed, while a SA algorithm that solves the *maximal independent set (MIS)* problem with probability 1 is presented in [20] under the assumption that the graph is free of self-loops, we prove in Section 3 that if the graph is augmented with self-loops, then no SA algorithm can solve this problem with a bounded failure probability. To distinguish between the original model of [20] and the one considered in the current paper, we hereafter denote the latter by  $SA^\circ$ .

<sup>3</sup> The only assumption we make on the event scheduling is FIFO message delivery: a message sent by node  $u$  at time  $t$  is written into port  $\psi_v(u)$  of its neighbor  $v$  before the message sent by  $u$  at time  $t' > t$ .

## 1.2 Results

Throughout, the number of nodes and the diameter of the graph  $G$  are denoted by  $n$  and  $D$ , respectively. We say that an event occurs *with high probability (whp)* if its probability is at least  $1 - n^{-c}$  for an arbitrarily large constant  $c$ . Our main technical contribution is cast in the following two theorems.

► **Theorem 1.** *For any constant  $k$ , there exists a  $\text{SA}^\circ$  algorithm that solves the  $k$ -leader selection problem in  $\tilde{O}(D)$  time whp.<sup>4</sup>*

► **Theorem 2.** *If the upper bound  $k$  on the number of candidates may grow as a function of  $n$ , then there does not exist a SA algorithm (operating on graphs with no self-loops) that solves the  $k$ -leader selection problem with a failure probability bounded away from 1.*

We emphasize that the failure probability of the  $\text{SA}^\circ$  algorithm promised in Theorem 1 (i.e., the probability that the algorithm selects multiple leaders or that it runs for more than  $\tilde{O}(D)$  time) is inverse polynomial in  $n$  even though each individual node does not (and cannot) possess any notion of  $n$  – to a large extent, this, together with the termination detection requirement, capture the main challenge in designing the promised algorithm.<sup>5</sup> The theorem assumes that  $k = O(1)$  and hides the dependency of the algorithm’s parameters on  $k$ . A closer look at its proof reveals that our  $\text{SA}^\circ$  algorithm uses local memory and messages of size  $O(\log k)$  bits. Theorem 2 asserts that the dependence of these parameters on  $k$  is unavoidable. Whether this dependence can be improved beyond  $O(\log k)$  remains an open question.

## 1.3 Additional Related Literature

As mentioned earlier, the SA model was introduced by Emek and Wattenhofer in [20] as an abstraction for distributed computing in networks of devices whose computation and communication capabilities are far weaker than those of a modern digital computer. Their main focus was on distributed problems that can be solved in sub-diameter (specifically,  $\log^{O(1)} n$ ) time including MIS, tree coloring, coloring bounded degree graphs, and maximal matching. This remained the case also in [19], where Emek and Uitto studied SA algorithms for the MIS problem in dynamic graphs. In contrast, the current paper considers the  $k$ -leader selection problem – an inherently global problem that requires  $\Omega(D)$  time.

Computational models based on networks of finite automata have been studied for many years. The best known such model is the extensively studied *cellular automata* that were introduced by Ulam and von Neumann [31] and became popular with Martin Gardner’s Scientific American column on Conway’s *game of life* [24] (see also [37]).

Another popular model that considers a network of finite automata is the *population protocols* model, introduced by Angluin et al. [5] (see also [6, 30]), where the network entities communicate through a sequence of atomic pairwise interactions controlled by a fair (adversarial or randomized) scheduler. This model provides an elegant abstraction for networks of mobile devices with proximity derived interactions and it also fits certain types of chemical reaction networks [18]. Some work on population protocols augments the model with

<sup>4</sup> The asymptotic notation  $\tilde{O}(\cdot)$  may hide  $\log^{O(1)} n$  factors.

<sup>5</sup> If we aim for a failure probability inverse polynomial in  $k$  (rather than  $n$ ) and we do not insist on termination detection, then the problem is trivially solved by the algorithm that simply assigns a random ID from a set of size  $k^{O(1)}$  to each candidate and then eliminates a candidate if it encounters an ID larger than its own.

a graph defined over the population's entities so that the pairwise interactions are restricted to graph neighbors, thus enabling some network topology to come into play. However, for the kinds of networks we are interested in, the fundamental assumption of sequential atomic pairwise interactions may provide the population protocol with unrealistic advantage over weaker message passing variants (including the SA model) whose communication schemes do not enable a node to interact with its individual neighbors independently. Furthermore, population protocols are typically required to *eventually converge* to a correct output and are allowed to return arbitrary (wrong) outputs beforehand, a significantly weaker requirement than the termination detection requirement considered in this paper.

The neat *amoebot model* introduced by Dolev et al. [17] also considers a network of finite automata in a (hexagonal) grid topology, but in contrast to the models discussed so far, the particles in this network are augmented with certain mobility capabilities, inspired by the amoeba contraction-expansion movement mechanism. Since its introduction, this model was successfully employed for the theoretical investigation of self-organizing particle systems [36, 15, 13, 16, 14, 10, 12], especially in the context of *programmable matter*.

Leader election is arguably the most fundamental problem in distributed systems coordination and has been extensively studied from the early days of distributed computing [23, 22]. It is synonymous in most models to the construction of a spanning tree – another fundamental problem in distributed computing – where the root is typically the leader. Leader election has many applications including deadlock detection, choosing a key/password distribution center, and implementing a distributed file system manager. It also plays a key role in tasks requiring a reliable centralized coordinating node, e.g., Paxos and Raft, where leader election is used for consensus – yet another fundamental distributed computing problem, strongly related to leader election. Notice that in our model, leader selection does not (and cannot) imply a spanning tree, but it does imply consensus.

Angluin [4] proved that uniform algorithms cannot break symmetry in a ring topology with success probability 1. Following this classic impossibility result, many symmetry breaking algorithms (with and without termination detection) that relax some of the assumptions in [4] were introduced [1, 7, 26, 33, 3]. Itai and Rodeh [26] were the first to design randomized leader election algorithms with bounded failure probability in a ring topology, assuming that the nodes know  $n$ . Schieber and Snir [33] and Afek and Matias [3] extended their work to arbitrary topology graphs.

## 2 SA<sup>○</sup> Algorithm for $k$ -Leader Selection

In this section, we present our SA<sup>○</sup> algorithm and establish Theorem 1. We start with some preliminary definitions and assumptions presented in Section 2.1. Sections 2.2 and 2.3 are dedicated to the basic subroutines on which our algorithm relies. The algorithm itself is presented in Section 2.4, where we also establish its correctness. Finally, in Section 2.5, we analyze the algorithm's run-time.

### 2.1 Preliminaries

As explained in Section 1.1, the execution in the SA (and SA<sup>○</sup>) model is controlled by an asynchronous scheduler. One of the contributions of [20] is a SA *synchronizer* implementation (cf. the  $\alpha$ -synchronizer of Awerbuch [8]). Given a synchronous SA algorithm  $\mathcal{A}$  whose execution progresses in fully synchronized *rounds*  $t \in \mathbb{Z}_{>0}$  (with simultaneous wake-up), the synchronizer generates a valid (asynchronous) SA algorithm  $\mathcal{A}'$  whose execution progresses in *pulses* such that the actions taken by  $\mathcal{A}'$  in pulse  $t$  are identical to those taken by  $\mathcal{A}$

in round  $t$ .<sup>6</sup> The synchronizer is designed so that the asynchronous algorithm  $\mathcal{A}'$  has the same bounding parameter  $b$  ( $= 1$  in the current paper) and asymptotic run-time as the synchronous algorithm  $\mathcal{A}$ .

Although the model considered by Emek and Wattenhofer [20] assumes that the graph has no self-loops, it is straightforward to apply their synchronizer to graphs that do include self-loops, hence it can work also in our  $\text{SA}^\circ$  model. Consequently, in what follows, we restrict our attention to synchronous  $\text{SA}^\circ$  algorithms. Specifically, we assume that the execution progresses in synchronous rounds  $t \in \mathbb{Z}_{>0}$ , where in round  $t$ , each node  $v$

- (1) receives the messages sent by its neighbors in round  $t - 1$ ;
- (2) updates its state; and
- (3) sends a message to its neighbors (same message to all neighbors).

Since we make no effort to optimize the size of the messages used by our algorithm, we assume hereafter that the message alphabet  $\Sigma$  is identical to the state space  $Q$  and that node  $v$  simply sends its current state to its neighbors at the end of every round. Nevertheless, for clarity of the exposition, we sometimes describe the algorithm in terms of sending designated messages, recalling that this simply means that the states of the nodes encode these messages.

To avoid cumbersome presentation, our algorithm's description does not get down to the resolution of the state space  $Q$  and transition function  $\tau$ . It is straightforward though to implement our algorithm as a randomized finite automaton, adhering to the model presented in Section 1.1. In this regard, at the risk of stating the obvious, we remind the reader that if  $k$  is a constant, then a finite automaton supports arithmetic operations modulo  $O(k)$ .

In the context of the  $k$ -leader selection problem, we use the verb *withdraw* when referring to a node that ceases to be a candidate.

## 2.2 The Ball Growing Subroutine

We present a generic *ball growing* subroutine in graph  $G = (V, E)$  with at most  $k$  candidates. The subroutine is initiated at (all) the candidates, not necessarily simultaneously, through designated signals discussed later on. During its execution, some candidates may withdraw; in the context of this subroutine, we refer to the surviving candidates as *roots*.

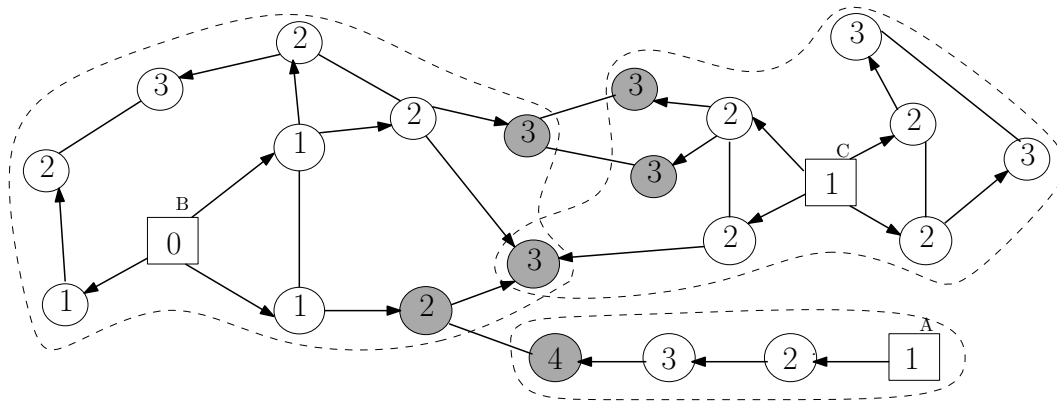
The ball growing subroutine assigns a *level* variable  $\lambda(v) \in \{0, 1, \dots, M - 1\}$  to each node  $v$ , where  $M = 2k + 2$ . Path  $P = (v_1, \dots, v_q)$  in  $G$  is called *incrementing* if  $\lambda(v_{j+1}) = \lambda(v_j) + 1 \pmod{M}$  for every  $1 \leq j \leq q - 1$ . The set of nodes reachable from a root  $r$  via an incrementing path is referred to as the *ball* of  $r$ , denoted by  $B(r)$ . We design this subroutine so that the following lemma holds.

► **Lemma 3.** *Upon termination of the ball growing subroutine,*

- (1) *every incrementing path is a shortest path (between its endpoints) in  $G$ ;*
- (2) *every root belongs to exactly one ball (its own); and*
- (3) *every non-root node belongs to at least one ball.*

**Intuition spotlight:** A natural attempt to design the ball growing subroutine is to grow a breadth first search tree around candidate  $r$ , layer by layer, so that node  $v$  at distance  $d$  from  $r$  is assigned with level variable  $\lambda(v) = d \pmod{M}$ . This is not necessarily

<sup>6</sup> We emphasize the role of the assumption that when the execution begins, the ports hold the designated initial message  $\sigma_0$ . Based on this assumption, a node can “sense” that some of its neighbors have not been activated yet, hence synchronization can be maintained right from the beginning.



■ **Figure 1** The result of a ball growing process invoked at candidate A in round 1, candidate B in round 2, and candidate C in round 3. The level variables  $\lambda(\cdot)$  are depicted by the numbers written inside the nodes and the balls are depicted by the dashed curves. The boundary nodes appear with a gray background. The DAG  $\vec{G}$  is depicted by the oriented edges.

possible though when multiple candidates exist: What happens if the ball growing processes of different candidates reach  $v$  in the same round? What happens if these ball growing processes reach several adjacent nodes in the same round? If we are not careful, these scenarios may lead to incrementing paths that are not shortest paths and even to cyclic incrementing paths. Things become even more challenging considering the weak communication capabilities of the nodes that may prevent them from distinguishing between the ball growing processes of different candidates.

The ball growing subroutine is implemented under the  $SA^\odot$  model by disseminating  $\mathbf{GrowBall}(\ell)$  messages,  $\ell \in \{0, 1, \dots, M - 1\}$ , throughout the graph. Consider a candidate  $r$  and let  $s(r)$  be the round in which it is signaled to invoke the ball growing subroutine. If  $r$  receives a  $\mathbf{GrowBall}(\cdot)$  message in some round  $t \leq s(r)$ , then  $r$  withdraws and subsequently follows the protocol like any other non-root node; otherwise,  $r$  becomes a root in round  $s(r)$ . If  $s(r)$  is even (resp., odd), then  $r$  assigns  $\lambda(r) \leftarrow 0$  (resp.,  $\lambda(r) \leftarrow 1$ ) and sends a  $\mathbf{GrowBall}(\lambda(r))$  message.

Consider a non-root node  $v$  and let  $g(v)$  be the first round in which it receives a  $\mathbf{GrowBall}(\cdot)$  message. Notice that  $v$  may receive several  $\mathbf{GrowBall}(\ell)$  messages with different arguments  $\ell$  in round  $g(v)$  – let  $L$  be the set of all such arguments  $\ell$ . Node  $v$  assigns  $\lambda(v) \leftarrow \ell'$  and sends a  $\mathbf{GrowBall}(\ell')$  message at the end of round  $g(v)$ , where  $\ell'$  is chosen to be any integer in  $\{0, 1, \dots, M - 1\}$  that satisfies:

- (i)  $\ell' - 1 \bmod M \in L$ ; and
- (ii)  $\ell' + 1 \bmod M \notin L$ .

This completes the description of the ball growing subroutine. Refer to Figure 1 for an illustration.

**Intuition spotlight:** Condition (i) ensures that  $v$  joins the ball  $B(r)$  of some root  $r$ . By condition (ii), nodes do not join  $B(r)$  “indirectly” (this could have led to incrementing paths that are not shortest paths).

**Proof of Lemma 3.** Consider a (root or non-root) node  $v \in V$  and let  $p(v)$  be the round in which  $v$  starts its active participation in the ball growing process. More formally, if  $v$  is a root (i.e., it is a candidate signaled to invoke the ball growing subroutine strictly before



receiving any `GrowBall`( $\cdot$ ) message), then  $p(v) = s(v)$ ; otherwise,  $p(v) = g(v)$ . The following properties are established by (simultaneous) induction on the rounds:

- In any round  $t \geq p(v)$ , variable  $\lambda(v)$  is even if and only if  $p(v)$  is even.
- In any round  $t \geq p(v)$ , node  $v$  has a neighbor  $u$  with  $\lambda(u) = \lambda(v) - 1 \pmod M$  if and only if  $v$  is not a root.
- In any round  $t \geq p(v)$ , node  $v$  belongs to ball  $B(r)$  for some root  $r$ .
- In any round  $t \geq p(v)$ , if  $v \in B(r)$  for some root  $r$ , then the incrementing path(s) that realize this relation are shortest paths in the graph.
- If  $u, v \in B(r)$  for some root  $r$  and  $p(u) = p(v)$ , then  $\lambda(u) = \lambda(v)$ .
- The total number of different arguments  $\ell$  in the `GrowBall`( $\ell$ ) messages sent during a single round is at most  $k$ .
- Non-root node  $v$  finds a valid value to assign to  $\lambda(v)$  in round  $g(v) = p(v)$ .

The assertion follows. ◀

► **Observation 4.** *If  $t$  is the earliest round in which the ball growing process is initiated at some candidate, then the process terminates by round  $t + O(D)$ .*

**Boundary Nodes.** We will see in Section 2.4 that our algorithm detects candidate multiplicity by identifying the existence of multiple balls in the graph. The key notion in this regard is the following one (see Figure 1): Node  $v$  is said to be a *boundary* node if

- (1)  $v \in B(r) \cap B(r')$  for roots  $r \neq r'$ ; or
- (2)  $v \in B(r)$  for some root  $r$  and there exists a neighbor  $v'$  of  $v$  such that  $v' \notin B(r)$ .

► **Observation 5.** *If the graph has multiple roots, then every ball includes at least one boundary node.*

Node  $v$  is said to be a *locally observable boundary* node if it has a neighbor  $v'$  such that  $\lambda(v') \notin \{\lambda(v) + \ell \pmod M \mid \ell = -1, 0, +1\}$ . Notice that by Lemma 3, there cannot be a ball that includes both  $v$  and  $v'$  since then, at least one of the incrementing paths that realize these inclusions is not a shortest path. Therefore, a locally observable boundary node is in particular a boundary node.

**The Directed Acyclic Graph  $\vec{G}$ .** Given two adjacent nodes  $u$  and  $v$ , we say that  $v$  is a *child* of  $u$  and that  $u$  is a *parent* of  $v$  if  $\lambda(v) = \lambda(u) + 1 \pmod M$ ; a childless node is referred to as a *leaf*. This induces an orientation on a subset  $F$  of the edges, say, from parents to their children (up the incrementing paths), thus introducing a directed graph  $\vec{G}$  whose edge set is an oriented version of  $F$  (see Figure 1). Lemma 3 guarantees that  $\vec{G}$  is acyclic (so, it is a directed acyclic graph, abbreviated *DAG*) and that it spans all nodes in  $V$ . Moreover, the sources and sinks of  $\vec{G}$  are exactly the roots and leafs of the ball growing subroutine, respectively, and the source-to-sink distances in  $\vec{G}$  are upper-bounded by the diameter  $D$  of  $G$ .

We emphasize that the in-degrees and out-degrees in  $\vec{G}$  are unbounded. Nevertheless, the simplifying assumption that the messages sent by the nodes encode their local states, including the level variables  $\lambda(\cdot)$  (see Section 2.1), ensures that node  $v$  can distinguish between messages received from its children, messages received from its parents, and messages received from nodes that are neither children nor parents of  $v$ .



## 2.3 Broadcast and Echo over $\vec{G}$

The assignment of level variables  $\lambda(\cdot)$  by the ball growing subroutine and the child-parent relations these variables induce provide a natural infrastructure for *broadcast and echo* ( $B\&E$ ) over the aforementioned DAG  $\vec{G}$  so that the broadcast (resp., echo) process progresses up (resp., down) the incrementing paths. These are implemented based on **Broadcast** and **Echo** messages as follows.

The broadcast subroutine is initiated at (all) the roots, not necessarily simultaneously, through designated signals discussed later on and root  $r$  becomes *broadcast ready* upon receiving such a signal. A non-root node  $v$  becomes broadcast ready in the first round in which it receives **Broadcast** messages from all its parents. A (root or non-root) node  $v$  that becomes broadcast ready in round  $t_0^b = t_0^b(v)$  keeps sending **Broadcast** messages throughout the round interval  $[t_0^b, t_1^b)$ , where  $t_1^b = t_1^b(v)$  is defined to be the first round (strictly) after  $t_0^b$  in which

- (i)  $v$  receives **Broadcast** messages from all its children; and
- (ii)  $v$  does not receive a **Broadcast** message from any of its parents.

(Notice that conditions (i) and (ii) are satisfied vacuously for the leaves and roots, respectively.)

The echo subroutine is implemented in a reversed manner: It is initiated at (all) the leaves, not necessarily simultaneously, after their role in the broadcast subroutine ends so that leaf  $v$  becomes *echo ready* in round  $t_1^e(v)$ . A non-leaf node  $v$  becomes *echo ready* in the first round in which it receives **Echo** messages from all its children. A (leaf or non-leaf) node  $v$  that becomes echo ready in round  $t_0^e = t_0^e(v)$  keeps sending **Echo** messages throughout the round interval  $[t_0^e, t_1^e)$ , where  $t_1^e = t_1^e(v)$  is defined to be the first round (strictly) after  $t_0^e$  in which

- (i)  $v$  receives **Echo** messages from all its parents; and
- (ii)  $v$  does not receive an **Echo** message from any of its children.

(Notice that conditions (i) and (ii) are satisfied vacuously for the roots and leaves, respectively.)

► **Lemma 6.** *The following properties hold for every  $B\&E$  process:*

- *Rounds  $t_0^b(v)$ ,  $t_1^b(v)$ ,  $t_0^e(v)$ , and  $t_1^e(v)$  exist and  $t_0^b(v) < t_1^b(v) \leq t_0^e(v) < t_1^e(v)$  for every node  $v$ .*
- *If node  $v$  is reachable from node  $u \neq v$  in DAG  $\vec{G}$ , then  $t_i^b(u) < t_i^b(v)$  and  $t_i^e(u) > t_i^e(v)$  for  $i \in \{0, 1\}$ .*
- *If  $t$  is the latest round in which the process is initiated at some root, then the process terminates by round  $t + O(D)$ .*

**Proof.** Follows since  $\vec{G}$  is a DAG and all paths in  $\vec{G}$  are shortest paths. ◀

**Auxiliary Conditions.** In the aforementioned implementation of the broadcast (resp., echo) subroutine, being broadcast (resp., echo) ready is both a necessary and sufficient condition for a node to start sending **Broadcast** (resp., **Echo**) messages. In Section 2.4, we describe variants of this subroutine in which being broadcast (resp., echo) ready is a necessary, but not necessarily sufficient, condition and the node starts sending **Broadcast** (resp., **Echo**) messages only after additional conditions, referred to later on as *auxiliary conditions*, are satisfied.

**Acknowledged Ball Growing.** As presented in Section 2.2, the ball growing subroutine propagates from the roots to the leaves. To ensure that root  $r$  is signaled when the construction of its ball  $B(r)$  has finished (cf. termination detection),  $r$  initiates a B&E process one round after it invokes the ball growing subroutine. The valid operation of this process is guaranteed

since the ball growing process propagates at least as fast as the B&E process. We call the combined subroutine *acknowledged ball growing*.

## 2.4 The Main Algorithm

Our  $k$ -leader selection algorithm consists of two *phases* executed repeatedly in alternation:

- phase 0, a.k.a. the *detection* phase, that detects the existence of multiple candidates whp; and
- phase 1, a.k.a. the *elimination* phase, in which all candidates but one withdraw with probability at least  $1/4$ .

Starting with a detection phase, the algorithm executes the phases in alternation until the first detection phase that does not detect candidate multiplicity. Each node  $v$  maintains a *phase* variable  $\phi(v) \in \{0, 1\}$  that indicates  $v$ 's current phase.

The two phases follow a similar structure: The (surviving) candidates start by initiating an acknowledged ball growing process. Among its other “duties”, this ball growing process is responsible for updating the phase variables  $\phi(\cdot)$  of the nodes: node  $v$  with  $\phi(v) = p$  that receives a **GrowBall**( $\cdot$ ) message from node  $u$  with  $\phi(u) = p + 1 \bmod 2$  assigns  $\phi(v) \leftarrow p + 1 \bmod 2$ . When updating the phase variable  $\phi(v)$  to  $\phi(v) = p + 1 \bmod 2$ , node  $v$  ceases to participate in phase  $p$ , resetting all phase  $p$  variables. Recalling the definition of the ball growing subroutine (see Section 2.2), this means in particular that if a candidate  $r$  with  $\phi(r) = p$  receives a **GrowBall**( $\cdot$ ) message from node  $u$  with  $\phi(u) = p + 1 \bmod 2$ , then  $r$  withdraws and subsequently follows the protocol like any other non-root node.

**Intuition spotlight:** The ball growing process of phase  $p + 1 \bmod 2$  essentially “takes control” over the graph and “forcibly” terminates phase  $p$  (at nodes where it did not terminate already). We design the algorithm to ensure that at any point in time, there is at most one  $p$  value for which there is an ongoing ball growing process in the graph (otherwise, we may get to undesired situations such as all candidates withdrawing).

Upon termination of the acknowledged ball growing process, the roots run  $2k$  back-to-back *B&E iterations*, initiating the broadcast process of the next B&E iteration one round after the echo process of the previous B&E iteration terminates (the choice of the parameter  $2k$  will become clear soon). Each node  $v$  maintains a variable  $\iota(v) \in \{0, 1, \dots, 2k\}$  that stores  $v$ 's current B&E iteration. This variable is initialized to  $\iota(v) \leftarrow 0$  during the acknowledged ball growing process (considered hereafter as B&E iteration 0) and incremented subsequently from  $i - 1$  to  $i$  when  $v$  becomes broadcast ready in B&E iteration  $i$  (see Section 2.3). A phase ends when the echo process of B&E iteration  $2k$  terminates.

The  $\iota(\cdot)$  variables may differ across the graph and to keep the B&E iterations in synchrony, we augment the B&E subroutines with the following auxiliary conditions (see Section 2.3): Node  $v$  with  $\iota(v) = i$  (i.e., in B&E iteration  $i$ ) does not start to send **Broadcast** (resp., **Echo**) messages as long as it has a non-child (resp., non-parent) neighbor  $u$  with  $\iota(u) = i - 1$ .<sup>7</sup> We emphasize that this includes neighbors  $u$  that are neither children nor parents of  $v$ .

For the sake of the next observation, we globally map the B&E iterations to *sequence numbers* so that B&E iterations  $0, 1, \dots, 2k$  of the first phase (which is a detection phase) are mapped to sequence numbers  $1, 2, \dots, 2k + 1$ , respectively, B&E iterations  $0, 1, \dots, 2k$  of the second phase (which is an elimination phase) are mapped to sequence numbers  $2k + 2, 2k + 3, \dots, 4k + 2$ , respectively, and so on. Let  $\sigma(v)$  be a variable (defined only for the sake of the analysis) indicating the sequence number of node  $v$ 's current B&E iteration.

<sup>7</sup> This can be viewed as imposing the  $\alpha$ -synchronizer of [8] on the B&E iterations of the balls.

► **Observation 7.** *For every two roots  $r$  and  $r'$ , we have  $|\sigma(r) - \sigma(r')| \leq k - 1$ .*

We say that round  $t$  is *0-dirty* (resp., *1-dirty*) if some node  $v$  with  $\phi(v) = 0$  (resp.,  $\phi(v) = 1$ ) sends a **GrowBall**( $\cdot$ ) message in round  $t$ ; the round is said to be *clean* if it is neither 0-dirty nor 1-dirty. Observation 7 implies that if  $\phi(r) = p$  and  $\iota(r) = k$  for some root  $r$  in round  $t$ , then  $\phi(r') = p$  and  $1 \leq \iota(r') \leq 2k - 1$  for any other root  $r'$  in round  $t$ , hence the ball growing process of this phase has already ended and the ball growing process of the next phase has not yet started.

► **Corollary 8.** *Let  $t_0$  and  $t_1$  be some 0-dirty and 1-dirty rounds, respectively. If  $t_0 \leq t_1$  (resp.,  $t_1 \leq t_0$ ), then there exists some  $t_0 < t' < t_1$  (resp.,  $t_1 < t' < t_0$ ) such that round  $t'$  is clean.*

### 2.4.1 The Detection Phase

In the detection phase, the nodes test for candidate multiplicity in the graph. If the graph contains a single candidate  $r$ , then the algorithm terminates upon completion of this phase and  $r$  is declared to be the leader. Otherwise, certain boundary nodes (see Section 2.2) realize whp that multiple balls exist in their neighborhoods and signal the roots that they should proceed to the elimination phase (rather than terminate the algorithm) upon completion of the current detection phase. This signal is carried by **Proceed** messages delivered from the boundary nodes to the roots of their balls down the incrementing paths in conjunction with the **Echo** messages of the (subsequent) B&E iterations.

For the actual candidate multiplicity test, once all nodes in the (inclusive) neighborhood of node  $v$  participate in the detection phase, node  $v$  checks if it is a locally observable boundary node and triggers a **Proceed** message delivery if it is. As the name implies, this check can be performed (locally) under the  $SA^\circ$  model assuming that the messages sent by the nodes encode their local states, including the level variables.

**Intuition spotlight:** Although every locally observable boundary node is a boundary node, not all boundary nodes are locally observable: a node may belong to several different balls or two adjacent nodes with the same level variable may belong to different balls. For this kind of scenarios, randomness is utilized to break symmetry between the candidates and identify (some of) the boundary nodes.

Consider some root  $r$  with  $\phi(r) = 0$  upon termination of the acknowledged ball growing subroutine and recall that at this stage,  $r$  runs  $2k$  back-to-back B&E iterations. In each round of these  $2k$  B&E iterations,  $r$  picks some *symbol*  $s$  uniformly at random (and independently of all other random choices) from a sufficiently large (yet constant size) symbol space  $\mathcal{S}$  and sends a **RandSymbol**( $s$ ) message. This can be viewed as a random symbol *stream*  $S_r \in \mathcal{S}^*$  that  $r$  generates, round by round, and sends to its children.

The random symbol streams  $S_r$  are disseminated throughout  $B(r)$  and utilized by the nodes (the boundary nodes in particular) to test for candidate multiplicity. For clarity of the exposition, it is convenient to think of a node  $v$  that does not send a **RandSymbol**( $s$ ) message,  $s \in \mathcal{S}$ , as if it sends a **RandSymbol**( $\perp$ ) message for the default symbol  $\perp \notin \mathcal{S}$ . The mechanism in charge of disseminating  $S_r$  up the incrementing paths works as follows: If non-root node  $v$  with  $\phi(v) = 0$  receives **RandSymbol**( $s$ ) messages with the same argument  $s$  from all its parents at the beginning of round  $t$ , then  $v$  sends a **RandSymbol**( $s$ ) message at the end of round  $t$ ; in all other cases,  $v$  sends a **RandSymbol**( $\perp$ ) message.

Throughout this process, each node  $v$  verifies that

- (1) all  $\text{RandSymbol}(s)$  messages sent by  $v$ 's parents in round  $t$  carry the same argument  $s$ ; and
- (2) any  $\text{RandSymbol}(s)$  message sent by a neighbor  $u$  of  $v$  with  $\lambda(u) = \lambda(v)$  in round  $t$  carries the same argument  $s$  as in the  $\text{RandSymbol}(s)$  message that  $v$  sends in round  $t$  (this is checked by  $v$  in round  $t + 1$ ).

If any of these two conditions does not hold, then  $v$  triggers a **Proceed** message delivery. A root that completes all  $2k$  B&E iterations in the detection phase without receiving any **Proceed** message terminates the algorithm and declares itself as the leader.

**Intuition spotlight:** Since the aforementioned random tests should detect candidate multiplicity whp (i.e., with error probability inverse polynomial in  $n$ ) and since the size of the symbol space  $\mathcal{S}$  from which the random symbol streams  $S_r$  are generated is bounded, it follows that the length of the random symbol streams must be  $|S_r| \geq \Omega(\log n)$ . How can we ensure that  $|S_r| \geq \Omega(\log n)$  if the nodes cannot count beyond some constant?

To ensure that the random symbol stream  $S_r$  is sufficiently long, we augment the echo subroutine invoked during B&E iteration  $k$  of the detection phase (out of the  $2k$  B&E iterations in this phase) with one additional auxiliary condition referred to as the *geometric auxiliary condition*: Consider some node  $v$  with  $\phi(v) = 0$  and  $\iota(v) = k$  (i.e., in the  $k$ -th B&E iteration of the detection phase) and suppose that it becomes echo ready (for B&E iteration  $k$ ) in round  $t_0$ . Then,  $v$  tosses a fair coin  $c(t) \in_r \{0, 1\}$  in each round  $t \geq t_0$  until the first round  $t'$  for which  $c(t') = 1$ ; node  $v$  does not send **Echo** messages until round  $t'$ . This completes the description of the detection phase.

► **Lemma 9.** *If multiple roots start a detection phase, then all of them receive a **Proceed** message before completing their (respective)  $2k$  B&E iterations whp.*

**Intuition spotlight:** The proof's outline is as follows. We use the geometric auxiliary conditions to argue that there exists some root that spends  $\Omega(\log n)$  rounds in B&E iteration  $k$  whp. Employing Observation 7, we conclude that the random symbol stream generated by every root  $r$  is  $\Omega(\log n)$ -long whp. Conditioned on that, we prove that there exists some boundary node  $v \in B(r)$  that triggers a **Proceed** message delivery whp and that the corresponding **Proceed** message is delivered to  $r$  before the phase ends.

**Proof of Lemma 9.** Fix some detection phase. For a root  $r$ , let  $c_r$  be the number of rounds  $r$  spends in B&E iterations  $1, 2, \dots, 2k - 1$ , that is, the number of rounds in which  $1 \leq \iota(r) \leq 2k - 1$  (during this detection phase). We first argue that  $c_r \geq \Omega(\log n)$  for all roots  $r$  whp. To that end, let  $X_v$  be the number of rounds in which node  $v$  is prevented from sending its **Echo** messages in B&E iteration  $k$  due to the geometric auxiliary condition ( $t' - t_0$  in the aforementioned notation of the geometric auxiliary condition) and notice that this auxiliary condition is designed so that  $X_v$  is a geometric random variable with parameter  $1/2$ . Therefore,

$$\Pr \left( \bigwedge_{v \in V} X_v < \log(n)/2 \right) = \left( 1 - 2^{-\log(n)/2} \right)^n = \left( 1 - 1/\sqrt{n} \right)^n \leq e^{-\sqrt{n}}.$$

Condition hereafter on the event that  $X_{v^*} \geq \log(n)/2$  for some node  $v^*$ , namely,  $v^*$  is prevented from sending its **Echo** messages (in B&E iteration  $k$ ) for at least  $\log(n)/2 = \Omega(\log n)$  rounds. Let  $r^*$  be a root such that  $v^* \in B(r^*)$ . By the definition of auxiliary

conditions, B&E iteration  $k$  of  $r^*$  takes at least  $\Omega(\log n)$  rounds. Observation 7 guarantees that by the time  $r^*$  starts B&E iteration  $k$ , every other root must have already started B&E iteration 1 (of this detection phase). Moreover, no root can start B&E iteration  $2k$  before  $r^*$  finishes B&E iteration  $k$ . We conclude that every root  $r$  spends at least  $\Omega(\log n)$  rounds in B&E iterations  $1, 2, \dots, 2k - 1$ , thus establishing the argument.

Let  $Z_r$  be the prefix of the random symbol stream  $S_r$  generated by root  $r$  during the first  $c_r - 1$  rounds it spends in B&E iterations  $1, 2, \dots, 2k - 1$ , i.e., during all but the last round of these B&E iterations (the reason for this missing round is explained soon), and let  $z_r = |Z_r|$ . We have just showed that  $z_r = c_r - 1 \geq \Omega(\log n)$  for all roots  $r$  whp.

The assertion is established by proving that if multiple roots  $r$  exist in the graph and  $z_r \geq \Omega(\log n)$  for all of them, then for every root  $r$ , there exists some node  $v \in B(r)$  that triggers a **Proceed** message delivery while  $\iota(v) \leq 2k - 1$  whp. Indeed, if the **Proceed** message delivery is triggered by  $v$  while  $\iota(v) \leq 2k - 1$ , then a **Proceed** message is delivered to  $r$  with the **Echo** messages of B&E iteration  $2k$  at the latest, thus  $r$  does not terminate the algorithm at the end of this detection phase and by the union bound, this holds simultaneously for all roots  $r$  whp.

To that end, recall that node  $v$  sends a **RandSymbol**( $s$ ) message with some symbol  $s \in \mathcal{S} \cup \{\perp\}$  in every round of the detection phase. In the scope of this proof, we say that  $v$  *posts* the symbol stream  $(s_1, \dots, s_z)$  in rounds  $t_1, \dots, t_z$  if  $s_j$  is the argument of the **RandSymbol**( $\cdot$ ) message sent by  $v$  in round  $t_j$  for every  $1 \leq j \leq z$ .

Consider some root  $r$  and let  $v$  be a boundary node in  $B(r)$  that minimizes the distance to  $r$ . If  $v$  is locally observable, then it triggers a **Proceed** message delivery (deterministically) already when  $\iota(v) = 0$ , so assume hereafter that  $v$  is not locally observable. Let  $Q$  be an incrementing  $(r, v)$ -path and denote the length of  $Q$  by  $q$ . Taking  $\hat{t}$  to be the round in which B&E iteration 1 of  $r$  begins, recall that  $r$  posts  $Z_r$  in rounds  $\hat{t}, \hat{t} + 1, \dots, \hat{t} + z_r - 1$ . The choice of  $v$  ensures that all nodes of  $Q$  other than  $v$  are not boundary nodes, therefore if  $q \geq 1$  (i.e., if  $v \neq r$ ), then the node that precede  $v$  along  $Q$  – denote it by  $u$  – posts  $Z_r$  in rounds  $\hat{t} + q - 1, \hat{t} + q, \dots, \hat{t} + q + z_r - 2$ . Moreover, by the definition of  $Z_r$ , specifically, by the choice of  $z_r = c_r - 1$ , we know that  $0 \leq \iota(v) \leq 2k - 1$  (and  $\phi(v) = 0$ ) in all rounds  $\hat{t} \leq t \leq \hat{t} + q + z_r$ .

If  $v$  belongs to multiple balls, which necessarily means that  $v \neq r$  and  $q \geq 1$  (see Lemma 3), then  $v$  has another parent  $u' \neq u$  such that  $u' \in B(r')$  for some root  $r' \neq r$ . The probability that  $u'$  posts  $Z_r$  in rounds  $\hat{t} + q - 1, \hat{t} + q, \dots, \hat{t} + q + z_r - 2$  is at most  $|\mathcal{S}|^{-z_r}$ . Otherwise, if  $v$  belongs only to ball  $B(r)$ , then all its parents post  $Z_r$  in rounds  $\hat{t} + q - 1, \hat{t} + q, \dots, \hat{t} + q + z_r - 2$  (this holds vacuously if  $q = 0$  and  $v = r$  has no parents), thus  $v$  posts  $Z_r$  in rounds  $\hat{t} + q, \hat{t} + q + 1, \dots, \hat{t} + q + z_r - 1$ . Since  $v$  is a non-locally observable boundary node (that belongs exclusively to ball  $B(r)$ ), it must have a neighbor  $v'$  with  $\lambda(v') = \lambda(v)$  such that  $v' \notin B(r)$ . The probability that  $v'$  posts  $Z_r$  in rounds  $\hat{t} + q, \hat{t} + q + 1, \dots, \hat{t} + q + z_r - 1$  is at most  $|\mathcal{S}|^{-z_r}$  as well. Therefore, the probability that  $v$  does not trigger a **Proceed** message delivery while  $\iota(v) \leq 2k - 1$  is upper-bounded by  $|\mathcal{S}|^{-z_r}$  which completes the proof since  $z_r \geq \Omega(\log n)$  and since  $|\mathcal{S}|$  is an arbitrarily large constant. ◀

## 2.4.2 The Elimination Phase

In the elimination phase, each candidate  $r$  picks a *priority*  $\pi(r)$  uniformly at random (and independently) from a totally ordered priority space  $\mathcal{P}$ ; a candidate whose priority is (strictly) smaller than  $\pi_{\max} = \max_r \pi(r)$  is withdrawn. Taking the priority space to be  $\mathcal{P} = \{1, \dots, k\}$ , it follows by standard balls-in-bins arguments that the probability that exactly one candidate picks priority  $k$ , which implies that exactly one candidate survives, is at least  $1/4$  (in fact, it tends to  $1/4$  as  $k \rightarrow \infty$ ).

**Intuition spotlight:** The priorities of the candidates are disseminated in the graph so that candidate  $r$  withdraws if it encounters a priority  $\pi > \pi(r)$ . This is implemented on top of the ball growing subroutine invoked at the beginning of the elimination phase so that the ball growing process of root  $r$  “consumes” the ball of root  $r'$  if  $\pi(r) > \pi(r')$ , eventually reaching  $r'$  and instructing it to withdraw. The structure of the phase (specifically, the  $2k$  B&E iterations that follow the ball growing process) guarantees that only roots  $r$  with  $\pi(r) = \pi_{\max}$  reach the end of the phase (without being withdrawn).

We augment the ball growing subroutine invoked at the beginning of the elimination phase with the following mechanism: When candidate  $r$  is signaled to invoke the ball growing subroutine (so that it becomes a root), it appends its priority  $\pi(r)$  to the `GrowBall( $\cdot$ )` message it sends. A non-root node  $v$  that joins the ball of  $r$  records  $r$ 's priority in variable  $\pi(v) \leftarrow \pi(r)$ . A (root or non-root) node  $v$  that receives a `GrowBall( $\cdot$ )` message with priority (strictly) larger than  $\pi(v)$ , behaves as if this is the first `GrowBall( $\cdot$ )` message it receives in this phase. In particular,  $v$  resets all the variables of this phase and (re-)joins a ball from scratch. If  $v$  is a root, then it also withdraws.

Notice that Observation 7 still holds for the aforementioned augmented implementation of the ball growing subroutine. Therefore, when root  $r$  reaches B&E iteration  $k$ , i.e.,  $\iota(r) = k$ , all other roots  $r'$  are in some B&E iteration  $1 \leq \iota(r') \leq 2k - 1$  which means that there is no “active” ball growing processes in the graph, that is, the current round is clean (of `GrowBall( $\cdot$ )` messages). Since a candidate  $r$  with  $\pi(r) < \pi_{\max}$  is certain to be withdrawn by some `GrowBall( $\cdot$ )` message appended with priority  $\pi > \pi(r)$ , we obtain the following observation.

► **Observation 10.** *If root  $r$  completes its  $2k$  B&E iterations in an elimination phase, then with probability at least  $1/4$ , no other candidates exist in the graph.*

## 2.5 Run-Time

The correctness of our algorithm follows from Lemma 9 and Observation 10. To establish Theorem 1, it remains to analyze the algorithm’s run-time.

The first thing to notice in this regard is that the geometric auxiliary condition does not slow down the  $k$ -th iteration of the detection phase by more than an  $O(\log n)$  factor whp. Combining Observation 4 with Lemma 6, we can prove by induction on the phases that the  $j$ -th phase (for  $j \leq n^{O(1)}$ ) ends by round  $O(D(k + \log n))$  whp, which is  $O(D \log n)$  assuming that  $k$  is fixed. The analysis is completed due to Observation 10 ensuring that the algorithm terminates after  $O(\log n)$  elimination phases whp.

## 3 Negative Results

We now turn to establish some negative results that demonstrate the necessity of the assumption that  $k = O(1)$ . Our attention in this section is restricted to SA and SA<sup>○</sup> algorithms operating under a fully synchronous scheduler on graph families  $\{L_n\}_{n \geq 1}$  and  $\{L_n^\circ\}_{n \geq 1}$ , where  $L_n$  is a simple path of  $n$  nodes and  $L_n^\circ$  is  $L_n$  augmented with self-loops.

The main lemma established in this section considers the  $k$ -candidate binary consensus problem, a version of the classic binary consensus problem [21]. In this problem, each node  $v$  gets a binary input  $\text{in}(v) \in \{0, 1\}$  and returns a binary output  $\text{out}(v) \in \{0, 1\}$  under the following two constraints: (1) all nodes return the same output; and (2) if the nodes return output  $b \in \{0, 1\}$ , then there exists some node  $v$  such that  $\text{in}(v) = b$ . In addition, at most  $k$  (and at least 1) nodes are initially marked as candidates (thus distinguished from the rest of



the nodes). We emphasize that the marked candidates do not affect the validity of the output. Since a  $k$ -leader selection algorithm clearly implies a  $k$ -candidate binary consensus algorithm, Theorem 2 is established by proving Lemma 11. Note that the proof of this lemma is based on a probabilistic indistinguishability argument, similar to those used in many distributed computing negative results, starting with the classic result of Itai and Rodeh [26].

► **Lemma 11.** *If the upper bound  $k$  on the number of candidates may grow as a function of  $n$ , then there does not exist a SA algorithm that solves the  $k$ -candidate binary consensus problem on the graphs in  $\{L_n\}_{n \geq 1}$  with a failure probability bounded away from 1.*

**Proof.** Assume by contradiction that there exists such an algorithm  $\mathcal{A}$  and let  $\Sigma$  denote its message alphabet. For  $b = 0, 1$ , consider the execution of  $\mathcal{A}$  on an instance that consists of path  $L_2$ , where node  $v_1$  is a candidate, node  $v_2$  is not a candidate, and  $\text{in}(v_1) = \text{in}(v_2) = b$ . By definition, there exist constants  $p_b > 0$  and  $\ell_b$  and message sequences  $S_{b,1}, S_{b,2} \in \Sigma^{\ell_b}$  such that when  $\mathcal{A}$  runs on this instance, with probability at least  $p_b$ , node  $v_j$ ,  $j \in \{1, 2\}$ , reads message  $S_{b,j}(t)$  in its (single) port in round  $t = 1, \dots, \ell_b$  and outputs  $\text{out}(v_j) = b$  at the end of round  $\ell_b$ .

Now, consider graph  $L_n$  for some sufficiently large  $n$  (whose value will be determined later on) and consider a subgraph of  $L_n$ , referred to as a  $Q_b$ -gadget, that consists of  $2\ell_b + 2$  contiguous nodes  $v_1, \dots, v_{2\ell_b+2}$  of the underlying path  $L_n$ , all of which receive input  $\text{in}(v_i) = b$ . Moreover, the nodes  $v_1, \dots, v_{2\ell_b+2}$  are marked as candidates in an alternating fashion so that if  $v_i$  is a candidate, then  $v_{i+1}$  is not a candidate, constrained by the requirement that  $v_{\ell_b+1}$  is a candidate (and  $v_{\ell_b+2}$  is not). The key observation is that when  $\mathcal{A}$  runs on  $L_n$ , with probability at least  $q_b = p_b^{2\ell_b+2}$ , the nodes  $v_{\ell_b+1}$  and  $v_{\ell_b+2}$  of the  $Q_b$ -gadget read messages  $S_{b,1}(t)$  and  $S_{b,2}(t)$ , respectively, in (all) their ports in round  $t = 1, \dots, \ell_b$  and output  $b$  at the end of round  $\ell_b$ , independently of the random bits of the nodes outside the  $Q_b$ -gadget.

Fix  $\ell = \ell_0 + \ell_1 + 2$  and define a  $Q$ -gadget to be a subgraph of  $L_n$  that consists of a  $Q_0$ -gadget appended to a  $Q_1$ -gadget, so, in total, the  $Q$ -gadget is a (sub)path that contains  $2\ell_0 + 2\ell_1 + 4 = 2\ell$  nodes,  $\ell$  of which are candidates. Following the aforementioned observation, when  $\mathcal{A}$  runs on  $L_n$ , with probability at least  $q = q_0 \cdot q_1$ , some nodes in the  $Q$ -gadget output 0 and others output 1; we refer to this (clearly invalid) output as a *failure* event of the  $Q$ -gadget.

Since  $p_0, p_1, \ell_0$ , and  $\ell_1$  are constants that depend only on  $\mathcal{A}$ ,  $\ell = \ell_0 + \ell_1 + 2$ ,  $q_0 = p_0^{2\ell_0+2}$  and  $q_1 = p_1^{2\ell_1+2}$  are also constants that depend only on  $\mathcal{A}$ , and thus  $q = q_0 \cdot q_1$  is also a constant that depends only on  $\mathcal{A}$ . Take  $z$  to be an arbitrarily large constant. If  $n$  is sufficiently large, then we can embed  $y = \lceil z/q \rceil$  pairwise disjoint  $Q$ -gadgets in  $L_n$ . Indeed, these  $Q$ -gadgets account to a total of  $\ell \cdot y$  candidates and recalling that  $z, q$ , and  $\ell$  are constants, this number is smaller than  $k = k(n)$  for sufficiently large  $n$ . When  $\mathcal{A}$  runs on  $L_n$ , each of these  $y$   $Q$ -gadgets fails with probability at least  $q$  (independently). Therefore, the probability that all nodes return the same binary output is at most  $(1 - q)^y$ . The assertion follows since this expression tends to 0 as  $y \rightarrow \infty$  which is obtained as  $z \rightarrow \infty$ . ◀

The proof of Lemma 11 essentially shows that no SA algorithm can distinguish between  $L_2$  and  $L_n$  with a bounded failure probability. When the path is augmented with self-loops, we can use a very similar line of arguments to show that no  $\text{SA}^\circ$  algorithm can distinguish between  $L_1^\circ$  and  $L_n^\circ$  with a bounded failure probability. This allows us to establish the following lemma that should be contrasted with the SA MIS algorithm of [20] that works on general topology graphs (with no self-loops) and succeeds with probability 1.

► **Lemma 12.** *There does not exist a  $\text{SA}^\circ$  algorithm that solves the MIS problem on the graphs in  $\{L_n^\circ\}_{n \geq 1}$  with a failure probability bounded away from 1.*

---

**References**

---

- 1 Karl R. Abrahamson, Andrew Adler, Lisa Higham, and David G. Kirkpatrick. Probabilistic solitude verification on a ring. In *Proceedings of ACM Symposium on Principles of Distributed Computing (PODC)*, pages 161–173, 1986.
- 2 Yehuda Afek, Noga Alon, Ziv Bar-Joseph, Alejandro Cornejo, Bernhard Haeupler, and Fabian Kuhn. Beeping a maximal independent set. In *Proceedings of International Symposium on Distributed Computing (DISC)*, pages 32–50, 2011.
- 3 Yehuda Afek and Yossi Matias. Elections in anonymous networks. *Inf. Comput.*, 113(2):312–330, 1994.
- 4 Dana Angluin. Local and global properties in networks of processors (extended abstract). In *Proceedings of ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 82–93, 1980.
- 5 Dana Angluin, James Aspnes, Zoë Diamadi, Michael J. Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed Computing*, 18(4):235–253, 2006.
- 6 James Aspnes and Eric Ruppert. *An Introduction to Population Protocols*, pages 97–120. Springer Berlin Heidelberg, 2009.
- 7 Hagit Attiya, Marc Snir, and Manfred K. Warmuth. Computing on an anonymous ring. *J. ACM*, 35(4):845–875, 1988.
- 8 Baruch Awerbuch. Complexity of network synchronization. *J. ACM*, 32(4):804–823, 1985.
- 9 Baruch Awerbuch. Optimal distributed algorithms for minimum weight spanning tree, counting, leader election, and related problems. In *Proceedings of ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 230–240, 1987.
- 10 Sarah Cannon, Joshua J. Daymude, Dana Randall, and Andréa W. Richa. A Markov chain algorithm for compression in self-organizing particle systems. In *Proceedings of ACM Symposium on Principles of Distributed Computing (PODC)*, pages 279–288, 2016.
- 11 Alejandro Cornejo and Fabian Kuhn. Deploying wireless networks with beeps. In *Proceedings of International Symposium on Distributed Computing (DISC)*, pages 148–162, 2010.
- 12 Joshua J. Daymude, Zahra Derakhshandeh, Robert Gmyr, Alexandra Porter, Andréa W. Richa, Christian Scheideler, and Thim Strothmann. On the runtime of universal coating for programmable matter. *Natural Computing*, 17(1):81–96, 2018.
- 13 Zahra Derakhshandeh, Robert Gmyr, Andréa W. Richa, Christian Scheideler, and Thim Strothmann. An algorithmic framework for shape formation problems in self-organizing particle systems. In *Proceedings of International Conference on Nanoscale Computing and Communication (NANOCOM)*, pages 21:1–21:2, 2015.
- 14 Zahra Derakhshandeh, Robert Gmyr, Andréa W. Richa, Christian Scheideler, and Thim Strothmann. Universal shape formation for programmable matter. In *Proceedings of ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 289–299, 2016.
- 15 Zahra Derakhshandeh, Robert Gmyr, Andréa W. Richa, Christian Scheideler, Thim Strothmann, and Shimrit Tzur-David. Infinite object coating in the Amoebot model. *CoRR*, abs/1411.2356, 2014. [arXiv:1411.2356](https://arxiv.org/abs/1411.2356).
- 16 Zahra Derakhshandeh, Robert Gmyr, Thim Strothmann, Rida Bazzi, Andréa W. Richa, and Christian Scheideler. Leader election and shape formation with self-organizing programmable matter. In *Proceedings of International Conference on DNA Computing and Molecular Programming (DNA)*, pages 117–132, 2015.
- 17 Shlomi Dolev, Robert Gmyr, Andréa W. Richa, and Christian Scheideler. Ameba-inspired self-organizing particle systems. *CoRR*, abs/1307.4259, 2013. [arXiv:1307.4259](https://arxiv.org/abs/1307.4259).
- 18 David Doty. Timing in chemical reaction networks. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 772–784, 2014.



- 19 Yuval Emek and Jara Uitto. Dynamic networks of finite state machines. In *Proceedings of International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, pages 19–34, 2016.
- 20 Yuval Emek and Roger Wattenhofer. Stone age distributed computing. In *Proceedings of ACM Symposium on Principles of Distributed Computing (PODC)*, pages 137–146, 2013.
- 21 Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.
- 22 Greg N. Frederickson and Nancy A. Lynch. Electing a leader in a synchronous ring. *J. ACM*, 34(1):98–115, 1987.
- 23 Robert G. Gallager, Pierre A. Humblet, and Philip M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. Program. Lang. Syst.*, 5(1):66–77, 1983.
- 24 M. Gardner. The fantastic combinations of John Conway’s new solitaire game ‘life’. *Scientific American*, 223(4):120–123, 1970.
- 25 Lauri Hella, Matti Järvisalo, Antti Kuusisto, Juhana Laurinharju, Tuomo Lempäinen, Kerkko Luosto, Jukka Suomela, and Jonni Virtema. Weak models of distributed computing, with connections to modal logic. *Distributed Computing*, 28(1):31–53, 2015.
- 26 Alon Itai and Michael Rodeh. Symmetry breaking in distributed networks. *Inf. Comput.*, 88(1):60–87, 1990.
- 27 Laurent Keller and Peter Nonacs. The role of queen pheromones in social insects: queen control or queen signal? *Animal Behaviour*, 45(4):787–794, 1993.
- 28 Jennie J. Kuzdzal-Fick, David C. Queller, and Joan E. Strassmann. An invitation to die: initiators of sociality in a social amoeba become selfish spores. *Biology letters*, 6(6):800–802, 2010.
- 29 Ivan Lavallée and Christian Lavault. Spanning tree construction for nameless networks. In *Proceedings of International Workshop on Distributed Algorithms (WDAG)*, pages 41–56, 1990.
- 30 Othon Michail, Ioannis Chatzigiannakis, and Paul G. Spirakis. *New Models for Population Protocols*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2011.
- 31 John Von Neumann. *Theory of Self-Reproducing Automata*. University of Illinois Press, Champaign, IL, USA, 1966.
- 32 David Peleg. *Distributed Computing: A Locality-sensitive Approach*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.
- 33 Baruch Schieber and Marc Snir. Calling names on nameless networks. *Inf. Comput.*, 113(1):80–101, 1994.
- 34 Joanna M. Setchell, Marie Charpentier, and E. Jean Wickings. Mate guarding and paternity in mandrills: factors influencing alpha male monopoly. *Animal Behaviour*, 70(5):1105–1120, 2005.
- 35 Jonathan M.W. Slack. *Essential developmental biology*. John Wiley & Sons, 2009.
- 36 NSF workshop on self-organizing particle systems (SOPS). <http://sops2014.cs.upb.de/>, 2014.
- 37 Stephen Wolfram. *A New Kind of Science*. Wolfram Media Inc., Champaign, Illinois, US, United States, 2002.



# The Role of A-priori Information in Networks of Rational Agents

**Yehuda Afek**

Tel-Aviv University, Tel-Aviv, Israel  
afek@post.tau.ac.il

**Shaked Rafaeli**

Tel-Aviv University, Tel-Aviv, Israel  
shakedr@mail.tau.ac.il

**Moshe Sulamy**

Tel-Aviv University, Tel-Aviv, Israel  
moshe.sulamy@cs.tau.ac.il

---

## Abstract

Until now, distributed algorithms for rational agents have assumed a-priori knowledge of  $n$ , the size of the network. This assumption is challenged here by proving how much a-priori knowledge is necessary for equilibrium in different distributed computing problems. Duplication – pretending to be more than one agent – is the main tool used by agents to deviate and increase their utility when not enough knowledge about  $n$  is given.

We begin by proving that when no information on  $n$  is given, equilibrium is impossible for both Coloring and Knowledge Sharing. We then provide new algorithms for both problems when  $n$  is a-priori known to all agents. However, what if agents have partial knowledge about  $n$ ? We provide tight upper and lower bounds that must be a-priori known on  $n$  for equilibrium to be possible in Leader Election, Knowledge Sharing, Coloring, Partition and Orientation.

**2012 ACM Subject Classification** Theory of computation → Distributed computing models

**Keywords and phrases** rational agents, distributed game theory, coloring, knowledge sharing

**Digital Object Identifier** 10.4230/LIPIcs.DISC.2018.5

**Related Version** A full version of the paper is available at [6], <https://arxiv.org/abs/1711.04728>.

**Funding** This research was supported by the Israel Science Foundation (grant 1386/11).

**Acknowledgements** We would like to thank Doron Mukhtar for showing us the ring partition problem and proving it is *unbounded*, when we thought such problems do not exist. We would also like to thank Michal Feldman, Amos Fiat, and Yishay Mansour for helpful discussions.

## 1 Introduction

The complexity and simplicity of most distributed computing problems depend on the inherent a-priori knowledge given to all participants. Usually, the more information processors in a network start with, the more efficient and simple the algorithm for a problem is. Sometimes, this information renders an otherwise unsolvable problem, solvable.

In game-theoretic distributed computing, algorithms run in a network of *rational agents* that may deviate from an algorithm if they deem the deviation more profitable for them. Rational agents have always been assumed to know the number of participants in the



© Yehuda Afek, Shaked Rafaeli, and Moshe Sulamy;  
licensed under Creative Commons License CC-BY

32nd International Symposium on Distributed Computing (DISC 2018).

Editors: Ulrich Schmid and Josef Widder; Article No. 5; pp. 5:1–5:18

Leibniz International Proceedings in Informatics



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

network [1, 4, 7, 24, 43], when in fact this assumption is not only very unrealistic in today's Internet, but also provides agents with non-trivial information which is critical for equilibrium.

Consider for example a large world-wide social network on which a distributed algorithm between a large portion of its members is run. It does not necessarily have the time to verify the number of participants, or the service it provides with the algorithm will be irrelevantly slow. If  $n$  is known to all participants, as was assumed in previous works about rational agents, that would not be a problem. However, what if  $n$  is not known beforehand and this allows one of the participants to skew the result in his favor?

The problems we examine here can be solved in the game-theoretic setting when  $n$  is a-priori known. However, learning the size of the network reliably is not possible with rational agents and thus we show that some a-priori knowledge of  $n$  is critical for equilibrium. That is, without any knowledge of  $n$ , equilibrium for these problems is impossible. In contrast, these problems can be solved without knowledge of  $n$  if the participants are not rational, since we can acquire the size of the network using broadcast and echo.

When  $n$  is not a-priori known, agents may deviate from the algorithm by *duplicating* themselves to affect the outcome. This deviation is also known as a Sybil Attack [20], commonly used to manipulate internet polls, increase page rankings in Google [15] and affect reputation systems such as eBay [14, 16]. In this paper, we use a Sybil Attack to prove impossibility of equilibria. For each problem presented, an equilibrium when  $n$  is known is provided here, or in previous work. Thus when  $n$  is *not* known an agent must duplicate to increase its utility, or otherwise if no agent duplicates and the network is 2-connected, a simple broadcast and echo would reveal the actual network size  $n$  and the existing equilibrium would apply. Obviously, deviations from the algorithm that include both duplicating and additional cheating, such as lying about the input of duplicated agents or fixing the result of a random coin flip, are also possible. When considering a deviation, an agent assumes it is the only deviating agent, and we assume that there are no coalitions of cheating agents.

Intuitively, the more agents an agent is disguised as, the more power to affect the output of the algorithm it has. For every problem, we strive to find the maximum number of duplications a cheater may be allowed to duplicate without gaining the ability to affect the output, i.e., equilibrium is still possible. This maximum number of duplications depends on whether other agents will detect that a duplication has taken place, since the network could not possibly be this large. To detect this situation they need to possess some knowledge about the network size, or about a specific structure.

We translate this intuition into a precise relation between the lower bound  $\alpha$  and the upper bound  $\beta \geq \alpha$  on  $n$ , that must be a-priori known in order for equilibrium to be possible. We denote this relation  $f$ -bound. These bounds hold for both deterministic and non-deterministic algorithms.

These bounds show what algorithms may be used in specific networks. For example, in an internal business network, some algorithms may work because every member in the network knows there are no more than several thousand computers in the network, while for other algorithms this knowledge is not tight enough.

Table 1 summarizes our contributions and related previous work (where there is a citation). Known  $n$  refers to algorithms in which  $n$  is a-priori known to all agents. Unknown  $n$  refers to algorithms or impossibility of equilibrium when agents a-priori know *no* bound on  $n$ . The  $f$ -bound for each problem is a function  $f$  for which there is an equilibrium when the a-priori bounds on  $n$  satisfy  $\alpha \leq \beta \leq f(\alpha)$ , and no equilibrium exists when  $\beta > f(\alpha)$ . A problem is  $\infty$ -bound if there is an equilibrium given *any* finite bound, but no equilibrium exists if no bound or information about  $n$  is a-priori given. A problem is *unbounded* if there is an equilibrium even when neither  $n$  nor any bound on  $n$  is given.

■ **Table 1** Summary of paper contributions, equilibria and impossibility results for different problems with different a-priori knowledge about  $n$ .

\*  $f$ -bound proven for a ring graph, otherwise holds for any 2-connected graph

Problem	Known $n$	Unknown $n$	$f$ -bound
Coloring	✓ Section 4	Impossible Section 3	$\infty^*$
Leader Election	✓ ADH'13 [4]	Impossible ADH'13 [4]	$(\alpha + 1)$
Knowledge Sharing	✓ AGLS'14 [7]	Impossible Section 3	$(2\alpha - 2)^*$
2-Knowledge Sharing			$\infty$
Partition, Orientation	✓ Section 5	✓ Section 5	Unbounded

## 1.1 Related Work

The connection between distributed computing and game theory stemmed from the problem of secret sharing [37]. Further works continued the research on secret sharing and multiparty computation when both Byzantine and rational agents are present [2, 18, 21, 22, 23, 31].

Another line of research presented the BAR model (Byzantine, acquiescent and rational) [8, 33, 42], while a related line of research discusses converting solutions with a mediator to cheap talk [2, 3, 12, 13, 19, 27, 32, 38, 40, 41].

Abraham, Dolev, and Halpern [4] were the first to present protocols for networks of rational agents, specifically protocols for Leader Election. In [7] the authors continue this line of research by providing basic building blocks for game theoretic distributed algorithms, namely a wake-up and knowledge sharing equilibrium building blocks. Algorithms for consensus, renaming, and leader election are presented using these building blocks. Consensus was researched further by Halpern and Vilacça [24], who showed that there is no ex-post Nash equilibrium, and a Nash equilibrium that tolerates  $f$  failures under some minimal assumptions on the failure pattern. Yifrach and Mansour [43] studied fair Leader Election protocols, giving an almost tight resilience analysis. Bank, Sulamy, and Wasserman [11] examined the case where the  $id$  space is limited, calculating the minimal threshold for equilibrium.

Coloring and Knowledge Sharing have been studied extensively in a distributed setting [9, 10, 17, 26, 28, 29, 39]. An algorithm for Knowledge Sharing with rational agents was presented in [7], while Coloring with rational agents has not been studied previously, to the best of our knowledge.

Distributed algorithms in which  $n$  is not known either implicitly or explicitly have been extensively studied in many other contexts, see for example [5, 25]. In last year's DISC in the permissionless network model and the context of consensus for blockchain [34, 35, 36] similar bounds (factor 2 in their case) on the number of cheating agents have been proved for the consensus task, in the synchronous case.

## 2 Model

We use the standard message-passing, synchronous model, where the network is a bidirectional graph  $G = (V, E)$  with  $n$  nodes, each node representing an agent with unlimited computational power, and  $|E|$  edges over which they communicate in rounds.  $G$  is assumed to be 2-vertex-

connected<sup>1</sup>. Throughout the entire paper,  $n$  always denotes the *actual* number of nodes in the network.

Initially, each agent knows its own *id* and input, but not the *id* or input of any other agent. For any information that an agent does not know, we assume its prior is uniformly distributed over all possible values. For example, considering the prior of an agent over the *ids* of all other agents, at round 0 each possible permutation of the  $n - 1$  *ids* in the network is equally possible. Similarly for all possible sets of input vectors, preference vectors, network size, etc. Furthermore, we assume all agents start the protocol together at round 0, i.e., all agents wake-up at the same time. If not, we can use the Wake-Up [7] building block to relax this assumption.

## 2.1 Equilibrium in Distributed Algorithms

Informally, a distributed algorithm is an equilibrium if no agent at no point in the execution can do better by unilaterally deviating from the algorithm. When considering a deviation, an agent assumes all other agents follow the algorithm, i.e., it assumes it is the only agent deviating. We assume there are no coalitions of cheating agents.

Formally, let  $o_a$  be the output of agent  $a$ , let  $\Theta$  be the set of all possible output vectors, and denote the output vector  $O = (o_1, \dots, o_n) \in \Theta$ , where  $O[a] = o_a$ . Let  $\Theta_L$  be the set of *legal* output vectors, in which the protocol terminates successfully, and let  $\Theta_E$  be the set of *erroneous* output vectors, such that  $\Theta = \Theta_L \cup \Theta_E$  and  $\Theta_L \cap \Theta_E = \emptyset$ .

Each agent  $a$  has a utility function  $u_a : \Theta \rightarrow \mathbb{N}$ . The higher the value assigned by  $u_a$  to an output vector, the better this vector is for  $a$ . As in previous works [4, 7, 43], the utility function is required to satisfy the *Solution Preference* which guarantees that an agent never has an incentive to fail the algorithm. Otherwise, they would simply be Byzantine faults. An agent fails the algorithm only when it detects that another agent had deviated.

► **Definition 1** (Solution Preference). The utility function  $u_a$  of an agent  $a$  never assigns a higher utility to an erroneous output than to a legal one, i.e.:

$$\forall a, O_L \in \Theta_L, O_E \in \Theta_E : u_a(O_L) \geq u_a(O_E)$$

We differentiate the *legal* output vectors, which ensure the output is valid and not erroneous, from the *correct* output vectors, which are output vectors that are a result of a correct execution of the algorithm, i.e., without any deviation. For example, in a Consensus protocol that decides by a majority and a network where the majority of agents received 1 as input and at least one agent received 0, deciding on 0 is *legal*, as it is a valid output for Consensus, but *incorrect*, as it necessarily resulted in a deviation from the protocol in use. The Solution Preference guarantees agents never prefer an erroneous output. However, they may prefer a *legal* but *incorrect* output.

The Solution Preference property introduces the threat agents face when deviating: Agents know that if another agent catches them cheating, it outputs  $\perp$  and the algorithm fails. In other words, the output is erroneous, i.e., in  $\Theta_E$ .

For simplicity, we assume agents only have preferences over their own output, i.e., for any  $O_1, O_2 \in \Theta_L$  in which  $O_1[a] = O_2[a]$ ,  $u_a(O_1) = u_a(O_2)$ . Additionally, each agent  $a$  has a

---

<sup>1</sup> This property was shown necessary in [7], since if a bottleneck node exists it can alter any message passing through it. Such a deviation cannot be detected since all messages between the sub-graphs this node connects must traverse through it. This node can then skew the algorithm according to its preferences.

*single* preferred output value  $p_a$ , and we normalize the utility function values, such that<sup>2</sup>:

$$u_a(O) = \begin{cases} 1 & o_a = p_a \text{ and } O \in \Theta_L \\ 0 & o_a \neq p_a \text{ or } O \in \Theta_E \end{cases} \quad (1)$$

Our results hold for *any* utility function that satisfies Solution Preference. For clarity and ease of presentation we assume Equation 1.

► **Definition 2** (Expected Utility). Let  $r$  be a round in a specific execution of an algorithm. Let  $a$  be an arbitrary agent. For each possible output vector  $O$ , let  $x_O(s, r)$  be the probability, estimated by agent  $a$  at round  $r$ , that  $O$  is output by the algorithm if  $a$  takes step  $s$ <sup>3</sup>, and all other agents follow the algorithm. The *Expected Utility*  $a$  estimates for step  $s$  in round  $r$  of that specific execution is:

$$\mathbb{E}_{s,r}[u_a] = \sum_{O \in \Theta} x_O(s, r) \cdot u_a(O)$$

An agent will deviate whenever the deviating step has a strictly higher expected utility than the expected utility of the next step of the algorithm, even if that deviating step also increases the risk of an erroneous output.

Let  $\Lambda$  be an algorithm. If by deviating from  $\Lambda$  and taking step  $s$ , the expected utility of  $a$  is higher, we say that agent  $a$  has an *incentive to deviate* (i.e., cheat). For example, at round  $r$  algorithm  $\Lambda$  may dictate that  $a$  flips a fair binary coin and sends the result to all of its neighbors. Any other action by  $a$  is considered a *deviation*: whether the message was not sent to all neighbors, sent later than it should have, or whether the coin toss was not fair, e.g.,  $a$  only sends 0 instead of a random value. If no agent can unilaterally increase its expected utility by deviating from  $\Lambda$ , we say that the protocol is an *equilibrium*. Equilibrium is defined over a *single* deviating agent, i.e., there are no coalitions of agents.

► **Definition 3** (Distributed Equilibrium). Let  $s(r)$  denote the next step of algorithm  $\Lambda$  in round  $r$ .  $\Lambda$  is an equilibrium if for any deviating step  $\bar{s}$ , at any round  $r$  of every possible execution of  $\Lambda$ :

$$\forall a, r, \bar{s} : \mathbb{E}_{s(r),r}[u_a] \geq \mathbb{E}_{\bar{s},r}[u_a]$$

## 2.2 Knowledge Sharing

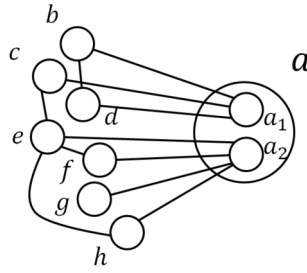
The Knowledge Sharing problem (adapted from [7]) is defined as follows:

1. Each agent  $a$  has a private input  $i_a$ , in addition to its *id*, and a function  $q$ , where  $q$  is identical at all agents.
2. A Knowledge Sharing protocol terminates *legally* if all agents output the *same* value, i.e.,  $\forall a, b : o_a = o_b \neq \perp$ . Thus the set  $\Theta_L$  is defined as:  $O \in \Theta_L \iff \forall a, b : O(a) = O(b) \neq \perp$ .
3. A Knowledge Sharing protocol terminates *correctly* (as described in Section 2.1) if each agent outputs at the end the value  $q(I)$  over the input values  $I = \{i_1, \dots, i_n\}$  of all other agents<sup>4</sup>.

<sup>2</sup> This is the weakest assumption since it still leaves a cheating agent with the highest incentive to deviate, while still satisfying Solution Preference. A utility assigning a lower value for failure than  $o_a \neq p_a$  would deter a cheating agent from deviating.

<sup>3</sup> A step specifies the entire operation of the agent in a round. This may include drawing a random number, performing any internal computation, and the contents and timing of any message delivery.

<sup>4</sup> Notice that any output is legal as long as it is the output of all agents, but only a single output value is considered *correct* for a given input vector.



■ **Figure 1** Agent  $a$  acting as separate agents  $a_1, a_2$ .

4. The function  $q$  satisfies the Full Knowledge property:

► **Definition 4** (Full Knowledge Property). A function  $q$  fulfills the *full knowledge* property if, for each agent that does not know at least one input value of another agent, any output in the range of  $q$  is *equally* possible. Formally, for any  $1 \leq j \leq m$ , fix  $(x_1, \dots, x_{j-1}, x_{j+1}, \dots, x_m)$  and denote  $z_y = |\{x_j | q(x_1, \dots, x_j, \dots, x_m) = y\}|$ . A function  $q$  fulfills the *full knowledge* property if, for any possible output  $y$  in the range of  $q$ ,  $z_y$  is the same<sup>5</sup>.

We assume that each agent  $a$  prefers a certain output value  $p_a$ .

### 2.2.1 2-Knowledge Sharing

The 2-Knowledge Sharing problem is a Knowledge Sharing problem with exactly 2 distinct possible output values.

## 2.3 Coloring

In the Coloring problem [17, 28],  $\Theta_L$  is any  $O$  such that  $\forall a : o_a \neq \perp$  and  $\forall (a, b) \in E : o_a \neq o_b$ . We assume that every agent  $a$  prefers a specific color  $p_a$ .

## 3 Impossibility With No Knowledge

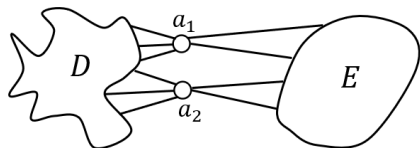
Here we prove that the common assumption that  $n$  is known is the key to the possibility of equilibrium for many problems: Without any a-priori knowledge about  $n$ , neither Knowledge Sharing nor Coloring have equilibria.

Let  $a$  be a malicious agent with  $\delta$  outgoing edges. A possible deviation for  $a$  is to simulate imaginary agents  $a_1, a_2$  and to answer over some of its edges as  $a_1$ , and over the others as  $a_2$ , as illustrated in Figure 1. From this point on  $a$  acts as if it is 2 agents. Here we assume that the  $id$  space is much larger than  $n$ , allowing us to disregard the probability that the fake  $id$  collides with an existing  $id$ , an issue dealt with in [11].

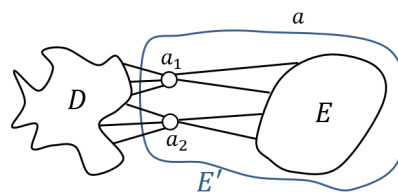
In our proofs we consider a weakened cheating agent that must decide on its duplication scheme at the very beginning of the algorithm, before any messages are exchanged. Thus, when the algorithm begins, it runs in a modified graph  $G'$  that is not the true graph  $G$  and contains duplications, but cannot be altered further by a cheater during the run of the algorithm. If this weakened cheater contradicts the possibility of equilibria, then surely a cheater that can make additional duplications while the algorithm runs would be able to

<sup>5</sup> The definition assumes input values are drawn uniformly, otherwise the definition of  $z_y$  can be expanded to the sum of probabilities over every input value for  $x_j$ .





■ **Figure 2** Graph  $H$  created by two arbitrary sub-graphs  $D, E$ .



■ **Figure 3** Example of agent  $a$  pretending to be  $E' = E \cup \{a_1, a_2\}$ .

adapt to the information it receives and increase its utility by creating more duplications<sup>6</sup>. This weakening only strengthens our impossibility proofs.

Regarding the output vector, notice that an agent that pretends to be more than one agent still outputs a *single* output at the end. The duplication causes agents to execute the algorithm as if it is executed on a graph  $G'$  (with the duplicated agents) instead of the original graph  $G$ ; however, the output is considered legal if  $O = (o_a, o_b, \dots) \in \Theta_L$  rather than if  $(o_{a_1}, o_{a_2}, o_b, \dots) \in \Theta_L$ .

It is important to emphasize that for any non-trivial distributed algorithm that is an equilibrium, the outcome cannot be calculated using only private data without communication. For rational agents, no agent can calculate the output privately at the beginning of the algorithm, since if it could calculate the output and know that its resulting utility will be 0, it would surely lie over its initial information to avoid losing, preventing equilibrium. If it knows its resulting utility is 1, it has no incentive to cheat. But there isn't always a solution in which everyone gains. This means that at round 0, for any agent  $a$  and any step  $s$  of the agent that does not necessarily result in algorithm failure, it must hold that:  $\mathbb{E}_{s,0}[u_a] \notin \{0, 1\}$  (a value of 0 means an agent will surely not get its preference, and 1 means it is guaranteed to get its preference).

In this section we label agents in the graph as  $a_1, \dots, a_n$ , set in a clockwise manner in a ring and in an arbitrary order in any other topology. These labels are not known to the agents themselves.

### 3.1 Impossibility of Knowledge Sharing

► **Theorem 5.** *There is no algorithm for Knowledge Sharing that is an equilibrium in a 2-connected graph when agents have no a-priori knowledge of  $n$ .*

**Proof.** Assume by contradiction that  $\Lambda$  is a Knowledge Sharing algorithm that is an equilibrium in any graph of agents who have absolutely no knowledge about  $n$ . Let  $D, E$  be two arbitrary 2-connected graphs of rational agents. Consider the execution of  $\Lambda$  on graph  $H$  created by  $D, E$ , and adding two nodes  $a_1, a_2$  and connecting these nodes to 1 or more arbitrary nodes in both  $D$  and  $E$  (see Figure 2).

Recall that the vector of agents' inputs is denoted by  $I = i_1, i_2, \dots, i_n$ , and  $n = |H| = |D| + |E| + 2$ . Let  $t_D$  be the first round after which  $q(I)$  can be calculated from the collective information that all agents in  $D$  have (regardless of the complexity of the computation), and

<sup>6</sup> A cheater can be forced to commit using a *Wake-Up* protocol. Since no mechanism exists to ensure authenticity, an agent will choose what information to send (false ID, false input, false neighbors). The exchanged information, as Theorem 5 shows, is already altered by a cheater and the process is *not* an equilibrium.

similarly  $t_E$  the first round after which  $q(I)$  can be calculated in  $E$ . Consider the following three cases:

1.  $t_E < t_D$ :  $q(I)$  cannot yet be calculated in  $D$  at round  $t_E$ . Let  $E' = E \cup \{a_1, a_2\}$ . Since  $E \subset E'$ , the collective information in  $E'$  at round  $t_E$  is enough to calculate  $q(I)$ . Since  $n$  is not known, an agent  $a$  could emulate the behavior of  $E'$ , making the agents believe the algorithm runs on  $H$  rather than  $D$ . In this case, this cheating agent knows at round  $t_E$  the value of  $q(I)$  in this execution, but the collective information of agents in  $D$  is not enough to calculate  $q(I)$ , which means the output of agents in  $D$  still depends on messages from  $E'$ , the cheater. Thus, if  $a$  learns that the output  $q(I) \neq p_a$ , it can simulate all possible runs of the algorithm in a state-tree, and select a course of action that has at least some probability of leading to an outcome  $q(I) = p_a$ . Such a message surely exists because otherwise,  $D$  would have also known the value of  $q(I)$ . In other words,  $a$  finds a set of messages that *might* cause the agents in  $D$  to decide a value  $x \neq q(I)$ . In the case where  $p_a = x$ , agent  $a$  increases its expected utility by sending a set of messages different than that decreed by the protocol. Thus, agent  $a$  has an incentive to deviate, contradicting distributed equilibrium.
2.  $t_D = t_E$ : both  $E$  and  $D$  have enough collective information to calculate  $q(I)$  at the same round. The collective information in  $E$  at round  $t_E$  already exists in  $E'$  at round  $t_E - 1$ . Since  $t_D = t_E$ , the collective information in  $D$  is not enough to calculate  $q(I)$  in round  $t_E - 1$ . Thus, similarly to Case 1,  $a$  can emulate  $E'$  and has an incentive to deviate.
3.  $t_E > t_D$ : Symmetric to Case 1.

Thus,  $\Lambda$  is not an equilibrium for the Knowledge Sharing problem. ◀

► **Corollary 6.** *When a cheating agent pretends to be more than  $n$  agents, there is no algorithm for Knowledge Sharing that is an equilibrium when agents have no a-priori knowledge of  $n$ .*

**Proof.** Let  $H$  be a graph such that  $|D| = |E|$ . Follow the proof of Theorem 5. ◀

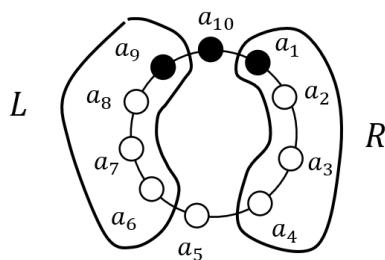
### 3.2 Impossibility of Coloring

The proof of Theorem 5 relies on the Full Knowledge property of the Knowledge Sharing problem, i.e., no agent can calculate the output before knowing all the inputs. Recall that the Coloring problem, however, is a more local problem [30], and nodes may color themselves without knowing anything about distant nodes.

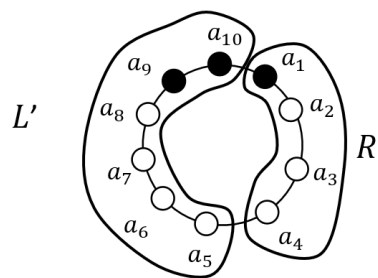
► **Theorem 7.** *There is no algorithm for Coloring that is an equilibrium in a 2-connected graph when agents have no a-priori knowledge of  $n$ .*

**Proof.** Our proof is constructed by showing a type of graph in which a cheater could deviate to increase its expected utility, regardless of the algorithm. Surprisingly, this graph is simply a *ring*. Recall that an agent outputs a *single* color, even if it pretends to be several agents. In Coloring, a cheating agent only wishes to influence the output color of its *original neighbors* to enable it to output its preferred color while maintaining the legality of the output. The key to showing an incentive to deviate is defining a way to assess the precise point in which a cheater gains an advantage. We do this by generalizing the notion of *expected utility*:

► **Definition 8 (Group Expected Utility).** Let  $r$  be a round in an execution  $\varepsilon$ , and let  $M$  be a group of agents. For any set  $S = \{s_1, \dots, s_{|M|}\}$  of steps of agents in  $M$ , let  $\Psi$  be the set of all possible executions for which the same messages traverse the links that income and outgo to/from  $M$  as in  $\varepsilon$  until round  $r$ , and in round  $r$  each agent in  $M$  takes the corresponding step in  $S$ . For each possible output vector  $O$ , let  $x_O$  be the sum of probabilities over  $\Psi$  that



■ **Figure 4** Ring with 3 agents  $a_9, a_{10}, a_1$  colliding on their preferred color.



■ **Figure 5** Ring with 3 agents colliding on their preferred color, with groups  $L', R'$ .

$O$  is decided by the protocol. For any agent  $v$ , the *Group Expected Utility* of  $u_v$  by  $M$  taking steps  $S$  at round  $r$  in execution  $\varepsilon$  is:  $\mathbb{E}_{M,S,r}[u_v] = \sum_{O \in \Theta} x_O u_v(O)$ .

Note that agents can also estimate the expected utility of *other* agents by considering a different utility function over the same output vectors of the execution  $\varepsilon$ .

Assume by contradiction that  $\Gamma$  is a Coloring algorithm that is an equilibrium in a ring with  $n$  agents  $\{a_1, \dots, a_n\}$ . Let  $G$  be a ring with a segment of  $k$  consecutive agents,  $k \geq 3$ , all of which have the same color preference  $p$ . Assume w.l.o.g., they are centered around  $a_n$  if  $k$  is odd and around  $a_n, a_1$  if even. Let  $L$  be the group of agents  $\{a_{n-1}, \dots, a_{\lfloor \frac{n}{2} \rfloor + 1}\}$ , and  $R$  the group of agents  $\{a_1, \dots, a_{\lceil \frac{n}{2} \rceil - 1}\}$ . Denote  $L' = L \cup \{a_{\lceil \frac{n}{2} \rceil}, a_n\}$  and  $R' = V \setminus L'$  (see Figures 4 and 5).

► **Definition 9.** Let  $Y$  be a group of agents (e.g.,  $L$  or  $R$ ). In any round  $r$  in an execution, let  $S^r(Y)$  denote the vector of steps of agents in  $Y$  according to the protocol. We say  $Y$  *knows* the utility of agent  $a$  if it holds that  $\mathbb{E}_{Y,S^r(Y)}[u_a] \in \{0, 1\}$ . We say  $Y$  *does not know* the utility of agent  $a$  if  $0 < \mathbb{E}_{Y,S^r(Y)}[u_a] < 1$ .

Recall that at round 0 no agent (or group of agents) knows its utility or the utility of any other agent. Consider an execution of  $\Gamma$  on ring  $G$  and the groups  $L, R$  in the following cases:

1.  $L$  does not know  $u_{a_n}$  throughout the entire execution of the algorithm, i.e., for agents in  $L$  it holds that  $0 < Pr[o_n \neq p] < 1$ . Then if  $L$  is emulated by a cheating agent, it has an incentive to deviate and set its output to  $p$  (as otherwise its utility is guaranteed to be 0).
2.  $L$  knows  $u_{a_n}$  at some round  $t_L$ , and  $R$  does not know  $u_{a_n}$  before round  $t_L$ . Consider round  $t_L - 1$  and group  $L'$ : In round  $t_L$ ,  $L$  knows the utility of  $a_n$ , thus the collective information of agents in  $L$  at round  $t_L$  already exists in  $L'$  at round  $t_L - 1$ . If  $L'$  knows that  $u_{a_n} = 1$ , then its utility is already 1; otherwise,  $L'$  knows that  $u_{a_n} = 0$ . Consider the group  $R' \subset R$ , that does not know  $u_{a_n}$  at round  $t_L - 1$ . If  $L'$  is emulated by a cheating agent  $a$ , it can send messages that increase its probability to output  $p$  from 0 to some positive probability, increasing its expected utility and thus it has an incentive to deviate.
3.  $R$  knows  $u_{a_n}$  before round  $t_L$ : symmetric to Case 2.

By the contradictory example for a ring, there is no equilibrium for Coloring 2-connected graphs when agents have no a-priori knowledge of  $n$ . ◀

## 4 Algorithms

Here we present algorithms for Knowledge Sharing (Section 4.1) and Coloring (Section 4.2). In the previous section we saw that in Knowledge Sharing, if a duplicating agent can pretend to be more than  $n$  agents equilibrium is impossible (Corollary 6). The Knowledge Sharing algorithm presented here is an equilibrium in a ring when no cheating agent pretends to be more than  $n$  agents, proving a tight bound and improving the Knowledge Sharing algorithm in [7]. The Coloring algorithm is an equilibrium in any 2-connected graph when agents a-priori know  $n$ .

Notice that using an algorithm as a subroutine is not trivial in this setting, even if the algorithm is an equilibrium, as the new context as a subroutine may allow agents to deviate towards a different objective than was originally proven. Thus, whenever a subroutine is used, its equilibrium should be proved.

### 4.1 Knowledge Sharing in a Ring

First we describe the **Secret-Transmit** ( $i_a, r, b$ ) building block in which an agent  $a$  transmits its input  $i_a$  to an agent  $b$  of its choosing, such that  $b$  learns  $i_a$  at round  $r$  and no other agent in the ring learns any information about this input. Several **Secret-Transmits** can be executed concurrently. To achieve this, agent  $a$  selects a random number  $R_a$ , and let  $X_a = R_a \oplus i_a$ . It sends  $R_a$  clockwise and  $X_a$  counter-clockwise until each reaches the agent before  $b$ . At round  $r - 1$ , these neighbors of  $b$  simultaneously send  $b$  the values  $X_a$  and  $R_a$ , thus  $b$  receives the information at round  $r$ .

We assume a global orientation around the ring. This assumption can be easily relaxed via Leader Election [7], which is an equilibrium in this application since the orientation has no effect on the output. The algorithm works as follows:

---

#### Algorithm 1 Knowledge Sharing in a Ring.

---

- 1: All agents execute Wake-Up [7] to learn the ids of all agents and  $n'$ , the size of the ring (which may include duplications)
  - 2: For each agent  $a$ , denote  $b_a^1$  the clockwise neighbor of  $a$ , and  $b_a^2$  the agent at distance  $\lfloor \frac{n'}{2} \rfloor$  counter-clockwise from  $a$
  - 3: Each agent  $a$  simultaneously performs:
    - SecretTransmit**( $i_a, n', b_a^1$ )
    - SecretTransmit**( $i_a, n', b_a^2$ )
  - 4: At round  $n' + 1$ , each agent sends its input around the ring
  - 5: At round  $2n'$  output  $q(I)$
- 

► **Theorem 10.** *In a ring, Algorithm 1 is an equilibrium when no cheating agent pretends to be more than  $n$  agents.*

**Proof.** Assume by contradiction that a cheating agent pretending to be  $d \leq n$  agents has an incentive to deviate. W.l.o.g., the duplicated agents are  $a_1, \dots, a_d$  (recall the indices  $1, \dots, n'$  are not known to the agents).

Let  $n'$  be the size of the ring including the duplicated agents, i.e.,  $n' = n + d - 1$ . The clockwise neighbor of  $a_{n'}$  is  $a_1$ , denoted  $b_{a_{n'}}^1$ . Denote  $a_c = b_{a_{n'}}^2$ , the agent at distance  $\lfloor \frac{n'}{2} \rfloor$  counter-clockwise from  $a_{n'}$ , and note that  $c \geq d$ .

When  $a_{n'}$  calls **Secret-Transmit** to  $a_1$ ,  $a_{n'}$  holds  $R_{n'}$  of that transmission until round  $n' - 1$ . When  $a_{n'}$  calls **Secret-Transmit** to  $a_c$ ,  $a_{c+1}$  holds  $X_{n'}$  of that transmission until

round  $n' - 1$ . By our assumption, the cheating agent duplicated into  $a_1, \dots, a_d$ . Since  $d < c + 1$ , the cheater receives at most one piece ( $X_{n'}$  or  $R_{n'}$ ) of each of  $a_{n'}$ 's transmissions before round  $n'$ . So, there is at least one input that the cheater does not learn before round  $n'$ . According to the Full Knowledge property (Definition 4), for the cheater at round  $n' - 1$  any output is equally possible, so its expected utility for any value it sends is the same, thus it has no incentive to cheat regarding the values it sends in round  $n' - 1$ .

Let  $a_j \in \{a_1, \dots, a_d\}$  be an arbitrary duplicated agent. In round  $n'$ ,  $i_{a_j}$  is known by its clockwise neighbor  $b_{a_j}^1$  and by  $b_{a_j}^2$ , the agent at distance  $\lfloor \frac{n'}{2} \rfloor$  counter-clockwise from  $a_j$ . Since the number of counter-clockwise consecutive agents in  $\{b_{a_j}^1, a_j, \dots, b_{a_j}^2\}$  is greater than  $\lceil \frac{n'}{2} \rceil \geq n$ , at least one of  $b_{a_j}^1, b_{a_j}^2$  is not a duplicated agent. Thus, at round  $n'$ , the input of each agent in  $\{a_1, \dots, a_d\}$  is already known by at least one agent  $\notin \{a_1, \dots, a_d\}$ .

At round  $n' - 1$  the cheater does not know the input value of at least one other agent, so by the Full Knowledge property it has no incentive to deviate. At round  $n'$  for each duplicated agent, its input is already known by a non-duplicated agent, which disables the cheater from lying about its input from round  $n'$  and on.

Thus, the cheating agent has no incentive to deviate, contradicting our assumption. ◀

In other words, in Algorithm 1 an agent has no incentive to deviate unless it duplicates more than  $n$  agents.

## 4.2 Coloring in General Graphs

Here, agents are given exact a-priori knowledge of  $n$ . Since agent *ids* are private and agents may cheat about their *id*, *ids* cannot be used to decide which of two neighbors that desire the same color actually gets it. However, an *orientation* over an edge is shared by both agents, and an acyclic orientation over the graph can be used to break ties.

Note that since the agents are rational, unless agent  $a$  knows that one or more of its neighbors output  $a$ 's preferred color  $p_a$ , it will output  $p_a$  itself, regardless of the algorithm step, which is a *deviation*. Thus, any coloring algorithm must ensure that whenever an agent can output its preferred color, it does, otherwise the agent has a trivial incentive to deviate.

We present Algorithm 2 that uses two subroutines to obtain a coloring. *Draw* (Algorithm 3) is an equilibrium in which agent  $a$  randomizes a number different from those of its neighbors and commits to it. *Prompt* (Algorithm 4) is a query that ensures  $a$  receives the correct drawn number from a neighbor. A full explanation is provided in the full paper [6].

► **Theorem 11.** *Algorithm 2 is an equilibrium for Coloring when agents a-priori know  $n$ .*

**Proof.** Let  $a$  be an arbitrary agent. Assume in contradiction that at some round  $r$  there is a possible cheating step  $s$  such that  $s \neq s_r$  and  $\mathbb{E}_{s,r}[u_a] > \mathbb{E}_{s_r,r}[u_a]$ .

Consider the possible deviations for  $a$  in every phase of Algorithm 2:

- **Wake-Up:** The order by which agents initiate Algorithm 3 has no effect on the order by which they will later set their colors. Hence,  $a$  has no incentive to publish a false *id* in the Wake-Up building block.
- ***Draw* is an equilibrium:** An agent and a witness send a random number simultaneously.
- Publishing a false  $S$  value will be caught by the verification in step 10 of Algorithm 2.
- Sending a color message not in order will be immediately recognized by the neighbors, since  $S$  values were verified.
- Agent  $a$  might output a different color than the color dictated by Algorithm 2. But if the preferred color is available, then outputting it is the only rational behavior. Otherwise, the utility for the agent is already 0 in any case. ◀

---

**Algorithm 2** Coloring via Acyclic Orientation (for agent  $a$ ).

---

```

1: Run Wake-Up                                ▷ After which all agents know graph topology
2: set  $T := \emptyset$                           ▷  $T$  is the set of values already taken by  $a$ 's neighbors ( $N(a)$ )
3: for  $i = 1, \dots, n$  do
4:   if  $id_a = i$ 'th largest  $id$  in  $V$  then      ▷ Draw random numbers in order of  $id$ 
5:      $Draw(T)$ 
6:   else
7:     wait  $|Draw|$  rounds                       ▷ Wait for  $Draw$ , takes a constant number of rounds
8:     if received  $S(v)$  from  $v \in N(a)$  then    ▷  $S(v)$  is the value of  $v$  from  $Draw$ 
9:        $T = T \cup \{S(v)\}$                    ▷ Add  $S(v)$  to set of taken values
10: for  $u \in N(a)$  simultaneously do
11:    $Prompt(u)$                                ▷ Since we must validate the value received in line 8
12: wait until all prompts are completed in the entire graph   ▷ At most  $n$  rounds
13: for round  $t = 1, \dots, n$  do:
14:   if  $S(a) = t$  then                          ▷ Wait for your turn, decreed by your  $S$  value
15:     if  $\forall v \in N(a) : o_v \neq p_a$  then  $o_a := p_a$ 
16:     else  $o_a :=$  minimum color unused by any  $v \in N(a)$ 
17:     send  $o_a$  to  $N(a)$ 

```

---



---

**Algorithm 3**  $Draw(T)$  Subroutine (for agent  $a$  and the witness  $w(a)$ ).

---

```

Denote  $X = \{1, \dots, n\} \setminus T$            ▷  $X$  is the set of numbers not drawn by neighbors
1:  $w(a) :=$  node  $b$  s.t.  $id_b$  is minimal in  $N(a)$    ▷  $N(a)$  is the set of neighbors of  $a$ 
   send witness to  $w(a)$                        ▷ choose neighbor with minimal  $id$  as witness
2:  $r(a) := random\{1, \dots, |X|\}$  drawn by  $a$ 
    $r(w(a)) := random\{1, \dots, |X|\}$  drawn by  $w(a)$ 
   send  $r(a)$  to  $w(a)$ 
   receive  $r(w(a))$  from  $w(a)$                    ▷  $a$  and witness jointly draw a random number
3: Let  $q := r(a) + r(w(a)) \bmod |X|$ .
   Set  $S(a) := q$ 'th largest number in  $X$ 
   send  $S(a)$  to all  $u \in N(a)$                  ▷ Calculate  $S(a)$  and publish to neighbors

```

---



---

**Algorithm 4**  $Prompt(b)$  Subroutine (for agent  $a$ ).

---

```

upon receiving a  $prompt(b)$  message from  $b \in N(a)$ :
1:  $p :=$  shortest simple path  $a \rightarrow w(a) \rightarrow b$    ▷  $w(a)$  is set by a preceding call to  $Draw$ 
   send  $S(a), b$  via  $p$                                        ▷ If  $v \neq w(a)$  is asked to relay  $S(a)$ ,  $v$  fails the algorithm
   send  $S(a)$  to  $b$  via  $e = (a, u)$    ▷  $b$  validates that both messages received are consistent

```

---

■ **Table 2** Knowledge Bounds; summary of results.

\* – Bound is tight only in rings.

Bound	Problem
$\alpha + 1$	Leader Election
$2\alpha - 2$	Knowledge Sharing*
$\infty$	Coloring*, 2-Knowledge Sharing
<i>unbounded</i>	Partition, Orientation

## 5 How Much Knowledge Is Necessary?

In Section 3 we have shown that with rational agents, knowledge of  $n$  is crucial; however, in some cases, *bounds* on the value of  $n$  may be enough for equilibrium. In this section we examine the effects of a-priori knowledge that *bound* the possible value of  $n$ . We show that the possibility of equilibria depends on the range  $[\alpha, \beta]$  in which  $n$  might be, and show these ranges for different problems. Table 2 summarizes our results.

Partition and Orientation have equilibria without *any* knowledge of  $n$ ; however, the former is constrained to even-sized rings, and the latter is a trivial problem in distributed computing (radius 1 in the *LOCAL* model [29]).

► **Definition 12** ( $(\alpha, \beta)$ -Knowledge). We say agents have  $(\alpha, \beta)$ -Knowledge about the actual number of agents  $n$ ,  $\alpha \leq \beta$ , if all agents a-priori know that the value of  $n$  is in  $[\alpha, \beta]$ . Agents have no information about the distribution over  $[\alpha, \beta]$ , i.e., they assume it is uniform.

► **Definition 13** ( $f$ -Bound). Let  $f : \mathbb{N} \rightarrow \mathbb{N}$ . A problem  $\mathbb{P}$  is  $f$ -bound if:

- There exists an algorithm for  $\mathbb{P}$  that is an equilibrium given  $(\alpha, \beta)$ -Knowledge for any  $\alpha, \beta$  such that  $\beta \leq f(\alpha)$ .
- For any algorithm for  $\mathbb{P}$ , there exist  $\alpha, \beta$  where  $\beta > f(\alpha)$  such that given  $(\alpha, \beta)$ -Knowledge the algorithm is not an equilibrium.

In other words, a problem is  $f$ -bound if given  $(\alpha, \beta)$ -Knowledge, there is an equilibrium when  $\beta \leq f(\alpha)$ , and there is no equilibrium if  $\beta > f(\alpha)$ .

A problem is  $\infty$ -bound if there is an equilibrium given *any* bound  $f$ , but there is no equilibrium with  $(1, \infty)$ -Knowledge. A problem is *unbounded* if there is an equilibrium with  $(1, \infty)$ -Knowledge.

Consider an agent  $a$  at the start of a protocol given  $(\alpha, \beta)$ -Knowledge. If  $a$  pretends to be a group of  $d$  agents, it can be caught when  $d + n - 1 > \beta$ , since agents might count the number of agents and catch the cheater. Moreover, *any* duplication now involves some risk since the actual value of  $n$  is not known to the cheater (similar to [11]).

An arbitrary cheating agent  $a$  simulates executions of the algorithm for every possible duplication, and evaluates its expected utility. Denote  $D$  a duplication scheme in which an agent pretends to be  $d$  agents. Let  $P_D = P[d + n - 1 \leq \beta]$  be the probability, from agent  $a$ 's perspective, that the overall size does not exceed  $\beta$ . If for agent  $a$  there exists a duplication scheme  $D$  at round 0 such that  $\mathbb{E}_{D,0}[u_a] \cdot P_D > \mathbb{E}_{s(0),0}[u_a]$ , then agent  $a$  has an incentive to deviate and duplicate itself. For each problem we look for the maximal range of  $\alpha, \beta$  where no  $d$  exists that satisfies the inequality above.

### 5.1 Knowledge Sharing

► **Theorem 14.** *Knowledge Sharing in a ring is  $(2\alpha - 2)$ -bound.*



**Proof.** Assume agents have  $(\alpha, \beta)$ -knowledge for some  $\alpha, \beta$ . A cheating agent  $a$  chooses  $d$ , the number of agents it pretends to be, that maximizes its expected utility.

Let  $k$  be the size of the range of the output function  $q$  (Definition 4). By Definition 4, any output is *equally* possible. Therefore, without deviation the expected utility of  $a$  at round 0 is:  $\mathbb{E}_{s(0),0}[u_a] = \frac{1}{k}$ .

Corollary 6 shows that when a cheating agent pretends to be *more* than  $n$  agents, it gains an advantage (thus there is no equilibrium). According to Theorem 10, Algorithm 1 is an equilibrium for Knowledge Sharing in a ring when a cheating agent pretends to be  $n$  agents or less. If  $n$  is in the range  $[\alpha, \beta]$ ,  $a$  duplicates to  $d$  agents to maximize the probability that  $d > n$  and thus the duplication increases its expected utility, while also minimizing the probability that  $d + n - 1 > \beta$  and  $a$  is caught.

To successfully gain an advantage  $a$  must duplicate to at least  $d \geq \alpha$ , or otherwise  $d$  is surely  $< n$  and by Theorem 10, there is an equilibrium. Further notice that  $d \leq \lceil \frac{\beta}{2} \rceil + 1$  (the  $+1$  is  $a$  itself) since a higher value of  $d$  increases the probability of  $a$  to be caught without increasing the probability of gaining any advantage.

From  $a$ 's perspective at the beginning of the algorithm, the value of  $n$  is uniformly distributed over  $[\alpha, \beta]$ . Let  $X > \frac{1}{k}$  be the utility  $a$  gains by pretending to be  $d > n$  agents if it is not caught, i.e., if  $d + n - 1 \leq \beta$ . The probability to duplicate to  $d > n$  agents and not be caught is  $\frac{d-\alpha}{\beta-\alpha+1}$ . On the other hand, when pretending to be  $d \leq n$  agents without being caught the utility of  $a$  does not change and is  $\frac{1}{k}$ , and this has a probability of  $\frac{\lceil \frac{\beta}{2} \rceil + 1 - d}{\beta - \alpha + 1}$ . In all other cases  $d + n - 1 > \beta$  and  $a$  is caught, resulting in a utility of 0. Thus, the expected utility of agent  $a$  at round 0 is:

$$\mathbb{E}_{D,0}[u_a] = X \cdot \frac{d - \alpha}{\beta - \alpha + 1} + \frac{1}{k} \cdot \frac{\lceil \frac{\beta}{2} \rceil + 1 - d}{\beta - \alpha + 1} \quad (2)$$

The expected utility in (2) reaches a maximum at  $d = \lceil \frac{\beta}{2} \rceil + 1$ , so set  $d$  to that number as the best cheating strategy. Recall that  $a$  deviates from the algorithm whenever  $\mathbb{E}_{D,0}[u_a] > \frac{1}{k}$ :

$$\mathbb{E}_{D,0}[u_a] = X \cdot \frac{\lceil \frac{\beta}{2} \rceil + 1 - \alpha}{\beta - \alpha + 1} + \frac{1}{k} \cdot \frac{\lceil \frac{\beta}{2} \rceil - \lceil \frac{\beta}{2} \rceil}{\beta - \alpha + 1} > \frac{1}{k} \quad (3)$$

As  $k$  grows,  $\frac{1}{k}$  approaches 0. By setting  $\frac{1}{k} = 0$  Equation 3 shows that agent  $a$  has an incentive to deviate when  $\lceil \frac{\beta}{2} \rceil + 1 - \alpha > 0$ . When  $\beta$  is even:  $\beta > 2\alpha - 2$ , otherwise:  $\beta > 2\alpha - 1$ . Thus, Algorithm 1 is an equilibrium for Knowledge Sharing when agents have  $(\alpha, \beta)$ -knowledge such that  $\beta \leq 2\alpha - 2$ , and there exist  $\alpha, \beta > 2\alpha - 2$  such that there is no equilibrium for Knowledge Sharing when agents have  $(\alpha, \beta)$ -knowledge. By Definition 13, Knowledge Sharing is  $(2\alpha - 2)$ -bound in rings. ◀

To find the  $f$ -bound for any specific value of  $k$  and in any graph, we derive  $\beta$  as a function of  $\alpha$ :

$$\begin{cases} \beta \text{ is even} & \beta(kX - 2) > 2\alpha kX - 2kX - 2\alpha + 2 \\ \beta \text{ is odd} & \beta(kX - 2) > 2\alpha kX - kX - 2\alpha \end{cases} \quad (4)$$

► **Corollary 15.** *2-Knowledge Sharing in a ring is  $\infty$ -bound.*

**Proof.** The inequalities in 4 are satisfiable only if  $X > 2 \cdot \frac{1}{k}$ . Since  $X \leq 1$ , the inequalities cannot be satisfied in 2-Knowledge Sharing ( $k = 2$ ) and  $a$  has no incentive to deviate, given *any* bound on  $n$ . ◀



**Algorithm 5** Coloring in a Ring.

- 
- 1: Wake-Up to learn the size of the ring.
  - 2: Assume arbitrary global direction over the ring (can be relaxed via Leader Election [7]).
  - 3: Run 2-Knowledge Sharing to randomize a single global bit  $b \in \{0, 1\}$ .
  - 4: Publish the preferred color of each agent simultaneously over the entire ring.
  - 5: In each group of consecutive agents that prefer the same color, if  $b = 0$  the even agents (according to the orientation) output their preferred color, else the odd agents do.
  - 6: If an agent has no neighbors who prefer the same color, it outputs its preferred color.
  - 7: Any other agent outputs the minimal available color.
- 

**5.2 Coloring**

► **Theorem 16.** *Coloring in a ring is  $\infty$ -bound.*

**Proof.** Consider Algorithm 5 which solves coloring in a ring using 2-Knowledge Sharing.

It is easy to see that Algorithm 5 is an equilibrium and results in a legal coloring of the ring. It uses 2-Knowledge Sharing and thus, following Corollary 15, it proves Theorem 16. ◀

**5.3 Leader Election**

In the Leader Election problem, each agent  $a$  outputs  $o_a \in \{0, 1\}$ , where  $o_a = 1$  means that  $a$  was elected leader and  $o_a = 0$  means otherwise.  $\Theta_L = \{O \mid \exists a : o_a = 1, \forall b \neq a : o_b = 0\}$ . An agent prefers either 0 or 1.

► **Theorem 17.** *Leader Election is  $(\alpha + 1)$ -bound.*

**Proof.** Recall that any Leader Election algorithm must be *fair* [4], i.e., every agent must have equal probability of being elected leader for the algorithm to be an equilibrium.

Given  $f(\alpha) = \alpha + 1$ , the actual number of agents  $n$  is either  $\alpha$  or  $\alpha + 1$ . If an agent follows the protocol, the probability of being elected is  $\frac{1}{n}$ . If it duplicates itself once, the probability that one of its instances is elected is  $\frac{2}{n+1}$ , but if  $n = \alpha + 1$  then  $n' > \beta$ , it is easily detected and its utility is 0. Thus  $\mathbb{E}_{D,0}[u_a] = \frac{1}{2} \frac{2}{n+1} < \frac{1}{n}$ , i.e., no agent has an incentive to deviate.

Given  $f(\alpha) = \alpha + 2$ , then  $n$  is in  $[\alpha, \alpha + 2]$ . If an agent follows the protocol, its expected utility is still  $\frac{1}{n}$ . If it duplicates itself once, the probability that a duplicate is elected is still  $\frac{2}{n+1}$ , however *only* if  $n = \alpha + 2$  then  $n' > \beta$  and the cheater is caught. Thus,  $\mathbb{E}_{D,0}[u_a] = \frac{2}{3} \frac{2}{n+1} > \frac{1}{n}$  for any  $n > 3$ . So the agent *has* an incentive to deviate. ◀

**5.4 Ring Partition**

In the Ring Partition problem, the agents of an even-sized ring are partitioned into two, equally-sized groups: group 0 and group 1. An agent prefers to belong to either group 0 or 1. In the full paper [6] we prove:

► **Theorem 18.** *Ring Partition is unbounded.*

**5.5 Orientation**

In the Orientation problem, the two ends of each edge must agree on a direction for the edge. An agent prefers certain directions for its edges. In the full paper [6] we prove:

► **Theorem 19.** *The Orientation problem is unbounded.*

## 6 Discussion

In this paper we have shown that the assumption that  $n$  is a-priori known, commonly made in previous works, has a critical role in the possibility of equilibrium. In realistic scenarios, the exact size of the network may not be known to all members, or only estimates on the exact size are known in advance. In such networks, the use of duplication gives an agent power to affect the outcome of most algorithms, and in some cases makes equilibrium impossible. In this work we did not identify any problem that requires exact knowledge of  $n$  for equilibrium. Even in Leader Election, equilibrium is possible as long as  $n$  is known to be in a margin of 2.

When bounds on  $n$  are known, the  $f$ -bounds we have proven in Section 5 show that the initial knowledge required for equilibrium depends on the balance between two factors: The amount of duplications necessary to increase an agent's expected utility, and the probability that the cheater is caught duplicating. In order for an agent to have an incentive to duplicate itself, an undetected duplication needs to be more profitable than following the algorithm while also involving low risk of being caught.

Our results suggest several open directions that may be of interest:

1. Finding an equilibrium for Knowledge Sharing in a general graph with at most  $n$  duplications. This would be the missing piece that, along with our impossibility proof in Theorem 5, would prove the  $f$ -bound of  $2\alpha - 2$  is tight for general graphs.
2. Algorithms and impossibility results for other problems, as well as tight  $f$ -bounds.
3. Finding a problem that is  $\alpha$ -bound, i.e., has an equilibrium only when  $n$  is known exactly.
4. Finding more problems that have equilibrium without any knowledge of  $n$  in any graph (unlike Partition) and a non-constant radius in the LOCAL model (unlike Orientation).
5. Exploring the effects of initial knowledge of  $n$  in an asynchronous setting.

---

## References

- 1 Ittai Abraham, Lorenzo Alvisi, and Joseph Y. Halpern. Distributed computing meets game theory: Combining insights from two fields. *SIGACT News*, 42(2):69–76, 2011. doi:10.1145/1998037.1998055.
- 2 Ittai Abraham, Danny Dolev, Rica Gonen, and Joseph Y. Halpern. Distributed computing meets game theory: robust mechanisms for rational secret sharing and multiparty computation. In *PODC*, pages 53–62, 2006. doi:10.1145/1146381.1146393.
- 3 Ittai Abraham, Danny Dolev, and Joseph Y. Halpern. Lower bounds on implementing robust and resilient mediators. In *TCC*, pages 302–319, 2008. doi:10.1007/978-3-540-78524-8\_17.
- 4 Ittai Abraham, Danny Dolev, and Joseph Y. Halpern. Distributed protocols for leader election: A game-theoretic perspective. In *DISC*, pages 61–75, 2013. doi:10.1007/978-3-642-41527-2\_5.
- 5 Norman Abramson. The aloha system: Another alternative for computer communications. In *Proceedings of the November 17-19, 1970, Fall Joint Computer Conference, AFIPS '70 (Fall)*, pages 281–285, New York, NY, USA, 1970. ACM.
- 6 Y. Afek, S. Rafaeli, and M. Sulamy. Cheating by Duplication: Equilibrium Requires Global Knowledge. *ArXiv e-prints*, 2017. arXiv:1711.04728.
- 7 Yehuda Afek, Yehonatan Ginzberg, Shir Landau Feibish, and Moshe Sulamy. Distributed computing building blocks for rational agents. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, 2014.
- 8 Amitanand S. Aiyer, Lorenzo Alvisi, Allen Clement, Michael Dahlin, Jean-Philippe Martin, and Carl Porth. Bar fault tolerance for cooperative services. In *SOSP*, pages 45–58, 2005. doi:10.1145/1095810.1095816.

- 9 Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley & Sons, 2004.
- 10 B. Awerbuch, M. Luby, A. V. Goldberg, and S. A. Plotkin. Network decomposition and locality in distributed computation. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, SFCS '89, pages 364–369, Washington, DC, USA, 1989. IEEE Computer Society. doi:10.1109/SFCS.1989.63504.
- 11 D. Bank, M. Sulamy, and E. Wasserman. Reaching Distributed Equilibrium with Limited ID Space. *ArXiv e-prints*, 2018. arXiv:1804.06197.
- 12 Imre Bárány. Fair distribution protocols or how the players replace fortune. *Math. Oper. Res.*, 17(2):327–340, 1992. doi:10.1287/moor.17.2.327.
- 13 Elchanan Ben-Porath. Cheap talk in games with incomplete information. *J. Economic Theory*, 108(1):45–71, 2003. doi:10.1016/S0022-0531(02)00011-X.
- 14 Rajat Bhattacharjee and Ashish Goel. Avoiding ballot stuffing in ebay-like reputation systems. In *Proceedings of the 2005 ACM SIGCOMM Workshop on Economics of Peer-to-peer Systems*, P2PECON '05, pages 133–137, New York, NY, USA, 2005. ACM.
- 15 Monica Bianchini, Marco Gori, and Franco Scarselli. Inside pagerank. *ACM Trans. Internet Technol.*, 5(1):92–128, 2005.
- 16 Alice Cheng and Eric Friedman. Sybilproof reputation mechanisms. In *Proceedings of the 2005 ACM SIGCOMM Workshop on Economics of Peer-to-peer Systems*, P2PECON '05, pages 128–132, New York, NY, USA, 2005. ACM.
- 17 Richard Cole and Uzi Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Inf. Control*, 70(1):32–53, 1986.
- 18 Varsha Dani, Mahnush Movahedi, Yamel Rodriguez, and Jared Saia. Scalable rational secret sharing. In *PODC*, pages 187–196, 2011. doi:10.1145/1993806.1993833.
- 19 Yevgeniy Dodis, Shai Halevi, and Tal Rabin. A cryptographic solution to a game theoretic problem. In *CRYPTO*, pages 112–130, 2000. doi:10.1007/3-540-44598-6\_7.
- 20 John R. Douceur. The sybil attack. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 251–260, London, UK, UK, 2002. Springer-Verlag.
- 21 Georg Fuchsbauer, Jonathan Katz, and David Naccache. Efficient rational secret sharing in standard communication networks. In *TCC*, pages 419–436, 2010. doi:10.1007/978-3-642-11799-2\_25.
- 22 S. Dov Gordon and Jonathan Katz. Rational secret sharing, revisited. In *SCN*, pages 229–241, 2006. doi:10.1007/11832072\_16.
- 23 Adam Groce, Jonathan Katz, Aishwarya Thiruvengadam, and Vassilis Zikas. Byzantine agreement with a rational adversary. In *ICALP (2)*, pages 561–572, 2012. doi:10.1007/978-3-642-31585-5\_50.
- 24 Joseph Y. Halpern and Xavier Vilaça. Rational consensus: Extended abstract. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, PODC '16, pages 137–146, New York, NY, USA, 2016. ACM.
- 25 L. Kleinrock and F. Tobagi. Packet switching in radio channels: Part i - carrier sense multiple-access modes and their throughput-delay characteristics. *IEEE Transactions on Communications*, 23(12):1400–1416, December 1975.
- 26 Fabian Kuhn and Rogert Wattenhofer. On the complexity of distributed graph coloring. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Principles of Distributed Computing*, PODC '06, pages 7–15, New York, NY, USA, 2006. ACM. doi:10.1145/1146381.1146387.
- 27 Matt Lepinski, Silvio Micali, Chris Peikert, and Abhi Shelat. Completely fair sfe and coalition-safe cheap talk. In *PODC*, pages 1–10, 2004. doi:10.1145/1011767.1011769.

- 28 N. Linial. Legal coloring of graphs. *Combinatorica*, 6(1):49–54, 1986. doi:10.1007/BF02579408.
- 29 Nathan Linial. Distributive graph algorithms global solutions from local data. In *Proceedings of the 28th Annual Symposium on Foundations of Computer Science, SFCS '87*, pages 331–335, Washington, DC, USA, 1987. IEEE Computer Society. doi:10.1109/SFCS.1987.20.
- 30 Nathan Linial. Locality in distributed graph algorithms. *SIAM Journal on Computing*, 21(1):193–201, 1992. doi:10.1137/0221015.
- 31 Anna Lysyanskaya and Nikos Triandopoulos. Rationality and adversarial behavior in multi-party computation. In *CRYPTO*, pages 180–197, 2006. doi:10.1007/11818175\_11.
- 32 Robert McGrew, Ryan Porter, and Yoav Shoham. Towards a general theory of non-cooperative computation. In *TARK*, pages 59–71, 2003. doi:10.1145/846241.846249.
- 33 Thomas Moscibroda, Stefan Schmid, and Roger Wattenhofer. When selfish meets evil: byzantine players in a virus inoculation game. In *PODC*, pages 35–44, 2006. doi:10.1145/1146381.1146391.
- 34 Rafael Pass and Elaine Shi. Hybrid Consensus: Efficient Consensus in the Permissionless Model. In *31st International Symposium on Distributed Computing (DISC 2017)*, 2017.
- 35 Rafael Pass and Elaine Shi. The sleepy model of consensus. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology – ASIACRYPT 2017*, pages 380–409, Cham, 2017. Springer International Publishing.
- 36 Rafael Pass and Elaine Shi. Rethinking large-scale consensus. Cryptology ePrint Archive, Report 2018/302, 2018. URL: <https://eprint.iacr.org/2018/302>.
- 37 Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979. doi:10.1145/359168.359176.
- 38 Yoav Shoham and Moshe Tennenholtz. Non-cooperative computation: Boolean functions with correctness and exclusivity. *Theoretical Computer Science*, 343(1–2):97–113, 2005.
- 39 Mária Szegedy and Sundar Vishwanathan. Locality based graph coloring. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing, STOC '93*, pages 201–207, New York, NY, USA, 1993. ACM. doi:10.1145/167088.167156.
- 40 Amparo Urbano and Jose E. Vila. Computational complexity and communication: Coordination in two-player games. *Econometrica*, 70(5):1893–1927, September 2002. URL: <http://ideas.repec.org/a/ecm/emetrp/v70y2002i5p1893-1927.html>.
- 41 Amparo Urbano and José E. Vila. Computationally restricted unmediated talk under incomplete information. *Economic theory*, 2004.
- 42 Edmund L. Wong, Isaac Levy, Lorenzo Alvisi, Allen Clement, and Michael Dahlin. Regret freedom isn't free. In *OPODIS*, pages 80–95, 2011. doi:10.1007/978-3-642-25873-2\_7.
- 43 Assaf Yifrach and Yishay Mansour. Fair leader election for rational agents in asynchronous rings and networks. In *PODC '18*, 2018. doi:10.1145/3212734.3212767.

# Distributed Approximate Maximum Matching in the CONGEST Model

Mohamad Ahmadi<sup>1</sup>

University of Freiburg, Germany  
mahmadi@cs.uni-freiburg.de

Fabian Kuhn<sup>2</sup>

University of Freiburg, Germany  
kuhn@cs.uni-freiburg.de

Rotem Oshman

Tel Aviv University, Israel  
roshman@tau.ac.il

---

## Abstract

We study distributed algorithms for the maximum matching problem in the CONGEST model, where each message must be bounded in size. We give new deterministic upper bounds, and a new lower bound on the problem.

We begin by giving a distributed algorithm that computes an exact maximum (unweighted) matching in bipartite graphs, in  $O(n \log n)$  rounds. Next, we give a distributed algorithm that approximates the fractional weighted maximum matching problem in general graphs. In a graph with maximum degree at most  $\Delta$ , the algorithm computes a  $(1 - \varepsilon)$ -approximation for the problem in time  $O(\log(\Delta W)/\varepsilon^2)$ , where  $W$  is a bound on the ratio between the largest and the smallest edge weight. Next, we show a slightly improved and generalized version of the deterministic rounding algorithm of Fischer [DISC '17]. Given a fractional weighted maximum matching solution of value  $f$  for a given graph  $G$ , we show that in time  $O((\log^2(\Delta) + \log^* n)/\varepsilon)$ , the fractional solution can be turned into an integer solution of value at least  $(1 - \varepsilon)f$  for bipartite graphs and  $(1 - \varepsilon) \cdot \frac{g-1}{g} \cdot f$  for general graphs, where  $g$  is the length of the shortest odd cycle of  $G$ . Together with the above fractional maximum matching algorithm, this implies a deterministic algorithm that computes a  $(1 - \varepsilon) \cdot \frac{g-1}{g}$ -approximation for the weighted maximum matching problem in time  $O(\log(\Delta W)/\varepsilon^2 + (\log^2(\Delta) + \log^* n)/\varepsilon)$ .

On the lower-bound front, we show that even for unweighted fractional maximum matching in bipartite graphs, computing an  $(1 - O(1/\sqrt{n}))$ -approximate solution requires at least  $\tilde{\Omega}(D + \sqrt{n})$  rounds in CONGEST. This lower bound requires the introduction of a new 2-party communication problem, for which we prove a tight lower bound.

**2012 ACM Subject Classification** Mathematics of computing  $\rightarrow$  Graph algorithms, Mathematics of computing  $\rightarrow$  Approximation algorithms

**Keywords and phrases** distributed graph algorithms, maximum matching, deterministic rounding, communication complexity

**Digital Object Identifier** 10.4230/LIPIcs.DISC.2018.6

**Related Version** A full version of the paper is available at [1], <http://tr.informatik.uni-freiburg.de/reports/report286/report00286.pdf>.

---

<sup>1</sup> Supported by ERC Grant No. 336495 (ACDC).

<sup>2</sup> Supported by ERC Grant No. 336495 (ACDC).



## 1 Introduction

In the *maximum matching* problem, we are given a graph  $G$ , and asked to find a maximum-size set of edges of  $G$  which do not share any vertices. In the *weighted* version of the problem, the graph edges are associated with weights, and our goal is to find a set of vertex-disjoint edges that maximizes the total weight. Maximum matching is a fundamental graph optimization problem, extensively studied in the classical centralized setting, as well as in other settings such as streaming algorithms (e.g., [24]) and sublinear-time approximation (e.g., [29]). The problem has also received significant attention from the distributed computing community, so far focusing on approximation algorithms (cf. Section 2).

In this paper we study maximum matching in the CONGEST model, a synchronous network communication model where messages are bounded in size. We consider both exact and approximate maximum matching, weighted and unweighted, and give new upper bounds and a lower bound. Our upper bounds are deterministic, while the lower bound holds for randomized algorithms as well. Our contributions are as follows.

### 1.1 Exact Unweighted Maximum Matching in Bipartite Graphs

In the sequential world, the fastest-known algorithm for finding a maximum matching in unweighted bipartite graphs is the Hopcroft–Karp algorithm [17]. Its running time is  $O(m \cdot \sqrt{n})$  on graphs with  $n$  nodes and  $m$  edges. Its central building block is a fast way, using breadth-first-search, to find a maximal set of node-disjoint *augmenting paths*: paths of alternating matching and non-matching edges, used to increase the size of the matching.

A naive implementation of the Hopcroft–Karp algorithm in the CONGEST model would yield an algorithm requiring  $O(n^{3/2})$  rounds. Taking inspiration and ideas from Hopcroft–Karp, we are able to instead give an algorithm that takes only  $O(n \log n)$  rounds. More specifically, we obtain the following result.

► **Theorem 1.** *The deterministic round complexity in the CONGEST model of computing an exact maximum matching in unweighted, bipartite graphs is  $O(s^* \log s^*)$ , where  $s^*$  is the size of a maximum matching.*

Note that the algorithm is not assumed to initially know the value  $s^*$ .

The core of our algorithm is a procedure that finds a single augmenting path of length  $k$  in  $O(k)$  rounds. Together with the well-known fact that if we are given a matching of size  $s^* - \ell$ , we are guaranteed to have an augmenting path of length at most  $O(s^*/\ell)$ , this procedure implies the above result. To our knowledge, this is the first non-trivial algorithm for exact bipartite maximum matching in the CONGEST model.

### 1.2 Approximate Fractional Weighted Maximum Matching

One strategy for computing an approximate maximum matching is to first solve the *fractional* version of the problem, and then *round* the solution to obtain an integral matching. A *fractional* matching is the natural linear programming (LP) relaxation of the notion of a matching, where instead of taking a *set* of edges (where each edge is “taken” or “not taken”), we instead assign each edge  $e \in E$  a value  $y_e \in [0, 1]$ . Whereas before, we required that each node participate in at most one edge of the matching, we now require that for each node  $v$ , the sum of the values of  $v$ ’s edges must be at most 1. This is a linear constraint:  $\sum_{u \in N(v)} y_{\{u,v\}} \leq 1$ .

To compute a fractional matching, we can thus bring to bear the powerful machinery of linear programming (LP). In particular, the fractional maximum matching problem is a *packing LP*. Packing LPs and their duals, *covering LPs*, are a class of LPs for which there



are particularly efficient distributed solutions (e.g., [19, 26]). In this paper, we extend an approach that was developed by Eisenbrand, Funke, Garg, and Könemann [9] to solve the fractional set cover problem. We prove the following theorem.

► **Theorem 2.** *Let  $G = (V, E, w)$  be a weighted graph. Assume that  $\Delta$  is the maximum degree of  $G$ , and let  $W$  denote the ratio between the largest and smallest edge weight. Then, for every  $\varepsilon > 0$ , there is a deterministic  $O(\log(\Delta W)/\varepsilon^2)$ -time CONGEST algorithm to compute a  $(1 - \varepsilon)$ -approximation for the maximum weighted fractional matching problem in  $G$ .*

The algorithm is based on another distributed implementation of the algorithm of [9], which appeared in [19]. The algorithm of [19] is general: it approximates general covering and packing LPs. When applied to the weighted fractional matching problem, the algorithm of [19] computes a  $(1 - \varepsilon)$ -approximation in time  $O(\log(\Delta W)/\varepsilon^4)$ , which was the best  $(1 - \varepsilon)$ -approximation for the problem in the CONGEST model prior to the present work.

As we are only interested in the matching problem, our algorithm is simpler than the algorithm of [19], and more importantly, our algorithm significantly improves the  $\varepsilon$ -dependency of computing a  $(1 - \varepsilon)$ -approximate fractional matching in the CONGEST model.

### 1.3 Deterministic Rounding of Fractional Matchings

After computing a fractional matching, we wish to *round* the edge values to  $\{0, 1\}$ , to obtain an integral matching with roughly the same weight.

Randomized rounding of LP solutions, in order to obtain approximate solutions of the corresponding integer LPs, has been used for a while, even in the distributed context (e.g., [18, 19]). However, *deterministic* distributed rounding algorithms have only been studied recently. In [11], Fischer gave an amazingly simple and elegant deterministic  $O(\log^2 \Delta)$ -time algorithm, which rounds a fractional unweighted matching into an integral matching that is smaller by only a constant factor. Repeating this rounding step  $O(\log n)$  times, Fischer obtains a maximal matching in deterministic time  $O(\log^2 \Delta \log n)$ .<sup>3</sup>

At its core, the approach of Fischer [11] solves the problem on bipartite graphs, and it decomposes the problem of rounding a fractional matching to the problem of rounding fractional matchings on paths and even cycles. Our contribution in this part of the paper is two-fold. First, while Fischer loses a constant factor when rounding the matching, we show that a simple change in the algorithm allows us to only lose a factor  $(1 - \varepsilon)$  on bipartite graphs. Second, we generalize the technique to also work for weighted (fractional) matching.

► **Theorem 3.** *Let  $G = (V, E, w)$  be a weighted graph,  $\mathbf{y}$  be a fractional matching of  $G$ , and  $\varepsilon > 0$  be a parameter. There is a deterministic  $O(\frac{\log^2(\Delta/\varepsilon) + \log^* n}{\varepsilon})$ -time CONGEST algorithm that computes an matching  $M$  of  $G$  such that the ratio between the total weight of  $M$  and the value of the given fractional weighted matching  $\mathbf{y}$  is at least  $1 - \varepsilon$  if  $G$  is bipartite, and at least  $(1 - \varepsilon) \cdot \frac{g-1}{g}$  if  $G$  is not bipartite and  $g$  is the length of the shortest odd cycle of  $G$ .*

In combination with Theorem 2, we obtain a deterministic CONGEST algorithm to compute a  $(1 - \varepsilon)$ -approximate maximum weighted matching in bipartite graphs in time  $O(\frac{\log(\Delta W)}{\varepsilon^2} + \frac{\log^2(\Delta/\varepsilon) + \log^* n}{\varepsilon})$ . For general graphs, we obtain a  $(2/3 - \varepsilon)$ -approximate maximum weighted matching in the same asymptotic time. To the best of our knowledge, this is the first CONGEST algorithm that obtains an approximation ratio better than  $1/2$  for the weighted maximum matching problem in general graphs.

<sup>3</sup> Actually, the earlier polylog-time deterministic algorithms for computing a maximal matching [14, 15] can also be interpreted as approximate rounding algorithms. However, these algorithms are not explicitly phrased in this way.

## 1.4 Lower Bound for $(1 - O(1/\sqrt{n}))$ -Approximate Fractional Matching

As we said above, in this paper we show that a  $(1 - \epsilon)$ -approximate maximum matching in bipartite graphs can be computed in time  $\tilde{O}(1/\epsilon^2)$  (ignoring the logarithmic terms in  $n$ ,  $\Delta$  and  $W$ ). Is this dependence on  $\epsilon$  optimal? We do not yet know, but we are able to show that  $\tilde{O}(1/\epsilon)$  rounds *are* necessary, for sufficiently small  $\epsilon$ :

► **Theorem 4.** *There exists a constant  $\alpha \in (0, 1)$ , such that any randomized algorithm that computes a  $(1 - \alpha/\sqrt{n})$ -multiplicative approximation to the maximum fractional matching in unweighted, bipartite graphs with diameter  $O(\log n)$  requires  $\Omega(\sqrt{n}/\log n)$  rounds.*

The lower bound is based on the framework of [27], and it is shown by reduction from two-party communication complexity. Given a fast algorithm  $A$  for approximate fractional matching, we construct a protocol for two players, Alice and Bob, to solve a communication complexity problem, by simulating the execution of  $A$  in a network that the players construct.

In contrast to [27], here we are not interested in a *verification* problem. In [27], in addition to the network graph, there is a set of *marked edges*, and the goal is to check whether the marked edges satisfy some property. Thus, we can give the algorithm a “hard subgraph to check”, even if the corresponding *search problem* is easy: e.g., [27] shows that checking if the marked edges form a spanning tree is hard ( $\tilde{\Omega}(\sqrt{n} + D)$  rounds), even though *constructing* a spanning tree is easy ( $O(D)$  rounds). Here, we do not give the algorithm a set of marked edges, and instead we allow the algorithm to compute any feasible fractional matching.

To prove the lower bound, we argue that a good approximation to the maximum matching on odd paths “looks different” from one on even paths, and this difference allows us to solve a communication complexity problem, PXA, that we introduce for this purpose. We prove, using information complexity [4], that the randomized communication complexity of PXA is linear. One unusual feature of this lower bound is that at the end of the simulation, each player only knows part of the matching constructed. Thus, we cannot guarantee that both players will “see” the difference between odd and even paths, but at least one of them will. The problem PXA reflects this: instead of asking the players to *agree* on an output, each player produces its own output, and at least one of them must “be correct”.

For lack of space, many technical details are omitted in this version of the paper. They appear in the full version of the paper [1].

## 2 Related Work

We survey here only the most directly relevant work. In particular, we mostly focus on the CONGEST model, and we discuss only some of the work for the LOCAL model, where messages do not need to be of bounded size.

The first polynomial-time algorithm for unweighted maximum matching in general graphs was given by Edmonds [7, 8]. It was preceded by the algorithm of Hopcroft and Karp [17], which is restricted to bipartite graphs. Our exact algorithm for bipartite graphs is inspired by and uses insights from the Hopcroft–Karp algorithm.

Because exact maximum matching is a “global problem”, work on distributed algorithms has mostly been focused on approximation algorithms. The first ones were for the *maximal* matching problem; in the unweighted case, a maximal matching is also a  $1/2$ -approximation to the maximum matching. Even in the 80s, simple and elegant solutions for maximal matching in  $O(\log n)$  rounds were known [2, 16, 23]. (These papers give *PRAM* algorithms, but they translate to the CONGEST model easily.) The best randomized distributed algorithm for maximal matching is due to Barenboim et al. [5], and has time complexity  $O(\log \Delta + \log^3 \log n)$ .



On the deterministic side, maximal matching was first shown to be solvable in polylogarithmic distributed time,  $O(\log^4 n)$  rounds, in [14, 15]. While they do not explicitly analyze the message size, we believe that their algorithm can be implemented in the CONGEST model. Currently, the best deterministic algorithm (in CONGEST and LOCAL) is from [11], and requires  $O(\log^2 \Delta \log n)$  rounds. As one of our algorithms heavily builds on the techniques of [11], we discuss them in more detail in Section 6. The best lower bound for maximal matching, and more generally, for obtaining constant or polylogarithmic approximations for unweighted maximum matching, is  $\Omega(\min\{\frac{\log \Delta}{\log \log \Delta}, \sqrt{\frac{\log n}{\log \log n}}\})$  [20]. The lower bound even holds for randomized algorithms in the LOCAL model.

Beyond the simple  $1/2$ -approximation provided by a maximal matching, there is series of works on the distributed complexity of obtaining a  $(1 - \varepsilon)$ -approximate maximum cardinality matching. All are based on the framework of Hopcroft and Karp [17], of repeatedly computing a (nearly) maximal vertex-disjoint set of short augmenting paths. The first such algorithm is [21], a randomized CONGEST algorithm with time complexity  $O(\log n)$  for every constant  $\varepsilon > 0$ ; however, the dependence on  $\varepsilon$  is exponential in  $1/\varepsilon$ . This was recently improved in [3], which gives a randomized algorithm with time complexity  $O(\text{poly}(1/\varepsilon) \cdot \frac{\log \Delta}{\log \log \Delta})$ . Note that the  $\Delta$ -dependency of the running time matches the lower bound of [20]. There are also deterministic distributed algorithms to obtain a  $(1 - \varepsilon)$ -approximate maximum cardinality matching in polylogarithmic time [6, 10, 12, 13], but they require the LOCAL model.

As for weighted matching, the first paper to explicitly study distributed approximation of the weighted maximum matching is [28]. They give a randomized  $O(\log^2 n)$ -time algorithm with an approximation ratio of  $1/5$ . This result for the weighted case was later improved in [22] and in [21], which give  $O(\log n)$ -round randomized CONGEST algorithms with approximation ratios  $(1/4 - \varepsilon)$  and  $(1/2 - \varepsilon)$ , respectively. In [3], Bar-Yehuda et al. improve the running time and provide a  $(1/2 - \varepsilon)$ -approximation in time  $O(\log \Delta / \log \log \Delta)$ . The only known polylog-time deterministic CONGEST algorithm for approximate weighted maximum matching in general graphs is the  $(1/2 - \varepsilon)$ -approximation algorithm by Fischer [11], which runs in  $O(\log^2 \Delta \cdot \log \frac{1}{\varepsilon})$  rounds.

### 3 Model and Definitions

**Communication model.** Our algorithms and lower bounds are designed for the CONGEST model [25]. The network is modeled as an undirected  $n$ -node graph  $G = (V, E)$ , where each node has a unique  $O(\log n)$ -bit identifier. Time is divided into synchronous rounds; in each round, each node can send an  $O(\log n)$ -bit message to each of its neighbors in  $G$ . We are interested in the *time complexity* of an algorithm, which is defined as the number of rounds that are required until all nodes terminate.

For simplicity, we assume that all nodes know the maximum degree  $\Delta$  of  $G$ . In all our algorithms, one can replace the value of  $\Delta$  by a polynomial upper bound, without changing the asymptotic results. We note that at the cost of a slightly more complicated algorithm, the knowledge of  $n$  and  $\Delta$  can also be dropped completely. If the edges of  $G$  have weights, we assume that  $w_e > 0$  is the weight of edge  $e$ . We assume that the weights are normalized such that for all  $e \in E$ , we have  $0 < w_e \leq 1$ . We further assume that the nodes know a value  $W$  such that the smallest weight is at least  $1/W$ .

**Distributed matching.** When we say that a distributed algorithm *computes a matching*, we mean that when the algorithm terminates, each node of the graph knows which of its edges is in the matching (if any). Since the graph is undirected, both endpoints of an edge must agree about whether it is in the matching or not. For fractional matching, each node knows the value of all of its edges, and again, both endpoints of the edge agree on its value.

**Notation.** Let  $G = (V, E)$  be an undirected graph. The *bipartite double cover* of  $G$  is the graph  $G_2 := G \times K_2 = (V \times \{0, 1\}, E_2)$ , where there is an edge between two nodes  $(u, i)$  and  $(v, j)$  in  $E_2$  if and only if  $\{u, v\} \in E$  and  $i \neq j$ . Hence, in  $G_2$ , every node  $u$  of  $G$  is replaced by two nodes  $(u, 0)$  and  $(u, 1)$  and every edge  $\{u, v\}$  of  $G$  is replaced by the two edges  $\{(u, 0), (v, 1)\}$  and  $\{(u, 1), (v, 0)\}$ . If  $G$  is a weighted graph with edge weights  $w_e$  for  $e \in E$ , we assume that the bipartite double cover  $G_2$  is also weighted and each edge of  $G_2$  has the same weight as the underlying edge in  $G$ . Throughout the paper,  $\log$  refers to the logarithm to base 2.

#### 4 Exact Integral Maximum Matching in Bipartite Graphs

Here we present an  $O(n \log n)$ -round deterministic algorithm to compute a maximum integral matching for a given  $n$ -node bipartite graph. The algorithm is based on finding *augmenting paths* and using them to increase the size of the matching we are constructing, as in the celebrated Hopcroft–Karp sequential algorithm for matching in bipartite graphs [17]. We give a somewhat informal description of the algorithm here; the full version appears in [1].

Let us review some basic notions. Given a matching  $M$ , we say that a node  $v \in V$  is *matched* if one of its edges is in the matching, and otherwise we say that  $v$  is *free*. A path  $u_0, u_1, \dots, u_k$  is called *alternating* (with respect to  $M$ ) if  $u_0$  is free, every odd-numbered edge  $\{u_{2i}, u_{2i+1}\}$  (where  $2i + 1 \leq k$ ) is in the matching  $M$ , and every even-numbered edge  $\{u_{2i+1}, u_{2i+2}\}$  (where  $2i + 2 \leq k$ ) is not in the matching. If an alternating path ends in a free node, then it is called an *augmenting path*, and in this case we can increase the size of the matching by removing all the even-numbered edges along the path from the matching, and instead adding all the odd-numbered edges.

Our algorithm is based on the following observation, which forms the basis for the Hopcroft–Karp algorithm:

► **Lemma 5** ([17]). *Consider an unweighted graph  $G$ , and let  $M^*$  be a maximum matching in  $G$ . Then for any positive integer  $\ell$  and any matching  $M$  in  $G$ , if  $|M| \leq (1 - 1/\ell)|M^*|$ , then there is an augmenting path of length less than  $2\ell$  in  $G$  w.r.t.  $M$ .*

From Lemma 5 we get an upper bound on the length of the shortest augmenting path remaining for a matching of given size:

► **Corollary 6.** *If the maximum matching in  $G$  has size  $s^*$ , and  $M$  is a matching of size  $|M| = i$ , then  $M$  has an augmenting path of length less than  $2\lceil s^*/(s^* - i) \rceil$ .*

**Proof.** We have:  $|M| = i = s^* - (s^* - i) = s^* (1 - 1/(s^*/(s^* - i))) \leq s^* (1 - 1/\lceil s^*/(s^* - i) \rceil)$ . Therefore, by Lemma 5, there is an augmenting path of length less than  $2\lceil s^*/(s^* - i) \rceil$ . ◀

Note that the length of the shortest remaining augmenting path depends on the size  $s^*$  of the maximum matching, which we do not know. To get an *upper* bound on the length of the shortest augmenting path, we need a *lower* bound on  $s^*$ . Thus, we first deterministically compute a 2-approximation  $\hat{s} \in [s^*/2, s^*]$ , using, e.g., [11] or [19], in  $O(\log n)$  rounds. We set  $s = 2\hat{s}$ . We are guaranteed that  $s \geq s^*$ , and hence  $s/(s - i) \geq s^*/(s^* - i)$ , so we can safely use  $s$  in place of  $s^*$  when computing an upper bound on the length of the shortest augmenting path.

The core of our algorithm is a procedure called **SetupPath**: given an upper bound  $k$  on the length of the shortest augmenting path, **SetupPath**( $k$ ) finds an augmenting path in  $O(k)$  rounds. We describe this procedure below, but before showing how we find an augmenting path, let us describe the overall structure of the algorithm.

Let  $s^*$  be the size of the maximum matching in  $G$ . Our strategy is as follows: we start with an empty matching  $M$ , and improve it by searching for augmenting paths one-by-one: for each  $i = 1, 2, \dots, n-1$ , we call `SetupPath`( $2\lceil s/(s-i) \rceil$ ), spending  $O(s/(s-i))$  rounds searching for an augmenting path of length  $O(s/(s-i))$ ; if we find one, we apply it to  $M$  to increase its size by at least 1. Note that by Corollary 6 and the fact that  $s \geq s^*$ , if  $|M| = i$ , then indeed there is an augmenting path of length less than  $2\lceil s/(s-i) \rceil$ . (If  $|M| > i$ , then we might not find an augmenting path in the current iteration, but this is fine; we move on to the next value of  $i$ .)

The time spent constructing the matching is  $\sum_{i=1}^{n-1} O(s/(s-i)) = O(s \log s) = O(s^* \log s^*)$ .

Now let us explain how we find each augmenting path.

#### 4.1 Setting Up an Augmenting Path: Procedure `SetupPath`

In the `SetupPath` procedure, we are given an upper bound  $2k+1$  on the length of the shortest remaining augmenting path, and we want to find and “set up” a collection of vertex-disjoint augmenting paths, in  $O(k)$  rounds. Setting up a path means assigning the path a unique *path ID*, informing all path nodes that they are on the path, and having them confirm that they will participate in the path. Once this is done, we augment the matching along all augmenting paths that were successfully set up. (To augment along the path, we only need each path node to know its successor and predecessor on the path.) Note that unlike Hopcroft–Karp, here we do not insist on finding a maximal set of vertex-disjoint augmenting paths; we are satisfied with merely finding and setting up *one* augmenting path, provided there is one with length  $\leq 2k+1$ .

##### Finding the path

To find the augmenting path, we perform a  $k+1$ -round BFS along alternating paths, starting from all free nodes. Initially, each free node sends out a BFS token carrying its ID along all its edges, and tries to have the network propagate this token along alternating paths: in odd rounds the token is sent along edges that are not in the matching, and in even rounds it is sent along matching edges. However, every node in the network forwards *at most one BFS token*, the first one it receives. (If multiple BFS tokens are received in the same round, the one with the smallest ID will be forwarded.) BFS tokens received in subsequent rounds (or in the same round but with a larger ID) are discarded.

During the BFS traversal, each node  $v$  stores the BFS token that it propagated, if there is one, in a local variable  $src_v$ . Also, node  $v$  stores the neighbor from which  $src_v$  was received in the local variable  $pred_v$ . (If  $src_v$  was received from multiple neighbors in the same round, then  $v$  chooses an arbitrary neighbor.)

An augmenting path is *detected in round  $r$*  if in this round, two adjacent nodes  $u, v$  both send each other BFS tokens of distinct free nodes  $src_u \neq src_v$ . This means that the alternating-path BFS started by  $src_u$  has “met” the one started by  $src_v$ , yielding an augmenting path (i.e., an alternating path that starts and ends at a free node).

We show that if  $2\ell+1$  is the length of the shortest augmenting path in the graph,  $u$  is the free node with the minimum ID among the nodes that have augmenting paths of length  $2\ell+1$ , and  $v$  is the node with the minimum ID to which  $u$  has an augmenting path of length  $2\ell+1$ , then some augmenting path between  $u$  and  $v$  is detected in round  $\ell+1$ . This is because only free nodes start a BFS traversal, and no free node can “block” the BFS started by  $u$  unless it has a shorter alternating path to the same node. But this would then imply

a shorter augmenting path than the one  $u$  has, and we assumed that  $u$  has the shortest augmenting path present. Since we know that there exists an alternating path with length at most  $2k + 1$ , we only need to develop the BFS to depth  $k + 1$  before some augmenting path is detected. Therefore this phase requires  $k + 1$  rounds.

### Setting up the path

When  $u$  and  $v$  detect an augmenting path in round  $r$ , they assign it the path identifier  $(r, \{src_u, src_v\}, \{u, v\})$ . Path identifiers are sorted in lexicographic order, and we assume some fixed ordering on unordered pairs of IDs. If a node detects more than one augmenting path at the same time, it keeps the one with the smallest path identifier. (This can happen if the augmenting path has length 1, that is, if there are two adjacent free nodes.)

Next,  $u$  and  $v$  inform all the path nodes of the detected path's ID, by sending messages backwards along the *pred* pointers of the nodes on the path. As we traverse backwards, each node stores its successor on the path in a local variable *succ*. (Note that the *succ* fields point “inwards”, towards the edge that detected the path.)

Eventually, each free node receives a (possibly empty) list of augmenting path IDs for which it is an endpoint. (Note that each inner path node can only receive one path ID; only endpoints of the path may receive more than one path ID.) At this point, each free node selects the smallest path ID (in lexicographic order), and discards the others. We know that of the augmenting paths detected, the one with the smallest path ID will be selected by both its endpoints, so at least one augmenting path survives.

We now sweep forward along each selected path, to confirm that it is properly set up: the two endpoints send each other confirmation messages carrying the path ID, by having the path nodes forward the messages along the path. If an inner path node does not receive confirmation from both endpoints, it discards the path, and similarly, if an endpoint of the path does not receive confirmation from the other endpoint, it discards the path.

## 5 Fractional Matching Approximation

We first describe a distributed approximation scheme for the weighted fractional matching problem. The algorithm is based on distributed algorithm for general covering and packing linear programs, which appeared in [19]. Further, the distributed algorithm in [19] itself is based on a sequential fractional set cover algorithm by Eisenbrand, Funke, Garg, and Könemann [9].

**Reduction to the Bipartite Case:** We first show how to reduce the problem of computing a fractional (weighted) matching for a general graph  $G$  to the fractional maximum matching problem on two-colored bipartite graphs.

► **Lemma 7.** *Let  $G = (V, E)$  be a graph with positive edge weights  $w_e \geq 0$  for all  $e \in E$  and let  $H = (V \times \{0, 1\}, E_H)$  be the bipartite double cover of  $G$ .*

- (1) *Let  $\mathbf{x}$  be a fractional matching of  $G$  and let  $\mathbf{y}$  be an edge vector of  $H$  such that for every edge  $\{(u, i), (v, 1 - i)\}$  of  $H$ ,  $y_{\{(u, i), (v, 1 - i)\}} = x_{\{u, v\}}$ . Then,  $\mathbf{y}$  is a fractional matching of  $H$  of size  $\sum_{e \in E_H} w_e y_e = 2 \cdot \sum_{e' \in E} w_{e'} x_{e'}$ .*
- (2) *Let  $\mathbf{z}$  be a fractional matching of  $H$  and let  $\mathbf{y}$  be an edge vector of  $G$  such that for every edge  $\{u, v\}$  of  $G$ ,  $y_{\{u, v\}} = (z_{\{(u, 0), (v, 1)\}} + z_{\{(u, 1), (v, 0)\}})/2$ . Then  $\mathbf{y}$  is a fractional matching of  $G$  of size  $\sum_{e \in E} w_e y_e = \frac{1}{2} \cdot \sum_{e' \in E_H} w_{e'} z_{e'}$ .*

**Proof.** Follows immediately from the definition of the bipartite double (cf. Section 3). ◀

### Distributed Algorithm for 2-Colored Bipartite Graphs

In light of Lemma 7, we can w.l.o.g. assume that we are given a weighted bipartite graph  $B = (V_0 \dot{\cup} V_1, E, w)$  for which the bipartition is given (i.e., a node knows whether it is in  $V_0$  or in  $V_1$ ). We further define  $V := V_0 \cup V_1$  to be the set of all nodes.

**Formulation as a Linear Program.** The maximum weighted fractional matching problem can be phrased as a packing linear program (LP). As it will be convenient to describe our algorithm, we use the following non-standard way to describe the maximum matching problem as an LP. Consider some fractional matching  $\mathbf{z}$  that assigns a value  $z_e \geq 0$  to each edge  $e \in E$ . Instead of directly computing the variables  $z_e$ , we make a simple change of variable and we assign a value  $y_e \geq 0$  to each edge such that  $y_e = w_e \cdot z_e$ . In terms of the variables  $y_e$ , we then obtain the following packing LP:

$$\max \sum_{e \in E} y_e \quad \text{s.t.} \quad \forall v \in V : \sum_{e \in E: v \in e} \frac{y_e}{w_e} \leq 1 \quad \text{and} \quad \forall e \in E : y_e \geq 0. \quad (1)$$

After solving (1), we obtain a weighted fractional matching  $\mathbf{z}$  of the same quality by setting  $z_e := y_e/w_e$  for each edge  $e \in E$ . The dual covering LP of (1) is defined as follows:

$$\min \sum_{v \in V} x_v \quad \text{s.t.} \quad \forall e = \{u, v\} \in E : \frac{x_u}{w_e} + \frac{x_v}{w_e} \geq 1 \quad \text{and} \quad \forall v \in V : x_v \geq 0. \quad (2)$$

Note that (2) is a variation of the fractional vertex cover LP. We will design an algorithm that solves (2) and (1) at the same time. The algorithm is based on an adaptation of the greedy set cover algorithm (the vertex cover problem is a special case of the set cover problem). It is therefore most natural to think of the algorithm primarily as an algorithm for solving (2).

**The Distributed Fractional Matching Algorithm.** Our algorithm has a real-valued parameter  $\alpha > 1$  and an integer parameter  $f \geq 1$ . The values of both parameters will be fixed later. Recall that we assume that all edge weights  $w_e$  are normalized and the nodes know a value  $W \geq 1$  such that  $1/W < w_e \leq 1$  for all edges  $e \in E$ .

The algorithm maintains a variable  $x_v \geq 0$  for each node  $v \in V$  and variables  $y_e \geq 0$  and  $r_e \in [0, 1]$  for each edge  $e \in E$ . Initially, we set  $x_v := 0$ ,  $y_e := 0$ , and  $r_e := 1$  for all nodes  $v \in V$  and all edges  $e \in E$ . Throughout the algorithm, the values of  $x_v$  and  $y_e$  only increase and the value of  $r_e$  only decreases. We further define a generalized notion of the degree of a node  $v$  as  $\gamma(v) := \sum_{e \in E: v \in e} \frac{r_e}{w_e}$  and we define  $\hat{\gamma}(v) := \max_{u \in \{v\} \cup N(v)} \gamma(u)$ .

Our algorithm consists of phases: a node  $v$  participates in a phase as long as  $\gamma(v) > 0$  and  $v$  terminates as soon as  $\gamma(v) = 0$ . Alg. 1 gives the details of a single phase.

Before analyzing the algorithm in detail, we make some simple observations. First note that whenever we increase some variable  $x_v$  by 1, in line 8, we make sure that the total increase to the edge variables  $y_e$  is also equal to 1. The increase of the variables  $y_e$  is proportional to their contribution to the generalized node degree  $\gamma(v)$ . At the end, we therefore have  $\sum_{v \in V} x_v = \sum_{e \in E} y_e$ . Further, consider some node  $v \in V$  and some incident edge  $e$ . Each time, we increase  $x_v$  by 1, we divide  $r_e$  by a factor  $\alpha^{1/w_e}$ . We set  $r_e = 0$  as soon as  $r_e$  becomes less than  $\alpha^{-f}$  at the end of the algorithm, for every edge  $e = \{u, v\}$ , we therefore have  $x_u + x_v \geq w_e \cdot f$  and thus  $\frac{x_u}{w_e} + \frac{x_v}{w_e} \geq f$ . Hence, all inequalities of the LP (2) are “over-satisfied” by a factor at least  $f$  and we can therefore obtain a feasible solution  $\mathbf{x}'$  for LP (2) by setting  $x'_v := x_v/f$ . The solution  $\mathbf{y}$  for the fractional matching LP (1) is feasible. In order to obtain a feasible solution  $\mathbf{y}'$ , we compute the value  $Y_v := \sum_{e \in E: v \in e} \frac{y_e}{w_e}$  for each node  $v$  and for each edge  $e = \{u, v\}$ , we set  $y'_e := y_e / \max\{Y_u, Y_v\}$ . By LP duality,

---

**Algorithm 1:** A single phase of the fractional matching algorithm.

---

```

1 for  $i \in \{0, 1\}$  do
2   for all  $v \in V_i$  in parallel do
3     if  $\gamma(v) > 0$  then
4        $\theta_v := \hat{\gamma}(v)/\alpha$ ;
5       while  $\gamma(v) \geq \theta_v$  do
6          $x_v := x_v + 1$ ;
7         for all  $e \in E : v \in e$  do
8            $y_e := y_e + \frac{r_e/w_e}{\gamma(v)}$ ;  $r_e := r_e/\alpha^{1/w_e}$ ;
9           if  $r_e \leq \alpha^{-f}$  then  $r_e := 0$ 

```

---

the optimal solutions of (1) and (2) have the same values and we can therefore lower bound the approximation ratio of our fractional matching algorithm by the ratio  $f / \max_{v \in V} Y_v \leq 1$ . The following lemma and corollary show that for suitable choices of the parameters  $\alpha$  and  $f$ , this ratio can be made arbitrarily close to 1. The proof is similar to the analyses in [9, 19] and it appears in the full version of this paper [1].

► **Lemma 8.** *At the end of running the above fractional weighted matching algorithm, for all nodes  $v \in V$ , we have*

$$Y_v \leq \frac{\alpha^2}{\alpha - 1} \cdot (\ln(W\Delta) + (f + 1) \ln \alpha).$$

► **Corollary 9.** *Let  $\varepsilon \in (0, 1/2]$  be a parameter. By choosing  $\alpha = 1 + \varepsilon/c$  and choosing  $f = 2c \cdot \ln(\Delta W)/\varepsilon^2$  for a sufficiently large constant  $c$ , the above fractional matching algorithm can be used to compute a  $(1 - \varepsilon)$ -approximate fractional weighted matching in an arbitrary weighted graph  $G = (V, E)$ .*

It remains to bound the time complexity of the algorithm in the distributed setting. The proof appears in the full version of this paper [1]. It shows that a single phase of the algorithm can be implemented in  $O(1)$  CONGEST model rounds and that the total number of phases is  $O(\log(W\Delta)/\varepsilon^2)$ .

► **Lemma 10.** *The described fractional weighted matching algorithm can be implemented in  $O(f + \log_\alpha(W\Delta))$  rounds in the CONGEST model.*

Together with Corollary 9, Lemma 10 directly proves Theorem 2.

## 6 Deterministic Rounding of Fractional Matchings

For rounding the obtained fractional matching from Section 5, we adapt the technique by Manuela Fischer in [11]. In [11], Fischer shows how to round a fractional matching to an integral matching at the cost of losing a non-trivial constant factor (in the unweighted and in the weighted case). We show that a simple adaptation of the algorithm allows to keep the loss within a  $(1 + \varepsilon)$ -factor in the unweighted bipartite case. We further show that the method can also be generalized to the weighted bipartite case while only losing a  $(1 + \varepsilon)$ -factor in the rounding.



**Normalizing the Fractional Matching.** As for the fractional maximum matching problem in Section 5, we first solve the problem in 2-colored bipartite graphs, and we then show how to extend the solution to general graphs. The following lemma further shows that we can assume that we start with a *normalized* fractional matching where all the fractional edge values are of the form  $2^{-i}$  for some integer  $i \geq 0$ . The relatively straightforward proof of the following lemma is omitted in this short version of the paper.

► **Lemma 11.** *At the cost of at most an  $\varepsilon$ -fraction of an optimal matching, the problem of rounding a weighted fractional matching  $\mathbf{y}$  of a graph  $G$  with maximum degree  $\Delta$  can be reduced to the problem of rounding a weighted fractional matching  $\mathbf{y}'$  on a multigraph  $G'$  such that for all edges  $e$  of  $G'$ , we have  $\mathbf{y}'_e = 2^{-i}$  for some non-negative integer  $i = O(\log(\Delta/\varepsilon))$ .*

**Basic Rounding Strategy.** We use the same basic rounding approach as Fischer [11]. In the following, we assume that we are given a bipartite (multi-)graph  $B = (V_0 \dot{\cup} V_1, E)$  and a normalized fractional matching  $\mathbf{y}$  that assigns a value  $y_e = 2^{-i}$  for some integer  $i \geq 0$  to each edge  $e \in E$ . For convenience, let  $E_i$  be the set of edges  $e \in E$  for which  $y_e = 2^{-i}$  and let  $B_i := (V_0 \dot{\cup} V_1, E_i)$  be the subgraph of  $B$  induced by the edges in  $E_i$ . Assume further that  $k$  is the largest integer such that  $E_k \neq \emptyset$ , i.e., for which there is some edges  $e \in E$  with  $y_e = 2^{-k}$ . For a given parameter  $\delta > 0$ , we describe a rounding algorithm that rounds each edge  $e \in E_k$  either to value 0 or to value  $2^{-(k-1)}$  such that the total value of the fractional (weighted) matching does not decrease by more than a factor  $1 - \delta$ .

In order to do the rounding of the edges in  $E_k$ , we define a virtual graph  $B'_k$  as follows. For each node  $v \in V$ , let  $d_k(v)$  be the number of edges in  $E_k$  that are incident to  $v$ . If  $d_k(v) \geq 1$ , we create  $s_v := \lceil d_k(v)/2 \rceil$  virtual nodes  $v_1, \dots, v_{s_v}$  and we arbitrarily divide the  $d_k(v)$  edges in  $E_k$  that are incident to  $v$  among the nodes  $v_1, \dots, v_{s_v}$  such that each node  $v_i$  receives at most two such edges (i.e., if  $d_k(v)$  is even, all virtual nodes  $v_i$  get two edges and if  $d_k(v)$  is odd, one of the virtual nodes gets one edge and the others get two edges). Note that the graph  $B'_k$  has maximum degree 2 and because  $B'_k$  is bipartite, it means that it consists of disjoint paths and even cycles. The next lemma shows that we can use an arbitrary matching of  $B'_k$  to select the set of edges in  $E_k$ , which are rounded up to value  $2^{-(k-1)}$ . The proof of the lemma appears in the full version [1].

► **Lemma 12.** *Let  $M'_k$  be a matching of the graph  $B'_k$  and let  $M_k$  the corresponding subset of edges of  $E_k$ . Further, let  $\mathbf{y}'$  be obtained from the fractional matching  $\mathbf{y}$  of  $B$  by setting  $y'_e = y_e$  for all  $e \notin E_k$ ,  $y'_e = 2y_e$  for all  $e \in M_k$  and  $y'_e = 0$  for all  $e \in E_k \setminus M_k$ . Then  $\mathbf{y}'$  is a valid fractional matching of  $B$ .*

*Further, if the total weight of  $M'_k$  is at least  $(1 - \delta)/2$  of the total weight of  $B'_k$  for some  $\delta \geq 0$ , the total weight of  $\mathbf{y}'$  is at most a  $(1 - \delta)$ -factor smaller than the total weight of  $\mathbf{y}$ .*

The above rounding of edges is the main difference between the approach of [11] and our algorithm. In [11], to be on the safe side, the fractional matching value of the edge incident to a virtual node of degree 1 is always rounded down unless the total fractional matching value at the respective node is far from 1. This simple change makes the rounding more efficient and it also makes it easier to argue that the rounded matching is not much smaller than the original matching, in particular in the case of weighted matchings. Lemma 12 implies that rounding fractional matchings to integral matchings essentially boils down to computing almost maximum (weighted) matchings in graphs of maximum degree 2.

**Approximating Maximum Matching in Paths and Cycles.** As discussed above, Lemma 12 essentially reduces the problem of rounding (weighted) fractional matchings to solving the weighted maximum matching problem in paths and cycles.

► **Lemma 13.** *Let  $G = (V, E)$  be a weighted  $n$ -node graph with maximum degree 2 and assume that  $\mathcal{W}$  is the total weight of all edges of  $G$ . Let  $\delta > 0$  be a parameter, and let  $g$  be the length of the shortest odd cycle of  $G$ .<sup>4</sup> In the CONGEST model, a matching of weight at least  $\frac{g-1}{g} \cdot (1 - \delta) \cdot \mathcal{W}/2$  can be computed in time  $O(\frac{1}{\delta} \cdot \log^* n)$ .*

*If  $G = (V_0 \dot{\cup} V_1, E)$  is a bipartite graph for which the bipartition  $(V_0, V_1)$  is given, there is an  $O(1/\delta)$ -time algorithm that computes a matching of total weight at least  $(1 - \delta)\mathcal{W}/2$ .*

**Proof Sketch.** The full proof of the lemma appears in the full version [1]. The main idea of the proof is as follows. For short paths and cycles, it is straightforward to compute the required matchings. For long paths, we define an edge to be  $\ell$ -light if its weight is at most the average weight in some subpath of length  $\ell$ . We then choose a set  $L$  of  $\ell$ -light edges such that after removing those edges, the matchings of the remaining paths can be computed efficiently (either because the paths are sufficiently short or in the case of 2-colored bipartite graphs because we have a 2-edge coloring of the paths that allows to find the matching). If the  $\ell$ -light edges in  $L$  are sufficiently separated, one can show that we only lose a  $(1 - O(1/\ell))$ -factor in the matching size. ◀

**Putting the Pieces Together.** We now have all the tools that are needed for the rounding and we can therefore prove Theorem 3.

**Proof of Theorem 3.** First of all, we assume that  $\mathbf{y}$  is at least a  $1/3$ -approximation. If not, one can directly apply the weighted  $(2 + \varepsilon)$ -approximation algorithm of [11] to obtain the claim of the theorem. Because  $\mathbf{y}$  is at least a  $1/3$ -approximation and because the optimal fractional matching size is at least  $\sum_{e \in E} w_e / \Delta$ , we directly round down matching values that are smaller than  $\varepsilon / (12\Delta)$ , i.e., if  $y_e \leq \varepsilon / (12\Delta)$ , we set  $y_e := 0$ . This reduces the value of the weighted fractional matching  $\mathbf{y}$  at most by a factor  $(1 - \varepsilon/4)$ .

Using Lemma 7, we now first move to the bipartite double cover of  $G$  and by using Lemma 11, we create a multi-graph in which all matching values are negative powers of 2. Assume that the smallest matching value is  $2^{-k}$ . Because all matching values of  $\mathbf{y}$  are at least  $\varepsilon / (12\Delta)$ , we have  $k = O(\log(\Delta/\varepsilon))$ . We apply  $k$  iterations of the basic rounding, each time, we round the edges of the currently smallest values. In order to lose at most another  $(1 - \varepsilon/4)$ -factor throughout the  $k$  phases of rounding, we make sure that in each of the  $k$  iterations, we only lose a  $(1 - O(\varepsilon/k))$ -factor. In Lemma 12, we therefore have to set  $\delta = O(\varepsilon/k) = O(\varepsilon/\log(\Delta/\varepsilon))$ . Because  $B'_k$  is a 2-colored bipartite graph, Lemma 13 implies that the matching of  $B'_k$  which is necessary by Lemma 12 can be computed in time  $O(1/\delta) = O(\log(\Delta/\varepsilon)/\varepsilon)$ . After the  $k$  steps of rounding, we therefore obtain a matching of the bipartite double cover  $H$  of  $G$  of size at least  $(1 - \varepsilon/2)$  times the value of the given fractional matching of  $H$ . When using Lemma 7 to transform this matching back to  $G$ , we only obtain a fractional matching of  $G$ . However, this fractional matching is half-integral and rounding it to an integer matching can therefore be achieved by another application of Lemma 13. However, this time, we do not have a 2-coloring of the graph and  $G$  might also not be bipartite. The time for this last rounding step is therefore  $O(\log^* n/\varepsilon)$  and we lose a factor  $(1 - O(\varepsilon)) \cdot \frac{g-1}{g}$ . ◀

<sup>4</sup> Note that if  $G$  is bipartite, we have  $g = \infty$ .



## 7 Lower Bound

In this section we show that computing a  $(1 - O(1/\sqrt{n}))$ -approximation to the maximum fractional matching requires  $\Omega(\sqrt{n})$  rounds, even in bipartite graphs of diameter  $O(\log n)$ , and even for randomized algorithms which may err with constant probability.

Our graph construction is based on [27]: it is a collection of  $\Theta(\sqrt{n})$  long paths, of length  $\Theta(\sqrt{n})$ , connected to each other by a tree, which reduces the diameter to  $O(\log n)$ . The lower bound is by reduction from a 2-party communication complexity problem that we introduce for this purpose. Our lower bound is, very informally speaking, “all about” distinguishing even paths from odd paths; the communication complexity problem reflects this, and it asks the players to distinguish “odd inputs” from “even inputs”.

One might wonder why we do not simply reduce from Set Disjointness, as is usually done (e.g., in [27]). The reason is that Set Disjointness is a *decision problem*: given sets  $X, Y \subseteq [n]$ , the players must decide whether  $X \cap Y = \emptyset$ . This suffices for [27], because the typical setup there is that the input graph has some *marked edges* in it, and the goal is to decide whether the subgraph induced by the marked edges satisfies some property. In contrast, here we are interested in a *search problem*: unlike [27], we do not have a set of marked edges as part of the input; there is only the network graph, on which the algorithm must approximate the maximum fractional matching.

Another difficulty that we must overcome is that our reduction does not allow both players to compute the same output. Instead, the players may output different bits, and we view their answer is the Boolean AND of their output bits.

### The 2-Player Communication Problem

Let XOR-to-And, or XA for short, be the following problem: the players receive input bits  $x, y \in \{0, 1\}$ , respectively, and their goal is to output bits  $a, b \in \{0, 1\}$ , respectively, such that  $a \wedge b = x \oplus y$ . That is, if  $x \oplus y = 1$ , then both players should output 1, but if  $x \oplus y = 0$ , then at least one player should output 0.

For  $n \geq 1$ , let  $PXA_{n,\delta}$  (“promise XOR-to-And”) be the following problem: the players are given  $n$  copies of XA,  $x_1, y_1, \dots, x_n, y_n$ , with the *promise* that for at least  $n/3$  copies  $i$  we have  $x_i \oplus y_i = 1$ , and for at least  $(2/5)n$  copies  $i$  we have  $x_i \oplus y_i = 0$ . The goal is to solve at least  $\delta n$  of the copies correctly; that is, the players should produce outputs  $a_1, b_1, \dots, a_n, b_n$  such that for at least  $\delta n$  coordinates  $i$  we have  $a_i \wedge b_i = x_i \oplus y_i$ . The players are not charged for writing their outputs, only for the communication between them.

► **Theorem 14.** *The randomized communication complexity of  $PXA_{n,\delta}$  is  $\Omega((1 - \delta)n)$ .*

We omit the proof of the communication lower bound here, as it uses standard techniques in information complexity [4]; see the full version [1] for this proof.

### The Reduction

Given a CONGEST algorithm  $\mathcal{A}$  that computes a  $(1 - \Theta(1/\sqrt{n}))$ -multiplicative approximation to the maximum fractional matching, we construct a 2-party protocol for  $PXA_{\Theta(\sqrt{n},\delta)}$  (for a small constant  $\delta$ ).

Fix a parameter  $k = \Theta(\sqrt{n})$ , and assume for simplicity that  $n/k$  is an integer. Assume that the algorithm  $\mathcal{A}$  runs in time at most  $n/k - 1 = O(\sqrt{n})$ .

On inputs  $x, y \in \{0, 1\}^k$ , Alice and Bob construct a graph  $G_{x,y} = (V, E_{x,y})$ , consisting of

- $k$  paths, each of length  $2n/k$ , denoted  $\pi_0, \dots, \pi_{k-1}$ , where  $\pi_i = (i, 0), (i, 1), \dots, (i, 2n/k)$  for each  $i \in [k]$ .

- A complete binary tree, with  $n/k + 1$  leaves denoted  $\ell_0, \dots, \ell_{n/k}$ . Each leaf  $\ell_i$  is connected to each path node  $(j, 2i)$  for  $j = 0, \dots, n/k$ . The edges  $\{\{\ell_i, (j, 2i)\}\}_{j \in [k], i \in [n/k+1]}$  are called *bridges*.
- An additional  $n/k + 1$  nodes denoted  $x_0, \dots, x_{n/k}$ , with an edge  $\{\ell_i, x_i\}$  connecting  $x_i$  to the tree leaf  $\ell_i$  for each  $i \in [n/k + 1]$ . Nodes  $x_0, \dots, x_{n/k}$  are called *spines*.
- For each  $i \in [k]$ , if  $x_i = 1$ , Alice prepends an edge  $e_i^A = \{(i, A), (i, 0)\}$  at the beginning of the path  $\pi_i$ .
- For each  $i \in [k]$ , if  $y_i = 1$ , Bob appends an edge  $e_i^B = \{(i, B), (i, 2n/k)\}$  at the end of the path  $\pi_i$ .

Let  $\pi_i^{x,y}$  be the extended path  $\pi_i$ , after Alice or Bob either add or do not add their edge to their respective endpoints of  $\pi_i$ . The length of each extended path  $\pi_i^{x,y}$  is  $2n/k + 1$  if  $x_i \oplus y_i = 1$ , and either  $2n/k$  or  $2n/k + 2$  if  $x_i \oplus y_i = 0$ . We argue that a good approximate matching algorithm must *distinguish* between these two cases on a noticeable fraction of the paths, allowing the players to solve PXA.

The players simulate the execution of  $\mathcal{A}$  for  $n/k - 1$  rounds, until it terminates. The simulation is very similar to the one in [27], with each player initially simulating almost all nodes of the graph, but simulating fewer and fewer nodes as the execution of the algorithm progresses. Specifically, at each time  $t \leq n/k - 1$ , let  $V_A^t = [k] \times \{A, 0, 1, \dots, 2n/k - t\}$  and  $V_B^t = [k] \times \{B, t, t + 1, \dots, 2n/k\}$  be the path nodes simulated by Alice and Bob, respectively, at time  $t$ . At each time  $t$ , each player can compute the messages that the nodes in  $V_A^{t+1}$  (resp.  $V_B^{t+1}$ ) will receive in the current round from their neighbors on the path, because it knows the neighbors' local states at time  $t$ . In addition to the path nodes, the players also simulate the tree nodes and the spine nodes, fewer and fewer in each round. This part of the simulation is again very similar to [27], and we omit it here.

When  $\mathcal{A}$  terminates, *both* players know the states of nodes  $(i, n/k - 1), (i, n/k), (i, n/k + 1)$  for each  $i \in [k]$ . This overlap is important for our reduction.

Let  $E_A$  be the set of edges such that at the end of the simulation, Alice has the local states of both endpoints of the edge, and therefore knows the value the algorithm assigned to this edge. Similarly define  $E_B$  for Bob. Let  $M$  be the fractional matching computed by the algorithm.

## Producing Outputs

After the simulation ends, the players examine the fractional matching produced by the algorithm, and use it to produce outputs, as follows.

For a path  $\pi = u_0, \dots, u_{k-1}$ , let  $\pi^{-1} = u_{k-1}, \dots, u_0$  be the inverse path, and let  $\text{odd-edges}(\pi) = \{\{u_{2i}, u_{2i+1}\} \mid 2i + 1 < k\}$  be the set of odd-numbered edges along the path (the first edge, the third edge, and so on). Note that if  $\pi$  has odd length (an odd number of edges), then  $\text{odd-edges}(\pi) = \text{odd-edges}(\pi^{-1})$ , but if  $\pi$  has even length, then  $\text{odd-edges}(\pi)$  and  $\text{odd-edges}(\pi^{-1})$  are a *partition* of the edges of  $\pi$ .

For each  $i \in [k]$ , Alice outputs  $a_i = 1$  iff from her perspective, every odd-numbered edge “that she can still see” has value greater than  $1/2$ . That is,  $a_i = 1$  iff for every  $e \in \text{odd-edges}(\pi_i^{x,y}) \cap E_A$  we have  $M(e) > 1/2$ . Bob does the same, but he views the path in reverse, because he is at the other end of it: he outputs  $b_i = 1$  iff for every  $e \in \text{odd-edges}((\pi_i^{x,y})^{-1}) \cap E_B$  we have  $M(e) > 1/2$ . We argue that this odd-looking decision rule indeed captures the fact that  $M$  “looks different” on odd-length and even-length paths.

### Correctness of the Deduction

First, note that we can effectively ignore the bridge edges, and assume that they have weight zero: If  $M$  assigns non-zero weight to some bridge edge  $\{\ell_i, (j, 2i)\}$ , we can “shift” this weight onto the corresponding spine edge  $\{\ell_i, x_i\}$  and zero out the weight of the bridge edge. The resulting matching  $M'$  agrees with  $M$  on all the path edges, but since it now induces disconnected components consisting of the  $k$  paths and the tree (the bridges have value 0), it must “solve each path individually”. We can also ignore the tree and spines, because they do not contribute too much to the total value. For simplicity, let us assume here that  $M$  is a  $(1 - \Theta(1/\sqrt{n}))$ -multiplicative approximation to the maximum matching on the paths  $\pi_1^{x,y}, \dots, \pi_k^{x,y}$ .

Next, observe that if  $M$  is a  $(1 - \Theta(1/\sqrt{n}))$ -multiplicative approximation to the maximum fractional matching, then for a constant fraction of our  $k = \Theta(\sqrt{n})$  paths,  $M$  must be an  $O(1)$ -additive approximation to the maximum fractional matching on the path: the optimal fractional matching has total value at most  $2n$  on all the paths (this is a coarse upper bound), so missing even a constant value  $\alpha \in (0, 1)$  on  $\beta\sqrt{n}$  paths leads to a multiplicative approximation of only  $1 - \alpha\beta/(2\sqrt{n})$ .

We set the constants so that for a  $\delta$ -fraction of the  $k$  paths,  $M$  is at least a  $1/3$ -additive approximation. Then we show that for any path  $\pi_i^{x,y}$ , if  $M$  achieves a  $1/3$ -additive approximation to the maximum matching on  $\pi_i^{x,y}$ , then the players correctly solve coordinate  $i$ , that is,  $a_i \wedge b_i = x_i \oplus y_i$ .

The heart of the lower bound is the following simple observation:

► **Lemma 15.** *Let  $\pi$  be a path with  $2r + 1$  edges,  $r \geq 0$ , and let  $M$  be a  $1/3$ -additive approximation to the maximum fractional matching on  $\pi$ . Then for all  $e \in \text{odd-edges}(\pi)$  we have  $M(e) > 1/2$ .*

**Proof.** Suppose not, and let  $e \in \text{odd-edges}(\pi)$  be an edge with  $M(e) \leq 1/2$ . Removing edge  $e$  from  $\pi$  splits the path into two even-length paths (the suffix and the prefix), with combined length  $2r$ . On an even path of  $2i$  edges, the maximum fractional matching has total value  $i$ , and therefore the total value of  $M$  on the two even-length paths is at most  $r$ . Because  $M(e) \leq 1/2$ , the total value of  $M$  on  $\pi$  is at most  $r + 1/2$ . But the maximum fractional matching on  $\pi$  has total value  $r + 1$ , so  $M$  is not a  $1/3$ -additive approximation. ◀

► **Corollary 16.** *If  $M$  is a  $1/3$ -additive approximation on  $\pi_i^{x,y}$ , and  $x_i \oplus y_i = 1$  (so  $\pi_i^{x,y}$  has odd length), then for every odd-numbered edge  $e \in \text{odd-edges}(\pi_i^{x,y})$  we have  $M(e) > 1/2$ .*

This shows that for odd-length paths that are well-approximated by  $M$ , the players do indeed solve XA correctly. What about even-length paths? Here the situation is even simpler:

► **Lemma 17.** *If  $x_i \oplus y_i = 0$ , then there exists an edge*

$$e \in (\text{odd-edges}(\pi_i^{x,y}) \cap E_A) \cup (\text{odd-edges}((\pi_i^{x,y})^{-1}) \cap E_B)$$

with  $M(e) \leq 1/2$ .

**Proof.** Recall that when  $x_i \oplus y_i = 0$ , the length of  $\pi_i^{x,y}$  is even, and therefore  $\text{odd-edges}(\pi_i^{x,y})$  and  $\text{odd-edges}((\pi_i^{x,y})^{-1})$  form a partition of the path edges. Recall also that the overlap of the edges simulated by the players at the end,  $E_A \cap E_B$ , contains at least two adjacent edges on each path. Because  $M$  is feasible, it assigns value  $\leq 1/2$  to at least one of these edges, and since the edge is either in  $\text{odd-edges}(\pi_i^{x,y})$  or in  $\text{odd-edges}((\pi_i^{x,y})^{-1})$ , at least one of the players will output 0. ◀

## References

- 1 M. Ahmadi, F. Kuhn, and R. Oshman. Distributed approximate maximum matching in the congest model. Technical Report 286, U. Freiburg, Dept. of Computer Science, 2018. URL: <http://tr.informatik.uni-freiburg.de/reports/report286/report00286.pdf>.
- 2 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.
- 3 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.
- 4 Ziv Bar-Yossef, T. S. Jayram, Ravi Kumar, and D. Sivakumar. An information statistics approach to data stream and communication complexity. *J. Comput. Syst. Sci.*, 68(4):702–732, 2004.
- 5 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.
- 6 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.
- 7 J. Edmonds. Maximum matching and a polyhedron with 0,1 vertices. *Canadian Journal of mathematics*, pages 449–467, 1965.
- 8 J. Edmonds. Paths, trees, and flowers. *J. of Res. the Nat. Bureau of Standards*, 69 B:125–130, 1965.
- 9 F. Eisenbrand, S. Funke, N. Garg, and J. Könemann. A combinatorial algorithm for computing a maximum independent set in a t-perfect graph. In *Proceedings of 14th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 517–522, 2003.
- 10 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.
- 11 M. Fischer. Improved deterministic distributed matching via rounding. In *Proceedings of 31st Symposium on Distributed Computing (DISC)*, pages 17:1–17:15, 2017.
- 12 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.
- 13 M. Ghaffari, D. G. Harris, and F. Kuhn. On derandomizing local distributed algorithms, 2017. [arXiv:1711.02194](https://arxiv.org/abs/1711.02194).
- 14 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.
- 15 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.
- 16 A. Israeli and Y. Shiloach. An improved parallel algorithm for maximal matching. *Inf. Process. Lett.*, 22(2):57–60, 1986.
- 17 R. M. Karp and J. E. Hopcroft. An  $n^{5/2}$  algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 1973.
- 18 F. Kuhn and T. Moscibroda. Distributed approximation of capacitated dominating sets. In *Proceedings of 19th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 161–170, 2007.
- 19 F. Kuhn, T. Moscibroda, and R. Wattenhofer. The price of being near-sighted. In *Proceedings of 17th Symposium on Discrete Algorithms (SODA)*, pages 980–989, 2006.

- 20 F. Kuhn, T. Moscibroda, and R. Wattenhofer. Local computation: Lower and upper bounds. *J. of the ACM*, 63(2), 2016.
- 21 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.
- 22 Z. Lotker, B. Patt-Shamir, and A. Rosén. Distributed approximate matching. *SIAM Journal on Computing*, 39(2):445–460, 2009.
- 23 M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM Journal on Computing*, 15:1036–1053, 1986.
- 24 Andrew McGregor. Finding graph matchings in data streams. In *Approximation, Randomization and Combinatorial Optimization. Algorithms and Techniques*, pages 170–181, 2005.
- 25 D. Peleg. *Distributed Computing: A Locality-Sensitive Approach*. SIAM, 2000.
- 26 S. Plotkin, D. Shmoys, and E. Tardos. Fast approximation algorithms for fractional packing and covering problems. *Mathematics of Operations Research*, 20:257–301, 1995.
- 27 Atish Das Sarma, Stephan Holzer, Liah Kor, Amos Korman, Danupon Nanongkai, Gopal Pandurangan, David Peleg, and Roger Wattenhofer. Distributed verification and hardness of distributed approximation. *SIAM J. Comput.*, 41(5):1235–1265, 2012.
- 28 M. Wattenhofer and R. Wattenhofer. Distributed weighted matching. In *Proceedings of 18th International Distributed Computing Conference (DISC)*, pages 335–348, 2004.
- 29 Yuichi Yoshida, Masaki Yamamoto, and Hiro Ito. An improved constant-time approximation algorithm for maximum matchings. In *STOC '09*, pages 225–234, 2009.



# State Machine Replication Is More Expensive Than Consensus

**Karolos Antoniadis**

EPFL, Lausanne, Switzerland  
karolos.antoniadis@epfl.ch

**Rachid Guerraoui**

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

**Dahlia Malkhi**

VMware Research, Palo Alto, USA  
dmalkhi@vmware.com

**Dragos-Adrian Seredinschi**

EPFL, Lausanne, Switzerland  
dragos-adrian.seredinschi@epfl.ch

---

## Abstract

Consensus and State Machine Replication (SMR) are generally considered to be equivalent problems. In certain system models, indeed, the two problems are computationally equivalent: any solution to the former problem leads to a solution to the latter, and vice versa.

In this paper, we study the relation between consensus and SMR from a *complexity* perspective. We find that, surprisingly, completing an SMR command can be more expensive than solving a consensus instance. Specifically, given a synchronous system model where every instance of consensus always terminates in constant time, completing an SMR command does *not* necessarily terminate in constant time. This result naturally extends to partially synchronous models. Besides theoretical interest, our result also corresponds to practical phenomena we identify empirically. We experiment with two well-known SMR implementations (Multi-Paxos and Raft) and show that, indeed, SMR is more expensive than consensus in practice. One important implication of our result is that – even under synchrony conditions – no SMR algorithm can ensure bounded response times.

**2012 ACM Subject Classification** Computing methodologies → Distributed algorithms

**Keywords and phrases** Consensus, State machine replication, Synchronous model

**Digital Object Identifier** 10.4230/LIPIcs.DISC.2018.7

**Related Version** A full version of the paper is available at [4], <https://infoscience.epfl.ch/record/256238>.

**Acknowledgements** We wish to thank our colleagues from the Distributed Computing Laboratory (DCL) for the many fruitful discussions. We are also thankful towards Willy Zwaenepoel and Jad Hamza, as well as the anonymous reviewers for providing us with helpful comments.



© Karolos Antoniadis, Rachid Guerraoui, Dahlia Malkhi, and Dragos-Adrian Seredinschi; licensed under Creative Commons License CC-BY

32nd International Symposium on Distributed Computing (DISC 2018).

Editors: Ulrich Schmid and Josef Widder; Article No. 7; pp. 7:1–7:18

Leibniz International Proceedings in Informatics



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

## 1 Introduction

Consensus is a fundamental problem in distributed computing. In this problem, a set of distributed processes need to reach agreement on a single value [32]. Solving consensus is one step away from implementing State Machine Replication (SMR) [49, 31]. Essentially, SMR consists of replicating a sequence of commands – often known as a log – on a set of processes which replicate the same state machine. These commands represent the ordered input to the state machine. SMR has been successfully deployed in applications ranging from storage systems, e.g., LogCabin built on Raft [43], to lock [13] and coordination [27] services. At a high level, SMR can be viewed as a sequence of consensus instances, so that each value output from an instance corresponds to a command in the SMR log.

From a *solvability* standpoint and assuming no malicious behavior, SMR can use consensus as a building block. When the latter is solvable, the former is solvable as well (the reverse direction is straightforward). Most previous work in this area, indeed, explain how to build SMR assuming a consensus foundation [21, 36, 33], or prove that consensus is equivalent from a solvability perspective with other SMR abstractions, such as atomic broadcast [14, 42]. An important body of work also studies the complexity of individual consensus instances [28, 22, 35, 47]. SMR is typically assumed to be a repetition of infinitely many consensus instances [29, 34] augmented with a reliable broadcast primitive [14], so at first glance it seems that the complexity of an SMR command can be derived from the complexity of the underlying consensus. We show that this is not the case.

In practice, SMR algorithms can exhibit irregular behavior, where some commands complete faster than others [12, 40, 54]. This suggests that the complexity of an SMR command can vary and may not necessarily coincide with the complexity of consensus. Motivated by this observation, we study the relation between consensus and SMR in terms of their *complexity*. To the best of our knowledge, we are the first to investigate this relation. In doing so, we take a formal, as well as a practical (i.e., experimental) approach. Counter-intuitively, we find that SMR is not necessarily a repetition of consensus instances.

We show that completing an SMR command can be more expensive than solving a consensus instance. Constructing a formalism to capture this result is not obvious. We prove our result by considering a fully synchronous system, where every consensus instance always completes in a *constant number of rounds*, and where at most one process in a round can be suspended (e.g., due to a crash or because of a network partition). A suspended process in a round is unable to send or deliver any messages in that round. Surprisingly, in this system model, we show that it is impossible to devise an SMR algorithm that can complete a command in constant time, i.e., completing a command can potentially require a *non-constant number of rounds*. We also discuss how this result applies in weaker models, e.g., partially synchronous, or if more than one process is suspended per round (see Section 3.2).

At a high level, the intuition behind our result is that a consensus instance “leaks,” so that some processing for that instance is deferred for later. Simply put, even if a consensus instance terminates, some protocol messages belonging to that instance can remain undelivered. Indeed, consensus usually builds on majority quorum systems [51], where a majority of processes is sufficient and necessary to reach agreement; any process which is not in this majority may be left out. Typically, undelivered messages are destined to processes which are not in the active majority – e.g., because they are slower, or they are partitioned from the other processes. Such a leak is inherent to consensus: the instance must complete after gathering a majority, and should not wait for additional processes. If a process is not in the active majority, that process might as well be faulty, e.g., permanently crashed.



In the context of an SMR algorithm, when successive consensus instances leak, the same process can be left behind across multiple SMR commands; we call this process a *straggler*. Consequently, the deferred processing accumulates. It is possible, however, that this straggler is in fact correct. This means that eventually the straggler can become part of the active quorum for a command. This can happen when another process fails and the quorum must switch to include the straggler. When such a switch occurs, the SMR algorithm might not be able to proceed before the straggler recovers the whole chain of commands that it misses. Only after this recovery completes can the next consensus instance (and SMR command) start. Another way of looking at our result is that a consensus instance can neglect stragglers, whereas SMR must deal with the potential burden of helping stragglers catch-up.<sup>1</sup>

We experimentally validate our result in two well-known SMR systems: a Multi-Paxos implementation (LibPaxos [3]) and a Raft implementation (etcd [2]). Our experiments include the wide-area and clearly demonstrate the difference in complexity, both in terms of latency and number of messages, between a single consensus instance and an SMR command. Specifically, we show that even if a single straggler needs to be included in an active quorum, SMR performance noticeably degrades. It is not unlikely for processes to become stragglers in practical SMR deployments, since these algorithms typically run on commodity networks [6]. These systems are subject to network partitions, processes can be slow or crashed, and consensus-based implementations can often be plagued with corner-cases or implementation issues [8, 30, 13, 25], all of which can lead to stragglers.

Our contribution in this paper is twofold. First, we initiate the study of the relation, in terms of complexity, between consensus and SMR. We devise a formalism to capture the difference in complexity between these two problems, and use this formalism to prove that completing a single consensus instance is not equivalent to completing an SMR command in terms of their complexity (i.e., number of rounds). More precisely, we prove that it is impossible to design an SMR algorithm that can complete a command in constant time, even if consensus always completes in constant time. Second, we experimentally validate our theoretical result using two SMR systems in both a single-machine and a wide-area network.

**Roadmap.** The rest of this paper is organized as follows. We describe our system model in Section 2. In Section 3 we present our main result, namely that no SMR algorithm can complete every command in a constant number of rounds. Section 4 presents experiments to support our result. We describe the implications of our result in Section 5, including ways to circumvent it and a trade-off in SMR. Finally, Section 6 concludes the paper.

## 2 Model

This paper studies the relation in terms of complexity between consensus and State Machine Replication (SMR). In this section we formulate a system model that enables us to capture this relation, and also provide background notions on consensus and SMR.

We consider a synchronous model and assume a finite and fixed set of processes  $\Pi = \{p_1, p_2, \dots, p_n\}$ , where  $|\Pi| = n \geq 3$ . Processes communicate by exchanging messages. Each message is taken from a finite set  $M = \{m_1, \dots\}$ , where each message has a positive and a bounded size, which means that there exists a  $B \in \mathbb{N}^+$  such that  $\forall m \in M, 0 < |m| \leq B$ .

<sup>1</sup> We note that this leaking property seems not only inherent in consensus, but in any equivalent replication primitive, such as atomic broadcast.

A process is a state machine that can change its state as a consequence of delivering a message or performing some local computation. Each process has access to a read-only global clock, called *round number*, whose value increases by one on every round. In each round, every process  $p_i$ : (1) sends one message to every other process  $p_j \neq p_i$  (in total  $p_i$  sends  $n - 1$  messages in each round);<sup>2</sup> (2) delivers any messages sent to  $p_i$  in that round; and (3) performs some local computation.

An *algorithm* in such a model is the state machine for each process and its initial state. A *configuration* corresponds to the internal state of all processes, as well as the current round number. An *initial configuration* is a configuration where all processes are in their initial state and the round number is one. In each round, up to  $n(n - 1)$  messages are transmitted. More specifically, we denote a transmission as a triplet  $(p, q, m)$  where  $p, q \in \Pi(p \neq q)$  and  $m \in M$ . For instance, transmission  $(p_i, p_j, m_{i,j})$  captures the sending of message  $m_{i,j}$  from process  $p_i$  to process  $p_j$ . We associate with each round an *event*, corresponding to the set of transmissions which take place in that round; we denote this event by  $\tau \subseteq \{(p_i, p_j, m_{i,j}) : i, j \in \{1, \dots, n\} \wedge i \neq j\}$ . An *execution* corresponds to an alternating sequence of configurations and events, starting with an initial configuration. An execution  $e^+$  is called an *extension* of a finite execution  $e$  if  $e$  is a prefix of  $e^+$ . Given a finite execution  $e$ , we denote with  $E(e)$  the set of all extensions of  $e$ . We assume deterministic algorithms: the sequence of events uniquely defines an execution.

**Failures.** Our goal is to capture the *complexity* – i.e., cost in terms of number of synchronous rounds – of a consensus instance and of an SMR command, and expose any differences in terms of this complexity. Towards this goal, we introduce a failure mode which omits *all* transmissions to and from at most one process per round.

We say that a process  $p_i$  is *suspended in round  $r$*  associated with the event  $\tau$ , if  $\forall m \in M$  and  $\forall j \in \{1, \dots, n\}$  with  $j \neq i$ ,  $(p_i, p_j, m) \notin \tau$  and  $(p_j, p_i, m) \notin \tau$ , hence  $|\tau| = n(n - 1) - 2(n - 1) = (n - 1)(n - 2)$ . If a process  $p_i$  is not suspended in a round  $r$ , we say that  $p_i$  is *correct in round  $r$* . In a round associated with an event  $\tau$  where all processes are correct there are no omissions, hence  $|\tau| = n(n - 1)$ . A process  $p_i$  is *correct* in a finite execution  $e$  if there is a round in  $e$  where  $p_i$  is correct. Process  $p_i$  is *correct* in an infinite execution  $e$  if there are infinitely many rounds in  $e$  where  $p_i$  is correct. For our result, it suffices that in each round a single process is suspended. Note that each round in our model is a communication-closed layer [18], so messages omitted in a round are not delivered in any later round.

A suspended process represents a scenario where a process is slowed down. This may be caused by various real-world conditions, e.g., a transient network disconnect, a load imbalance, or temporary slowdown due to garbage collection. In all of these, after a short period, connections are dropped and message buffers are reclaimed; such conditions can manifest as message omissions. The notion of being suspended also represents a model where processes may crash and recover, where any in-transit messages are typically lost.

There is a multitude of work [44, 45, 47, 9, 48] on message omissions (e.g., due to link failures) in synchronous models. Our system model is based on the mobile faults model [44]. Note however that our model is stronger than the mobile faults model, since we consider that either exactly zero or exactly  $2(n - 1)$  message omissions occur in a given round.<sup>3</sup> Other powerful frameworks, such as layered analysis [41], the heard-of model [15], or RRFD [20] can be used to capture omission failures, but we opted for a simpler approach that can specifically express the model which we consider.

<sup>2</sup> As a side note, if a process  $p_i$  does not have something to send to process  $p_j$  in a given round, we simply assume that  $p_i$  sends an empty message.

<sup>3</sup> If a process  $p$  is suspended, then  $n - 1$  messages sent by  $p$  and  $n - 1$  messages delivered to  $p$  are omitted.

**Algorithm 1** Consensus.

---

```

1: procedure PROPOSE( $p_i, v_i$ ) ▷  $p_i$  proposes value  $v_i$ 
2:    $decision \leftarrow \perp$ 
3:   ▷ round 1
4:    $\forall p \in \Pi \setminus \{p_i\}, \text{send}(p, v_i)$  ▷  $\Pi$  is the set of processes
5:    $values \leftarrow \{v_i\} \cup \{ \text{each value } v \text{ delivered from process } p (\forall p \in \Pi \setminus \{p_i\}) \}$ 
6:   if  $|values| \neq 1$  then ▷  $p_i$  is correct in round 1
7:      $decision \leftarrow \text{deterministicFunction}(values)$ 
8:   else ▷  $p_i$  was suspended
9:     ▷  $p_i$  cannot decide yet
10:  ▷ round  $k$  ( $k \geq 2$ ): consensus instance completes in round 2
11:   $\forall p \in (\Pi \setminus \{p_i\}) \cup \{client\}, \text{send}(p, decision)$  ▷ broadcast decided value
12:   $values \leftarrow \{decision\} \cup \{ \text{each decision } d \text{ delivered from process } p (\forall p \in \Pi \setminus \{p_i\}) \}$ 
13:   $decision \leftarrow d$  where  $d \in values$  and  $d \neq \perp$ 

```

---

## 2.1 Consensus

In the consensus problem, processes have initial values which they propose, and have to decide on a single value. Consensus [10] is defined by three properties: validity, agreement, and termination. Validity requires that a decided value was proposed by one of the processes, whilst agreement asks that no two processes decide differently. Finally, termination states that every correct process eventually decides. In the interest of having an “apples to apples” comparison with SMR commands (defined below, Section 2.2), we introduce a client (e.g., learner in Paxos terminology [33]), and say that a consensus instance completes as soon as the client learns about the decided value. This client is not subject to being suspended, and after receiving the decided value, the client broadcasts this value to the other processes. Algorithm 1 is a consensus algorithm based on this idea.

It is easy to see that in such a model consensus completes in two rounds: processes broadcast their input, and every process uses some deterministic function (e.g., maximum) to decide on a specific value among the set of values it delivers. Since all processes deliver exactly the same set of  $n - 1$  (or  $n$ ) values, they reach agreement. In the second round, all processes send their decided value (a process that was suspended in the first round might send  $\perp$ ) to all the other processes, including the client. Since  $n \geq 3$  and at least  $n - 1$  processes are correct in the second round, the client delivers the decided value (i.e., a value that is not  $\perp$ ) and thus the consensus instance completes by the end of round two. Afterwards (starting from the third round), the client broadcasts the decided value to all the processes, so eventually every correct process decides, satisfying termination. Note that if a process is suspended in the first round (but correct in the second round), it will decide in the second round, after delivering the decided value from some other process. Algorithm 1 represents this solution in which the red and blue lines correspond to the synchronous model’s send and deliver actions respectively.

We remark that Algorithm 1 does not contradict the lossy link impossibility result of Santoro and Widmayer [44], even though our model permits more than  $n - 1$  message omissions in a round, since the model we consider is stronger.

We emphasize that although correct processes can decide in the first round, we consider that the consensus instance *completes* when the client delivers the decided value. Hence, the consensus instance in Algorithm 1 completes in the second round. In more practical terms, this consensus instance has a constant cost.

## 2.2 State Machine Replication

The SMR approach requires a set of processes (i.e., replicas) to agree on an ordered sequence of commands [31, 49]. We use the terms replica and process interchangeably. Informally, each replica has a log of the commands it has performed, or is about to perform, on its copy of the state machine.

**Log.** Each replica is associated with a sequence of decided and known commands which we call the *log*. The commands are taken from a finite set  $C = \{c_1, \dots, c_k\}$ . We denote the log with  $\ell(e, p)$  where  $e$  is a finite execution,  $p$  is a replica, and each element in  $\ell(e, p)$  belongs to the set  $C \cup \{\epsilon\}$ . Specifically,  $\ell(e, p)$  corresponds to commands known by replica  $p$  after all the events in a finite execution  $e$  have taken place (e.g.,  $\ell(e, p) = c_{i_1}, \epsilon, c_{i_3}$ ). For  $1 \leq i \leq |\ell(e, p)|$ , we denote with  $\ell(e, p)_i$  the  $i$ -th element of sequence  $\ell(e, p)$ . If there is an execution  $e$  and  $\exists p \in \Pi$  and  $\exists i \in \mathbb{N}^+$  such that  $\ell(e, p)_i = \epsilon$ , this means that replica  $p$  does not have knowledge of the command for the  $i$ -th position in its log, while at least one replica does have knowledge of this command (i.e.,  $\exists p' \neq p \in \Pi : \ell(e, p')_i \neq \epsilon$ ). We assume that if a process knows about a command  $c$ , then  $c$  exists in  $\ell(e, p)$ . To keep our model at a high-level, we abstract over the details of how each command appears in the log of each replica, since this is typically algorithm-specific. Additionally, state-transfer optimizations or snapshotting [43] are orthogonal to our discussion.

An SMR algorithm is considered *valid* if the following property is satisfied for any finite execution  $e$  of that algorithm:  $\forall p, p' \in \Pi$  and for every  $i$  such that  $1 \leq i \leq \min(|\ell(e, p)|, |\ell(e, p')|)$ , if  $\ell(e, p)_i \neq \ell(e, p')_i$  then either  $\ell(e, p)_i = \epsilon$  or  $\ell(e, p')_i = \epsilon$ . In other words, consider a replica  $p$  which knows a command for a specific log position  $i$ , i.e.,  $\ell(e, p)_i = c_k$ , where  $c_k \in C$ . Then for the same log position  $i$ , any other process  $p'$  can either know command  $c_k$  (i.e.,  $\ell(e, p')_i = c_k$ ), not know the command (i.e.,  $\ell(e, p')_i = \epsilon$ ), or have no information regarding the command (i.e.,  $|\ell(e, p')| < i$ ). In this paper, we only consider valid SMR algorithms.

In what follows, we define what it means for a replica to be a *straggler*, as well as how replicas first learn about commands.

**Stragglers.** Intuitively, stragglers are replicas that are missing commands from their log. More specifically, let  $L$  be  $\max_p |\ell(e, p)|$ . We say that  $q$  is a  $k$ -*straggler* if the number of non- $\epsilon$  elements in  $\ell(e, q)$  is at most  $L - k$ . A replica  $p$  is a *straggler* in an execution  $e$  if there exists a  $k \geq 1$  such that  $p$  is a  $k$ -straggler. Otherwise, we say that the replica is a *non-straggler*. A replica that is suspended for a number of rounds could potentially miss commands and hence become a straggler.

**Client.** Similar to the consensus client, there is a client process in SMR as well. In SMR, however, the client proposes commands. The client acts like the  $(n + 1)$ -th replica in a system with  $n$  replicas and its purpose is to supply one command to the SMR algorithm, wait until it receives (i.e., delivers) a response for the command it sent, then send another command, etc. A client, however, is different from the other replicas, since an SMR algorithm has no control over the state machine operating in the client and the client is never suspended. A client operates in lock-step<sup>4</sup> as follows:

<sup>4</sup> Clients need not necessarily operate in lock-step, but can employ pipelining, i.e., can have multiple commands outstanding. Practical systems employ pipelining [43, 3, 2], and we account for this aspect later in our practical experiments of Section 4.

- sends a command  $c \in C$  to all the  $n$  replicas in some round  $r$ ;
- waits until some replica responds to the client's command (i.e., the response of applying the command).<sup>5</sup>

A replica  $p$  can respond to a client command  $c$  only if it has all commands preceding  $c$  in its log. This means that  $\exists i : \ell(e, p)_i = c$  and  $\forall j < i, \ell(e, p)_j \neq \epsilon$ . We say that the client is *suggesting* a command  $c$  at a round  $r$  if the client sends a message containing command  $c$  to all the replicas in round  $r$ . Similarly, we say that a client *gets a response* for command  $c$  at a round  $r$  if some replica sends a message to the client containing the response of the command  $c$  in round  $r$ .

**SMR Algorithm.** Algorithm 1 shows that consensus is solvable in our model. It seems intuitive that SMR is solvable in our model as well. To prove that this is the case, we introduce an SMR algorithm. Roughly speaking, this algorithm operates as follows. Each replica contains an ordered log of decided commands. A command is decided for a specific log position by executing a consensus instance similar to Algorithm 1. The SMR algorithm takes care of stragglers through the use of helping. Specifically, each replica tries to help stragglers by sending commands which the straggler might be missing. Due to space constraints, we defer the detailed description and the proof of the SMR algorithm, which can be seen as a contribution in itself, to our corresponding technical report [4]. As we show next (Section 3), no SMR algorithm can respond to a client in a finite number of rounds. Hence, even with helping, our SMR algorithm cannot guarantee a constant response either. Finally, note that our definition of a valid SMR algorithm does not include a liveness property since this is not needed for our result. Nevertheless, the SMR algorithm we propose guarantees that if a client suggests a command, then the client eventually gets a response.

### 3 Complexity Lower Bound on State Machine Replication

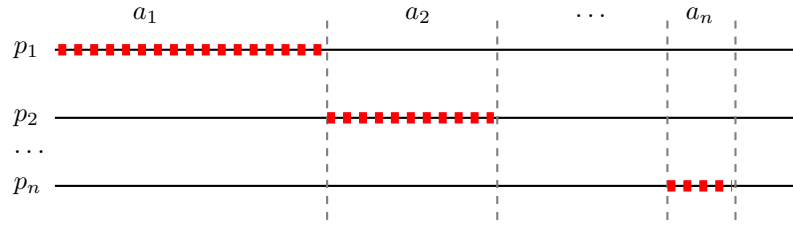
We now present the main result of our paper. Roughly speaking, we show that there is no State Machine Replication (SMR) algorithm that can always respond to a client in a constant number of rounds. We also discuss how this result extends beyond the model of Section 2.

#### 3.1 Complexity Lower Bound

We briefly describe the idea behind our result. We observe that there is a bounded number of commands that can be delivered by a replica in a single round, since messages are of bounded size, a practical assumption (Lemma 1). Using this observation, we show that in a finite execution  $e$ , if each replica  $p_i$  is missing  $\beta_i$  commands, then an SMR algorithm needs  $\Omega(\min_i \beta_i)$  rounds to respond to at least one client command suggested in an extension  $e^+ \in E(e)$  (Lemma 2). Finally, for any  $r \in \mathbb{N}^+$ , we show how to construct an execution  $e$  where each replica misses enough commands in  $e$ , so that a command suggested by a client in an extension  $e^+ \in E(e)$  cannot get a response in less than  $r$  rounds (Theorem 3). Hence, no SMR algorithm in our model can respond to every client command in a constant number of rounds.

► **Lemma 1.** *A single replica can deliver up to a bounded number (that we denote by  $\Psi$ ) of commands in a round.*

<sup>5</sup> We consider that a command is applied instantaneously on the state machine (i.e., execution time for any command is zero).



■ **Figure 1** Constructed execution of Theorem 3. Red dashed lines correspond to rounds where a replica is suspended. Replica  $p_1$  is suspended for  $a_1$  rounds, replica  $p_2$  for  $a_2$  rounds, etc.

**Proof.** Since any message  $m$  is of bounded size  $B$  ( $\forall m \in M, |m| \leq B$ ), the number of commands message  $m$  can contain is bounded. Let us denote with  $\psi$  the maximum number of commands any message can contain. Since the number of commands that can be contained in one message is at most  $\psi$ , a replica can transmit at most  $\psi$  commands to another replica in one round. Therefore, in a given round a replica can deliver from other replicas up to  $\Psi = (n-1)\psi$  commands. In other words, a replica cannot recover faster than  $\Psi$  commands per round. ◀

► **Lemma 2.** *For any finite execution  $e$ , if each replica  $p_i$  is a  $\beta_i$ -straggler (i.e.,  $p_i$  misses  $\beta_i$  commands), then there is a command suggested by the client in some execution  $e^+ \in E(e)$  such that we need at least  $\lceil \min_i(\beta_i/\Psi) \rceil$  rounds to respond to it.*

**Proof.** Consider an execution  $e^+ \in E(e)$  such that in a given round  $r$ , a client suggests to all replicas a command  $c$ , where round  $r$  exists in  $e^+$  but does not exist in  $e$ . This implies that replicas are not yet aware of command  $c$  in  $e$ , so command  $c$  should appear in a log position  $i$  where  $i$  is greater than  $\max_p |\ell(e, p)|$ . In order for a replica to respond to the client's command  $c$ , the replica first needs to have all the commands preceding  $c$  in its log. For this to happen, some replica needs to get informed about  $\beta_i$  commands. Note that from Lemma 1, a replica can only deliver  $\Psi$  commands in a round. Therefore, a replica needs at least  $\lceil \beta_i/\Psi \rceil$  rounds to get informed about the commands it is missing (i.e., recover), and hence we need at least  $\lceil \min_i(\beta_i/\Psi) \rceil$  rounds for the client to get a response for  $c$ . ◀

► **Theorem 3.** *For any  $r \in \mathbb{N}^+$  and any SMR algorithm with  $n$  replicas ( $n \geq 3$ ), there exists an execution  $e$ , such that a command  $c$  which the client suggests in some execution  $e^+ \in E(e)$  cannot get a response in less than  $r$  rounds.*

**Proof.** Assume by contradiction that, given an SMR algorithm, each command suggested by a client needs at most a constant number of rounds  $k$  to get a response. Since we can get a response to a command in at most  $k$  rounds, we can make a replica “miss” any number of commands by simply suspending it for an adequate amount of rounds.

To better convey the proof we introduce the notion of a phase. A phase is a conceptual construct that corresponds to a number of contiguous rounds in which a specific replica is suspended. Specifically, we construct an execution  $e$  consisting of  $n$  phases. Figure 1 conveys the intuition behind this execution. In the  $i$ -th phase, replica  $p_i$  is suspended for  $\alpha_i$  rounds, and  $\alpha_i \neq \alpha_j$  for  $i \neq j$ . The idea is that after the  $n$ -th phase, each replica is a straggler and needs more than  $k$  rounds to become a non-straggler and be able to respond to a client command suggested in a round  $o$ , where  $o$  exists in  $e^+$  but not in  $e$ . We start from the  $n$ -th phase, going backwards. In the  $n$ -th phase, we make replica  $p_n$  miss enough commands, say  $\beta_n$ . In general, the number  $\beta_n$  of commands is such, that if a client suggests a command



at the end of the  $n$ -th phase, the client cannot get a response from within  $k$  rounds of the command being suggested. For this to hold, it suffices to miss  $\beta_n = k\Psi + 1$  commands. In order to miss  $\beta_n$  commands, we have to suspend  $p_n$  for at least  $\beta_n k$  rounds, since a client may submit a new command every (at most)  $k$  rounds. Thus, we set  $\alpha_n = \beta_n k$ . Similarly, replica  $p_{n-1}$  has to miss enough commands ( $\beta_{n-1}$ ) such that it cannot get all the commands in less than  $k$  rounds. Note that after  $p_{n-1}$  was suspended for  $\alpha_{n-1}$  rounds, replica  $p_n$  took part in  $\alpha_n$  rounds. During these  $\alpha_n$  rounds, replica  $p_{n-1}$  could have recovered commands it was missing. Therefore,  $p_{n-1}$  must miss at least  $\beta_{n-1} = (\alpha_n + k)\Psi + 1$  commands and  $\alpha_{n-1} = \beta_{n-1} k$ . In the same vein,  $\forall i \in \{1, \dots, n\}$   $\beta_i = ((\sum_{j=i+1}^n \alpha_j) + k)\Psi + 1$ .

With our construction we succeed in having  $\beta_i/\Psi = (\sum_{j=i+1}^n \alpha_j) + k + 1/\Psi > k$  for every  $i \in \{1, \dots, n\}$ . Therefore, using Lemma 2, after the  $n$  phases, each replica needs more than  $k$  rounds to get informed about commands it is missing from its log, a contradiction. ◀

Theorem 3 states that there exists no SMR algorithm in our model that can respond to every client command in a constant number of rounds.

### 3.2 Extension to other Models

The system model we use in this paper (Section 2) lends itself to capture naturally the difference in complexity (i.e., number of rounds) between consensus and SMR. It is natural to ask whether this difference extends to other system models – and which are those models. Identifying all the models where our result applies, or does not apply, is a very interesting topic which is beyond the scope of this paper, but we briefly discuss it here.

Consider models which are stronger than ours. An example of a stronger model is one that is synchronous with no failures; such a model would disallow stragglers and hence both consensus and SMR can be solved in constant time. Similarly, if the model does not restrict the size of messages (see Lemma 1), then an SMR command can complete in constant time, circumventing our result. We further discuss how our result can be circumvented in Section 5.

A more important case is that of weaker, perhaps more realistic models. If the system model is too weak – if consensus is not solvable [19] – then it is not obvious how consensus relates to SMR in terms of complexity. Such a weak model, however, can be augmented, for instance with unreliable failure detectors [14], allowing consensus to be solved. Informally, during well-behaved executions of such models, i.e., executions when the system behaves synchronously and no failures occur [28], SMR commands can complete in constant time.

Most practical SMR systems [16, 13, 43, 40] typically assume a partially synchronous or an asynchronous model with failure detectors [14], and executions are not well-behaved, because failures are prone to occur [6]. We believe our result applies in these practical settings, concretely within synchronous periods (or when the failure detector is accurate, respectively) of these models. During such periods, if at least one replica can suffer message omissions, completing an SMR command can take a non-constant amount of time. Indeed, in the next section, we present an experimental evaluation showing that our result holds in a partially synchronous system.

## 4 The Empirical Perspective

Our goal in this section is to substantiate empirically the theoretical result of Section 3. We first cover details of the experimental methodology. Then we discuss the evaluation results both in a single-machine environment, as well as on a practical wide-area network (WAN).

## 4.1 Experimental Methodology

We use two well-known State Machine Replication (SMR) systems: (1) LibPaxos, a Multi-Paxos implementation [3], and (2) etcd [2], a mature implementation of the Raft protocol [43]. We note that LibPaxos distinguishes between three roles of a process: proposer, acceptor, and learner [33]. To simplify our presentation, we unify the terminology so that we use the term *replica* instead of *acceptor*, the term *client* replaces *learner*, and the term *leader* replaces *proposer*. Each system we deploy consists of three replicas, since this is sufficient to validate our result and moreover it is a common deployment practice [16, 23]. We employ one client. In LibPaxos, we use a single leader, which corresponds to a separate role from replicas. In Raft, one of the three replicas acts as the leader.

Using these two systems, we measure how consensus relates to SMR in terms of cost in the following three scenarios:

1. **Graceful:** when network conditions are uniform and no failures occur; this scenario only applies to the single-machine experiments of Section 4.2;
2. **Straggler:** a single replica is slower than the others (i.e., this is a straggler) but no failures occur, so the SMR algorithm needs not rely on the straggler;
3. **Switch:** a single replica is a straggler and a failure occurs, so the SMR algorithm has to include the straggler on the critical path of agreement on commands.

Due to the difficulty of running synchronous rounds in a practical system, our measurements are not in terms of rounds (as in the model of Section 2). Instead, we take a lower-level perspective. We report on the *cost*, i.e., number of messages, and the *latency* measured at the client.<sup>6</sup> Specifically, in each experiment, we report on the following three measurements.

First, we present the cost of each consensus instance  $i$  in terms of number of messages which belong to instance  $i$ , and which were exchanged between replicas, as well as the client. Each consensus instance has an identifier (called *iid* in LibPaxos and *index* in Raft), and we count these messages up to the point where the instance completes at the client. Recall that in our model (Section 2.1) we similarly consider consensus to complete when the client learns the decided value. This helps us provide an “apples to apples” comparison between the cost of consensus instances and SMR commands (which we describe next).

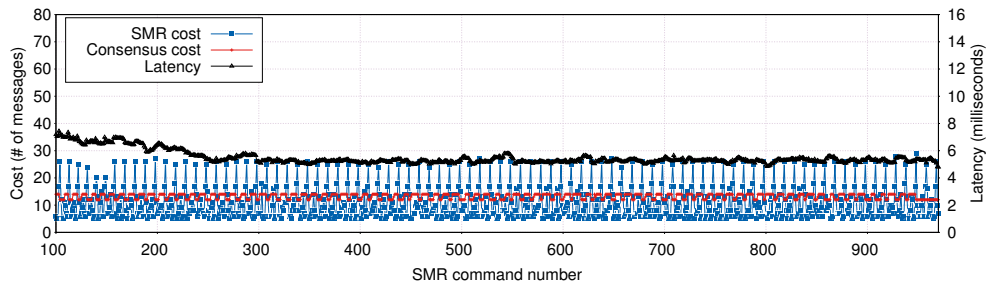
Second, we measure the cost of each SMR command  $c$ . Each command  $c$  is associated with a consensus instance  $i$ . The cost of  $c$  is similar to the cost of  $i$ : we count messages exchanged between replicas and the client for instance  $i$ .<sup>7</sup> The cost of a command  $c$ , however, is a more nuanced measurement. As we discussed already, a consensus instance typically leaks messages, which can be processed later. Also, both systems we consider use pipelining, so that a consensus instance  $i$  may overlap with other instances while a replica is working on command  $c$ . Specifically, the cost of  $c$  can include messages leaked from some instance  $j$ , where  $j < i$  (because a replica cannot complete command  $c$  without having finished all previous instances) but also from some instance  $k$ , with  $k > i$  (these future instances are being prepared in advance in a pipeline, and are not necessary for completing command  $c$ ).

Third, we measure the latency for completing each SMR command. An SMR command starts when the client submits this command to the leader, and ends when the client learns the command. In LibPaxos, this happens when the client gathers replies for that command from two out of three replicas; in Raft, the leader notifies the client with a response.

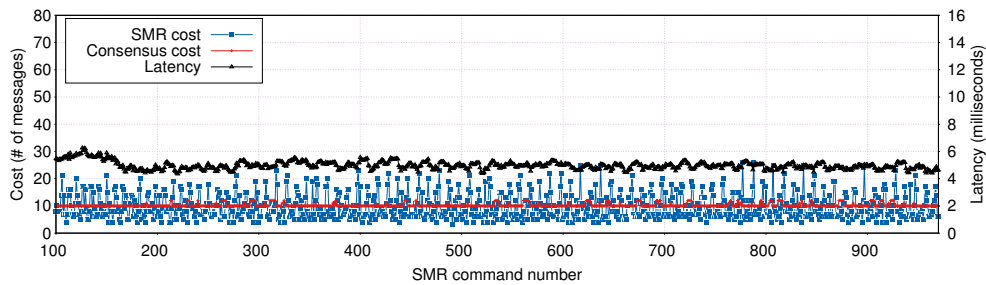
<sup>6</sup> Note that it is simple to convert rounds to messages, considering our description of rounds in Section 2.

<sup>7</sup> For LibPaxos, the cost of consensus and SMR includes additionally messages involving the leader.

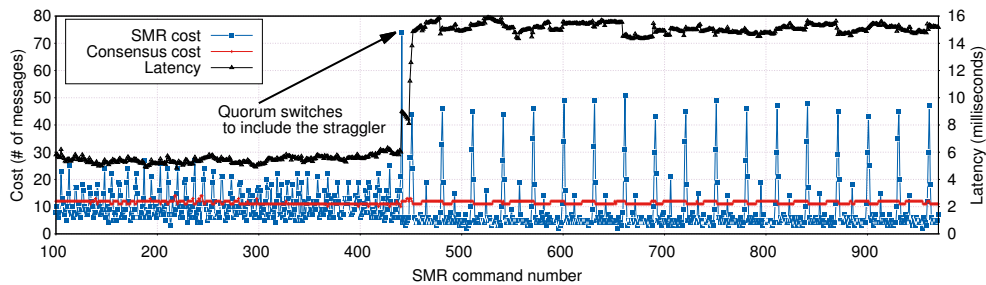




(a) **Graceful** scenario: all replicas experience uniform conditions and no failures occur.



(b) **Straggler** scenario: one of the three replicas is a straggler.



(c) **Switch** scenario: one of the three replicas is a straggler and the active quorum switches to include this straggler.

■ **Figure 2** Experimental results with LibPaxos on a single-machine setup. We compare the cost of SMR commands with the cost of consensus instances in three scenarios.

We consider both a single-machine setup and a WAN. The former setup serves as a controlled environment where we can vary specifically the variable we seek to study, namely the impact of a straggler when quorums switch. For this experiment, we use LibPaxos and we discuss the results thoroughly. The latter setup reflects real-world conditions which we use to validate against our findings in the single-machine setup, and we experiment with both systems. In all executions the client submits 1000 SMR commands; we ignore the first 100 (warm-up) and the last 50 commands (cool-down) from the results. We run the same experiment three times to confirm that we are not presenting outlying results.

## 4.2 Experimental Results on a Single Machine

We experiment on an Intel Core i7-3770K (3.50GHz) equipped with 16GB of RAM. Since there is no network in these experiments, spurious network conditions – which can arise in practice, as we shall see next in Section 4.3 – do not create noise in our results. To make one of the replicas a straggler, we make this replica relatively slower through a random delay (via the `select` system call) of up to 500 $\mu$ s when this replica processes a protocol message.

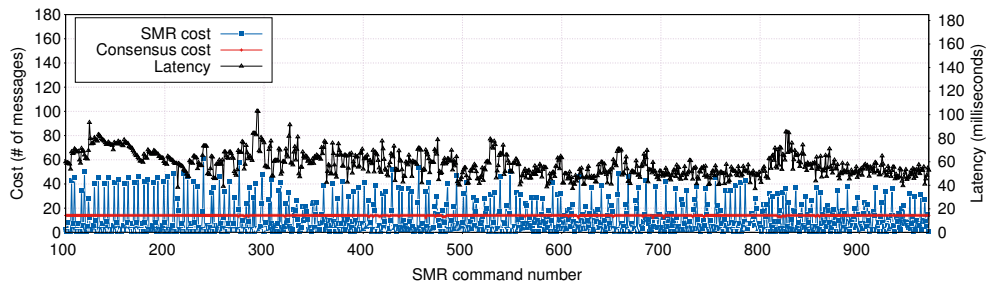
In Figure 2a we show the evolution of the three measurements we study for the **graceful** execution. The mean latency is 5590 $\mu$ s with a standard deviation of 730 $\mu$ s, i.e., the performance is very stable. This execution serves as a baseline.

In Figure 2b we present the result for the **straggler** scenario. The average latency, compared with Figure 2a, is slightly smaller, at 5005 $\mu$ s; the standard deviation is 403 $\mu$ s. The explanation for this decrease is that there is less contention (because the straggler backs-off periodically), so the performance increases. In this scenario, additionally, there is more variability in the cost of SMR commands, which is a result of the straggler replica being less predictable in how many protocol messages it handles per unit of time.

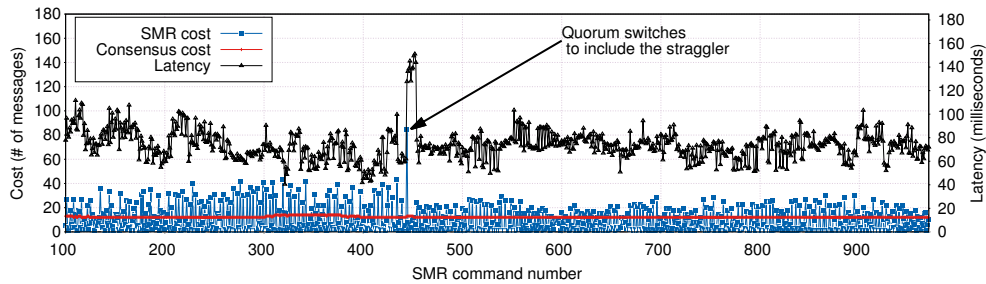
For both Figures 2a and 2b, the average cost of an SMR command is the same as the average cost of a consensus instance, specifically around 12 messages. There is, however, a greater variability in the cost of SMR commands – ranging from 5 to 30 messages – while consensus instances are more regular – between 11 and 13 messages. As we mentioned already, the variability in the cost of SMR springs from two sources: (1) consensus instances leak into each other, and (2) the use of pipelining, a crucial part in any practical SMR algorithm, which allows consensus instances to overlap in time [27, 46].

Pipelining allows the leader to have multiple outstanding proposals, and these are typically sent and delivered in a burst, in a single network-level packet. This means that some commands can comprise just a few messages (all the other messages for such a command have been processed earlier with previous commands, or have been deferred), whereas some commands comprise many more messages (e.g., messages leaked from previous commands, or processed in advance from upcoming commands). In our case, the pipeline has size 10, and we can distinguish in the plots that the bumps in the SMR cost have this frequency. Larger pipelines allow higher variability in the cost of SMR. Importantly, to reduce the effect of pipelining on the cost of SMR commands, this pipeline size of 10 is much smaller than it is used in practice, which can be 64, 128, or larger [2, 3].

Figure 2c shows the execution where we stop one replica, so the straggler has to take part in the active quorum. The moment when the straggler has to recover all the missing state and start participating is evident in the plot. This happens at SMR command 450. We observe that SMR command 451 has considerably higher cost. This cost comprises all the messages which the straggler requires to catch-up, before being able to participate in the next consensus instance. The cost of consensus instance 451 itself is no different than other consensus instances. Since the straggler becomes the bottleneck, the latency increases and remains elevated for the rest of the execution. The average latency in this case is noticeably higher than in the two previous executions, at 10730 $\mu$ s (standard deviation of 4726 $\mu$ s). For this execution, we observe the same periodical bumps in the cost of SMR commands. Because the straggler replica is on the critical path of agreement, these bumps are more pronounced and less frequent: the messages concerning the straggler (including to and from other replicas or the client) accumulate in the incoming and outgoing queues and are processed in bursts.



(a) **Straggler** scenario: the replica in Frankfurt is a straggler, since this is the farthest from the leader in Ireland. The system forms a quorum using the replicas in London and Paris.



(b) **Switch** scenario: at SMR command 450 we switch out the replica in London. The straggler in Frankfurt then becomes part of the active quorum.

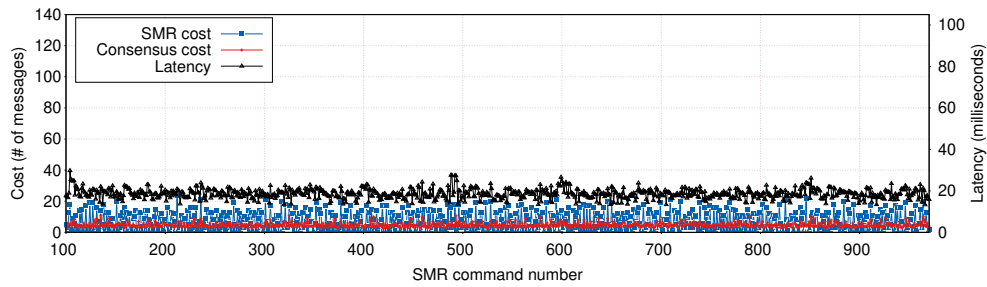
■ **Figure 3** Experimental results with LibPaxos on the WAN. Similar to Figure 2, we compare the cost of SMR commands with the cost of consensus instances.

### 4.3 Wide-area Experiments

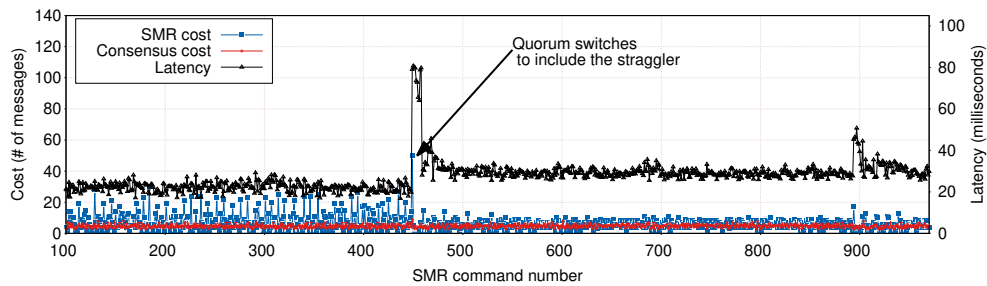
We deploy both LibPaxos and Raft on Amazon EC2 using *t2.micro* virtual machines [1]. For LibPaxos, we collocate the leader with the client in Ireland, and we place the three replicas in London, Paris, and Frankfurt, respectively. Similarly, for Raft we collocate the leader replica along with the client in Ireland, and we place the other two replicas in London and Frankfurt. Under these deployment conditions, the replica in Frankfurt is naturally the straggler, since this is the farthest node from Ireland (where the leader is in both systems). Therefore, we do not impose any delays, as we did in the earlier single-machine experiments. Furthermore, collocating the client with the leader minimizes the latency between these two, so the latency measurements we report indicate the actual latency of SMR.

Figures 3 and 4 present our results for LibPaxos and Raft, respectively. To enhance visibility, please note that we use different scales for the  $y$  and  $y_2$  axes. These experiments do not include the **graceful** scenario, because the WAN is inherently heterogeneous.

The most interesting observation is for the **switch** scenarios, i.e., Figures 3b and 4b. In these experiments, when we stop one of the replicas at command 450, there is a clear spike in the cost of SMR, which is similar to the spike in Figure 2c. Additionally, however, there is also a spike in latency. This latency spike does not manifest in single-machine experiments, where communication delays are negligible. Moreover, on the WAN the latency spike extends over multiple commands, because the system has a pipeline so the latency of each command being processed in the pipeline is affected while the straggler is catching up. After this spike, the latency decreases but remains slightly more elevated than prior to the switch, because the active quorum now includes the replica from Frankfurt, which is slightly farther away; the difference in latency is roughly  $5ms$ .



(a) **Straggler** scenario: the replica in Frankfurt is a straggler. The active quorum consists of the leader in Ireland and the replica in London.



(b) **Switch** scenario: we stop the replica in London at SMR command 450. Thereafter, the active quorum must switch to include the straggler in Frankfurt.

■ **Figure 4** Experimental results with Raft on the WAN. Similar to Figures 2 and 3, we compare the cost of SMR commands with the cost of consensus instances.

Beside the latency spike at SMR command 450, these experiments reveal a few other glitches, for instance around command 830 in Figure 3a, or command 900 in Figure 4b. In fact, we observe that unlike our single-machine experiments, the latency exhibits a greater variability. As we mentioned already, this has been observed before [12, 40, 54] and is largely due to the heterogeneity in the network and the spurious behavior this incurs. This effect is more notable in LibPaxos, but Raft also shows some variability. The latter system reports consistently lower latencies because an SMR command completes after a single round-trip between the leader and replicas [43].

As a final remark, our choice of parameters is conservative, e.g., execution length or pipeline width. For instance, in executions longer than 1000 commands we can exacerbate the difference in cost between SMR commands and consensus instances. Longer executions allow a straggler to miss even more state which it needs to recover when switching.

## 5 Discussion

The main implication of Theorem 3 is that it is impossible to devise a State Machine Replication (SMR) algorithm that can bound its response times. There are several conditions, however, which allow to circumvent our lower bound, which we discuss here. Moreover, when our result does apply, we observe that SMR algorithms can mitigate, to some degree, the performance degradation in the worst-case, i.e., when quorums switch and stragglers become necessary. These algorithms experience a trade-off between best-case and worst-case performance. We also discuss how various SMR algorithms deal with this trade-off.

**Circumventing the Lower Bound.** Informally, our result applies to SMR systems which fulfill two basic characteristics: i) messages are bounded in size, and ii) replicas can straggle for arbitrary lengths of time. Simply put, if one of these conditions does not hold, then we can circumvent Theorem 3. We discuss several cases when this can happen.<sup>8</sup>

For instance, if the total size of the state machine is bounded, as well as small in size, then the whole state machine can potentially fit in a single message, so a straggler can recover in bounded time. This is applicable in limited practical situations. We are not aware of any SMR protocol that caps its state. But this state can be very small in some applications, e.g., if SMR is employed towards replicating only a critical part of the application, such as distributed locks or coordination kernels [27, 39].

The techniques of load shedding or backpressure [53] can be employed to circumvent our result. These are application-specific techniques which, concretely, allow a system to simply drop or deny a client command if the system cannot fulfill that command within bounded time. Other, more drastic, approaches to enforce strict latencies involve resorting to weak consistency or combining multiple consistency models in the same application [24], or provisioning additional replicas proactively when stragglers manifest [17, 50].

**Best-case Versus Worst-case Performance Trade-off.** When our lower bound holds, an SMR algorithm can take steps to ameliorate the impact which stragglers have on performance in the worst-case (i.e., when quorums switch). Coping with stragglers, however, does not come for free. The best-case performance can suffer if this algorithm expends resources (e.g., additional messages) to assist stragglers. Concretely, these resources could have been used to sustain a higher best-case throughput. When a straggler becomes necessary in an active quorum, however, this algorithm will suffer a smaller penalty for switching quorums and hence the performance in the worst-case will be more predictable.

This is the trade-off between best- and worst-case performance, which can inform the design of SMR algorithms. Most of the current well-known SMR protocols aim to achieve superior best-case throughput by sacrificing worst-case performance. This is done by reducing the replication factor, also known as a *thrifty* optimization [40]. In this optimization, the SMR system uses only  $F + 1$  instead of  $2F + 1$  replicas – thereby stragglers are non-existent – so as to reduce the amount of transmitted messages and hence improve throughput or other metrics [3, 38, 40]. In the worst-case, however, when a fault occurs, this optimization requires the SMR system to either reconfigure or provision an additional replica on the spot [37, 38], impairing performance.

Multi-Paxos proposes a mode of operation that can strike a good balance between best- and worst-case performance [32]. Namely, replicas in this algorithm can have gaps in their logs. When gaps are allowed, a replica can participate in the agreement for some command on log position  $k$  even if this replica does not have earlier commands, i.e., commands in log positions  $l$  with  $l < k$ . As long as the leader has the full log, the system can progress. Even when quorums switch, stragglers can participate without recovery. If the leader fails, however, the protocol halts [52, 11] because no replica has the full log, and execution can only resume after some replica builds the full log by coordinating with the others. It would be interesting in future work to experiment with an implementation that allows gaps, but LibPaxos does not follow this approach [3], and we are not aware of any such implementation.

---

<sup>8</sup> We do not argue that we can guarantee bounded response times in a general setting, only in the model we consider in Section 2.

It is interesting to note that there is not much work on optimizing SMR performance for the worst-case, e.g., by expediting recovery [11], and this is a good avenue for future research, perhaps with applicability in performance-sensitive applications. We believe SMR algorithms are possible where replicas balance among themselves the burden of keeping each other up to date collaboratively, e.g., as attempted in [7]. This would minimize the amount of missing state overall (and at any single replica), so as to be prepared for the worst-case, while minimizing the impact on the best-case performance.

## 6 Concluding Remarks

We examined the relation between consensus and State Machine Replication (SMR) in terms of their complexity. We proved the surprising result that SMR is more expensive than a repetition of consensus instances. Concretely, we showed that in a synchronous system where a single instance of consensus always terminates in a constant number of rounds, completing one SMR command can potentially require a non-constant number of rounds. Such a scenario can occur if some processes are stragglers in the SMR algorithm, but later the stragglers become active and are necessary to complete a command. We showed that such a scenario can occur if even one process is a straggler at a time.

Our result – that an SMR algorithm cannot guarantee a constant response time, even if otherwise the system behaves synchronously – brought into focus a trade-off in SMR. In a nutshell, this is the trade-off between the best-case performance and the worst-case performance of an SMR algorithm. On the one hand, such an algorithm can optimize for the worst-case performance. In this case, the algorithm can dedicate resources (e.g., by provisioning additional processes or assisting stragglers) to preserve its performance even when faults manifest, translating into lower tail latencies; there are certain classes of SMR-based applications where latencies and their variability are very important [5, 16, 17]. On the other hand, an SMR algorithm can optimize for best-case performance, i.e., during fault-free periods, so that the algorithm advances despite stragglers being left arbitrarily behind [26, 40]. This strategy means that the algorithm can achieve superior throughput, but its performance will be more sensible to faults.

Additionally, we supported our formal proof with experimental results using two well-known SMR implementations (a Multi-Paxos and a Raft implementation). Our experiments highlighted the difference in cost between a single consensus instance and an SMR command. To the best of our knowledge, we are the first to formally – as well as empirically – investigate the performance-cost difference between consensus and SMR.

---

## References

- 1 Amazon EC2. <http://aws.amazon.com/ec2/>. [Online; accessed 9-May-2018].
- 2 etcd. <https://github.com/coreos/etcd>. [Online; accessed 9-May-2018].
- 3 LibPaxos3. <https://bitbucket.org/sciascid/libpaxos>. [Online; accessed 9-May-2018].
- 4 Karolos Antoniadis, Rachid Guerraoui, Dahlia Malkhi, and Dragos-Adrian Seredinschi. State Machine Replication is More Expensive Than Consensus. Technical Report 256238, EPFL, 2018. URL: <https://infoscience.epfl.ch/record/256238>.
- 5 B. Arun, S. Peluso, R. Palmieri, G. Losa, and B. Ravindran. Speeding up Consensus by Chasing Fast Decisions. In *DSN*, 2017.
- 6 Peter Bailis and Kyle Kingsbury. The network is reliable. *ACM Queue*, 12(7):20, 2014.
- 7 Alysson Bessani, Marcel Santos, João Felix, Nuno Neves, and Miguel Correia. On the efficiency of durable state machine replication. In *ATC*, 2013.



- 8 Alysson Bessani, João Sousa, and Eduardo EP Alchieri. State machine replication for the masses with bft-smart. In *DSN*, 2014.
- 9 Martin Biely, Peter Robinson, Ulrich Schmid, Manfred Schwarz, and Kyrill Winkler. Gracefully degrading consensus and k-set agreement in directed dynamic networks. *Theoretical Computer Science*, 726:41–77, 2018.
- 10 Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. Springer, 2011.
- 11 Lásaro Jonas Camargos, Rodrigo Malta Schmidt, and Fernando Pedone. Multicoordinated agreement protocols for higher availability. In *Network Computing and Applications*, 2008.
- 12 Daniel Cason, Parisa J Marandi, Luiz E Buzato, and Fernando Pedone. Chasing the tail of atomic broadcast protocols. In *SRDS*, 2015.
- 13 Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: An engineering perspective. In *PODC*, 2007.
- 14 Tushar D. Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2):225–267, 1996.
- 15 Bernadette Charron-Bost and André Schiper. The heard-of model: computing in distributed systems with benign faults. *Distributed Computing*, 22(1):49–71, Apr 2009.
- 16 James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google’s globally distributed database. *ACM TOCS*, 31(3):8:1–8:22, 2013.
- 17 Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.
- 18 Tzilla Elrad and Nissim Francez. Decomposition of distributed programs into communication-closed layers. *Science of Computer Programming*, 2(3):155–173, 1982.
- 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 Eli Gafni. Round-by-round fault detectors (extended abstract): Unifying synchrony and asynchrony. In *PODC*, 1998.
- 21 Álvaro García-Pérez, Alexey Gotsman, Yuri Meshman, and Ilya Sergey. Paxos consensus, deconstructed and abstracted (extended version). *CoRR*, abs/1802.05969, 2018.
- 22 Chryssis Georgiou, Seth Gilbert, Rachid Guerraoui, and Dariusz R. Kowalski. On the complexity of asynchronous gossip. In *PODC*, 2008.
- 23 Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *SOSP*, 2003.
- 24 Rachid Guerraoui, Matej Pavlovic, and Dragos-Adrian Seredinschi. Incremental consistency guarantees for replicated objects. In *OSDI*, 2016.
- 25 Heidi Howard and Jon Crowcroft. Coracle: Evaluating Consensus at the Internet Edge. In *SIGCOMM*, 2015.
- 26 Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. Flexible Paxos: Quorum Intersection Revisited. In *OPODIS*, 2016.
- 27 Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX ATC*, 2010.
- 28 Idit Keidar and Sergio Rajsbaum. On the cost of fault-tolerant consensus when there are no faults: Preliminary version. *SIGACT News*, 32(2):45–63, 2001.
- 29 M. Kleppmann. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O’Reilly Media, 2017.

- 30 Tim Kraska, Gene Pang, Michael J Franklin, Samuel Madden, and Alan Fekete. MDCC: Multi-data center consistency. In *EuroSys*, 2013.
- 31 Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, 1978.
- 32 Leslie Lamport. The part-time parliament. *ACM TOCS*, 16(2):133–169, 1998.
- 33 Leslie Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.
- 34 Leslie Lamport. Lower bounds for asynchronous consensus. In *Future Directions in Distributed Computing*, pages 22–23. Springer, 2003.
- 35 Leslie Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.
- 36 Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Stoppable Paxos. *TechReport, Microsoft Research*, 2008.
- 37 Leslie Lamport and Mike Massa. Cheap paxos. In *DSN*, 2004.
- 38 Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shriru, and Michael Williams. Replication in the Harp File System. In *SOSP*, 1991.
- 39 John MacCormick, Nick Murphy, Marc Najork, Chandu Thekkath, and Lidong Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *OSDI*, 2004.
- 40 Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is More Consensus in Egalitarian Parliaments. In *SOSP*, 2013.
- 41 Yoram Moses and Sergio Rajsbaum. A layered analysis of consensus. *SIAM Journal on Computing*, 31(4):989–1021, 2002.
- 42 A. Mostefaoui and M. Raynal. Low cost consensus-based atomic broadcast. In *Proceedings. 2000 Pacific Rim International Symposium on Dependable Computing*, 2000.
- 43 Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *ATC*, 2014.
- 44 Nicola Santoro and Peter Widmayer. Time is not a healer. In *STACS*, 1989.
- 45 Nicola Santoro and Peter Widmayer. Agreement in synchronous networks with ubiquitous faults. *Theoretical Computer Science*, 384(2):232–249, 2007.
- 46 Nuno Santos and André Schiper. Tuning paxos for high-throughput with batching and pipelining. In *International Conference on Distributed Computing and Networking*, pages 153–167. Springer, 2012.
- 47 Ulrich Schmid, Bettina Weiss, and Idit Keidar. Impossibility results and lower bounds for consensus under link failures. *SIAM Journal on Computing*, 38(5):1912–1951, 2009.
- 48 Ulrich Schmid, Bettina Weiss, and John Rushby. Formally verified byzantine agreement in presence of link faults. In *ICDCS*, 2002.
- 49 Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.
- 50 Emil Sit, Andreas Haeberlen, Frank Dabek, Byung-Gon Chun, Hakim Weatherspoon, Robert Morris, M Frans Kaashoek, and John Kubiatowicz. Proactive Replication for Data Durability. In *IPTPS*, 2006.
- 51 Robert H Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM TODS*, 4(2):180–209, 1979.
- 52 Gustavo M. D. Vieira, Islene C. Garcia, and Luiz Eduardo Buzato. Seamless paxos coordinators. *CoRR*, abs/1710.07845, 2017. [arXiv:1710.07845](https://arxiv.org/abs/1710.07845).
- 53 Matt Welsh, David Culler, and Eric Brewer. Seda: An architecture for well-conditioned, scalable internet services. In *SOSP*, 2001.
- 54 Benjamin Wester, James A Cowling, Edmund B Nightingale, Peter M Chen, Jason Flinn, and Barbara Liskov. Tolerating latency in replicated state machines through client speculation. In *NSDI*, 2009.



# Allocate-On-Use Space Complexity of Shared-Memory Algorithms

James Aspnes<sup>1</sup>

Yale University Department of Computer Science, New Haven, CT, USA

Bernhard Haeupler

Carnegie Mellon School of Computer Science, Pittsburgh, PA, USA

Alexander Tong<sup>2</sup>

Yale University Department of Computer Science, New Haven, CT, USA

Philipp Woelfel

University of Calgary, Department of Computer Science, Calgary, AB, Canada

---

## Abstract

---

Many fundamental problems in shared-memory distributed computing, including mutual exclusion [8], consensus [18], and implementations of many sequential objects [14], are known to require linear space in the worst case. However, these lower bounds all work by constructing particular executions for any given algorithm that may be both very long and very improbable. The significance of these bounds is justified by an assumption that any space that is used in some execution must be allocated for all executions. This assumption is not consistent with the storage allocation mechanisms of actual practical systems.

We consider the consequences of adopting a per-execution approach to space complexity, where an object only counts toward the space complexity of an execution if it is used in that execution. This allows us to show that many known randomized algorithms for fundamental problems in shared-memory distributed computing have expected space complexity much lower than the worst-case lower bounds, and that many algorithms that are adaptive in time complexity can also be made adaptive in space complexity.

For the specific problem of mutual exclusion, we develop a new algorithm that illustrates an apparent trade-off between low expected space complexity and low expected RMR complexity. Whether this trade-off is necessary is an open problem.

For some applications, it may be helpful to pay only for objects that are updated, as opposed to those that are merely read. We give a data structure that requires no space to represent objects that are not updated at the cost of a small overhead on those that are.

**2012 ACM Subject Classification** Theory of computation → Distributed computing models, Computing methodologies → Shared memory algorithms, Software and its engineering → Mutual exclusion

**Keywords and phrases** Space complexity, memory allocation, mutual exclusion

**Digital Object Identifier** 10.4230/LIPIcs.DISC.2018.8

---

<sup>1</sup> Supported in part by NSF grants CCF-1637385 and CCF-1650596.

<sup>2</sup> Supported by NSF grant CCF-1650596.



## 1 Introduction

The space complexity of shared-memory distributed data structures and protocols, measured in terms of the number of distinct objects needed to implement them, is typically linear in the number of processes. On the upper bound side, this follows from the ability to implement most algorithms using a single output register for each process (which might hold very large values). On the lower bound side, linear lower bounds have long been known for fundamental problems like mutual exclusion [8] and implementing many common shared-memory objects [14]; and have been shown more recently for consensus [10, 18].

Linear bounds are not terrible, but they do limit the scalability of concurrent data structures for very large numbers of processes. The structure of the known lower bound proofs suggest that executions requiring linear space may be rare: known bounds on mutual exclusion and perturbable objects may construct exponentially long executions, while the bounds on consensus depend on constructing very specific executions that are avoidable if the processes can use randomization.

We propose considering per-execution bounds on the space complexity of a shared-memory protocol, where the protocol is charged only for those objects that it actually uses during the execution. This allows for expected space-complexity bounds and high-probability space complexity bounds, which would be meaningless if an algorithm is charged for all objects, used or not.

We define this measure formally in Section 2. We believe that our measure gives a more refined description of the practical space complexity of many shared-memory algorithms, and observe in our analysis of previously known algorithms in Section 3 that our measure formalizes notions of allocate-on-use space complexity that have already been informally considered by other researchers.

Charging only for objects used has strong practical justifications:

- In a system that provides storage allocation as part of its memory management, it may be that unused registers or pages have no actual cost to the system. Alternatively, it may be possible to construct high-level storage allocation mechanisms even in an adversarial setting that allow multiple protocols with dynamic space needs to share a large fixed block of memory.
- Given an algorithm with low expected space complexity – or better yet, with high-probability guarantees of low space complexity – we may be able to run it in fixed space at the cost of accepting a small chance that the algorithm fails by attempting to exceed its space bound. Thus randomized space complexity can be a tool for trading off space for probability of failure.

To show the applicability of our measure, we also include several positive results: In Section 3, we demonstrate that many known algorithms for fundamental shared-memory algorithms either have, or can be made to have with small tweaks, low space complexity in most executions. In Section 4, we describe a new randomized algorithm for mutual exclusion that achieves  $O(\log n)$  space complexity with high probability for polynomially many invocations.

In Section 5, we consider an alternative measure that charges only objects that are updated and not those that are only read. We show that this is equivalent up to logarithmic factors to the allocate-on-use measure.

Finally, we discuss open problems in Section 6.

## 1.1 Model

We consider a standard asynchronous shared-memory model in which a collection of  $n$  **processes** communicate by performing **operations** on shared-memory **objects**. Concurrency is modeled by interleaving operations; each operation takes place atomically and is a **step** of the process carrying it out. For convenience, we assume that the identity of an operation includes the identity both of the process carrying out the operation and of the object to which it is applied. An **execution** is a sequence of operations.

Scheduling is controlled by an adversary. If the processes are randomized, then each process has access to local coins that may or may not be visible to the adversary. An **adaptive adversary** may observe the internal states of the processes, including the results of local coin-flips, but cannot predict the outcome of future coin-flips. An **oblivious adversary** simply provides in advance a list of which process carries out an operation at each step, without being able to react to the choices made by the processes. We may also consider adversaries with powers intermediate between these two extremes. In each case, the interaction between the processes and the adversary gives a probability distribution over executions. But rather than make this probability distribution explicit, we will usually just generalize the notion of an execution to a random variable  $H$  that maps to each possible execution with a probability determined by the distribution.

### 1.1.1 Time complexity

The **individual step complexity** of an algorithm executed by a single process is the number of steps carried out by that process before it finishes. The **total step complexity** is the total number of steps over all processes. For mutual exclusion algorithms, we may consider the **remote memory reference (RMR)** complexity, in which read operations on a register are not counted if (a) the register has not changed since the last read by the same process (in the **distributed shared memory model** or (b) no operation has been applied to the registers since the last read by the same process (in the **cache-coherent model**).

## 2 Space complexity

The traditional measure of space complexity is **worst-case space complexity**, the number of distinct objects used by the protocol across all executions. We consider instead the space complexity of individual executions.

► **Definition 1.** The **space complexity** of an execution  $H$  of a shared-memory system is the number of distinct objects  $O$  such that  $H$  includes at least one operation on  $O$ .

For randomized algorithms, this allows us to talk about **expected space complexity** – the expected value of the space complexity of the execution resulting from the random choices of the processes – and **high-probability** bounds on space complexity – where the bound applies to the space complexity of all but a polynomially-small fraction of executions.

For adaptive algorithms, this allows the space complexity of an execution to depend on the number of participating processes.

## 3 Examples of allocate-on-use space complexity

In this section, we analyze the space complexity of several recent algorithms from the literature. These include the current best known algorithms (in terms of expected individual step complexity) for implementing test-and-set [11] and consensus [3] from atomic registers,

assuming an oblivious adversary. We also include some related algorithms to demonstrate how charging only for objects used can illuminate trade-offs that might not otherwise be visible.

► **Theorem 2.**

1. Let  $H$  be an execution of the RatRace algorithm for adaptive test-and-set of Alistarh et al. [2], with  $k$  participants. Then the space complexity of  $H$  is  $\Theta(k)$  with high probability.
2. Let  $H$  be an execution of the randomized test-and-set of Alistarh and Aspnes [1]. Then the space complexity of  $H$  is  $\Theta(\log \log n)$  with high probability.
3. Let  $H$  be an execution of the randomized test-and-set of Giakkoupis and Woelfel [11]. Then the space complexity of  $H$  is  $\Theta(\log n)$  with high probability.
4. Let  $H$  be an execution of the  $\Theta(\log \log n)$  expected time  $m$ -valued randomized consensus protocol of Aspnes [3]. Then the space complexity of  $H$  is  $\Theta\left(\log \log n \cdot \frac{\log m}{\log \log m}\right)$  in expectation.

**Proof.**

1. The RatRace algorithm works by having each processes randomly select a path through a binary tree until it manages to acquire a node using a splitter [16], then fight its way back to the root by winning a 3-process consensus object at each node. Both the splitter and consensus object associated with each node require a constant number of registers to implement, so the space complexity is determined by the number of nodes in the subtree traversed by processes. An analysis of a similar algorithm for adaptive collect [6] is used to show that the size of the tree is  $\Theta(k)$  with high probability, so  $\Theta(k)$  of the  $O(n^3)$  registers pre-allocated in the RatRace algorithm are used. This implies that the algorithm uses  $\Theta(k)$  space with high probability.

Because our model does not require pre-allocating a specific bounded address space, RatRace can be modified to use an unbounded number of possible processes and still get the claimed bounds as a function of  $k$ .

2. The Alistarh-Aspnes TAS runs the processes through a sequence of  $\Theta(\log \log n)$  **sifter** objects, each implemented using a one-bit atomic register. The authors show that with high probability, a constant number of processes remain at the end of this sequence, which then enter a RatRace TAS object. The sifter array uses  $\Theta(\log \log n)$  space in all executions. From the previous argument, the RatRace object uses  $O(1)$  space with high probability.
3. The Giakkoupis-Woelfel TAS also uses a sequence of sifter objects; these reduce the number of remaining processes to  $O(1)$  in only  $\Theta(\log^* n)$  rounds, but the cost is an increase in the space required for each object to  $O(\log n)$ . However, after the first sifter the number of remaining processes drops to  $O(\log n)$  with high probability, so subsequent sifter objects can be implemented in  $O(\log \log n)$  space. This makes the space required dominated by the initial sifter object, giving the claimed bound.
4. The Aspnes consensus algorithm uses a sequence of rounds, where each round uses a structure based on the Alistarh-Aspnes sifter to reduce the number of distinct identities followed by an adopt-commit object to detect agreement. This produces agreement in  $\Theta(\log \log n)$  rounds on average.

Using the adopt-commit of Aspnes and Ellen [4], we get  $\Theta(1)$  space for each round for the sifter plus  $\Theta\left(\frac{\log m}{\log \log m}\right)$  for the adopt-commit object. Multiplying by  $\Theta(\log \log n)$  expected rounds gives the claimed bound. ◀

Curiously, all of the variation in space usage for the test-and-set algorithms analyzed above can be attributed to RatRace, either by itself or as a backup for a faster algorithm for winnowing the processes down to a constant number. Using a worst-case measure of space complexity hides the cost of these winnowing steps behind the polynomial worst-case space complexity of RatRace. Using our measure instead exposes an intriguing trade-off between time and space complexity, where the Alistarh-Aspnes algorithm obtains  $O(\log \log n)$  space complexity at the cost of  $\Theta(\log \log n)$  individual step complexity, while the Giakkoupis-Woelfel algorithm pays  $O(\log n)$  space complexity but achieves a much better  $\Theta(\log^* n)$  individual step complexity. Whether this trade-off is necessary is an open problem.

## 4 Monte Carlo Mutual Exclusion

In this section, we present a Monte Carlo mutual exclusion algorithm, which uses only  $O(\log n)$  registers, and against a weak adaptive adversary satisfies mutual exclusion with high probability for polynomially many passages through the critical section. This can be used directly, or can be combined with Lamport's fast mutual exclusion algorithm [15] to give an algorithm that uses  $O(\log n)$  space initially, then backs off to a traditional  $O(n)$  space algorithm when the Monte Carlo algorithm fails. The combined algorithm thus guarantees mutual exclusion in all executions while using only  $O(\log n)$  space with high probability for polynomially many passages through the critical section.

A mutual exclusion algorithm provides two methods, `lock()` and `unlock()`. Each process repeatedly calls `lock()` followed by `unlock()`. When a process's `lock()` call terminates, it is in the **critical section** (CS). The algorithm satisfies **mutual exclusion**, if for any execution, no processes are in the critical section at the same time. An infinite execution is **fair**, if each process that is in the CS or has a pending `lock()` or `unlock()` call either takes infinitely many steps or enters the remainder section (which happens when it is not in the CS and has no `lock()` or `unlock()` call pending). A mutual exclusion algorithm is **deadlock-free**, if in any infinite fair execution, each `lock()` and `unlock()` call terminates. If it is randomized, and in an infinite fair execution each `lock()` and `unlock()` call terminates with probability 1, then we call it **randomized deadlock-free**.

Burns and Lynch [8] proved that any deterministic deadlock-free mutual exclusion algorithm implemented from registers, requires at least  $n$  of them. For fewer than  $n$  registers, the proof constructs exponentially long executions such that at the end two processes end up in the CS. But there are no mutual exclusion algorithms known that use  $o(n)$  registers and do not fail provided that only polynomially many `lock()` calls are made. Here we present a randomized algorithm that has this property with high probability, i.e., it uses only  $O(\log n)$  registers, and in an execution with polynomially many `lock()` calls mutual exclusion is satisfied with high probability.

Our algorithm works for a weak adaptive adversary, which cannot intervene between a process's coin flip and its next shared step. I.e., it schedules a process based on the entire system state, and then that process flips its next coin, and immediately performs its following shared memory step.

The time efficiency of mutual exclusion algorithms is usually measured in terms of remote memory references (RMR) complexity. Here we consider the standard cache-coherent (CC) model. Each processor keeps local copies of shared variables in its cache; the consistency of copies in different caches is maintained by a coherence protocol. An RMR occurs whenever a process writes a register (which invalidates all valid cache copies of that register), and when a process reads a register of which it has no valid cache copy. The RMR complexity of a mutual

exclusion algorithm is the maximum number of RMRs any `lock()` and `unlock()` method incurs. The best deterministic mutual exclusion algorithms use  $O(n)$  registers and have an RMR complexity of  $O(\log n)$  [17], which is tight [5]. Randomized Las Vegas algorithms can beat the deterministic lower bound (e.g. [7, 13, 12]), but they all use at least a linear or even super-linear number of registers and stronger compare-and-swap primitives.

Our algorithm has an expected *amortized* RMR complexity of  $O(n)$ : In any execution with  $L$  `lock()` calls, the total expected number of RMRs is  $O(n \cdot L)$ .

#### 4.1 The algorithm

Pseudocode for our Monte Carlo mutual exclusion algorithm is given in Figure 1.

The idea of the algorithm is that to reach the critical section, a process must climb a slippery ladder whose rungs are a set of  $\Gamma = O(\log n)$  Boolean registers  $S_0, \dots, S_{\Gamma-1}$ . Each of these registers is initially 0.

To climb the ladder, a process executing a `lock()` call attempts to acquire each rung by flipping a coin. With probability  $1/2$ , it writes a 1 to the register and continues to the next. With probability  $1/2$ , it reads the register instead. If the process reads a 0, it tries again; if a 1, it falls back to the bottom of the ladder. The first process to write a 1 will always rise, preventing deadlock. Roughly half of the remaining processes that reach each rung will fall, leaving only a single process with high probability after  $O(\log n)$  rungs. For processes that fall, the number of steps they take in their ascent has a geometric distribution, so each such process takes  $O(1)$  steps per attempt.

At the bottom of the ladder, a process spins on an auxiliary register  $A$ , that is modified only by processes executing `unlock()` calls. This ensures that the expected amortized RMR complexity of each passage through the critical section is  $O(n)$ , as each call to `unlock()` releases at most  $n$  processes, each of which takes  $O(1)$  steps before spinning on  $A$  again.

To avoid ABAs, register  $A$  stores a sequence number that increases with each write. This means that for infinitely many `lock()` calls, the values stored in  $A$  are unbounded. But if each process calls `lock()` at most polynomially many times (after which no guarantee for the mutual exclusion property can be made anyway), then  $O(\log n)$  bits suffice for  $A$ .

► **Theorem 3.** *There is a randomized exclusion algorithm implemented from  $O(\log n)$  bounded shared registers with expected amortized RMR complexity  $O(n)$ , such that for a polynomial number of `lock()` calls, the algorithm is randomized deadlock-free, and satisfies mutual exclusion with high probability.*

#### 4.2 Proof of Theorem 3

The proof is divided into three parts. Section 4.2.1 shows mutual exclusion holds for polynomially many passages through the critical section with high probability. Section 4.2.2 shows deadlock-freedom. Section 4.2.3 gives the bound on RMR complexity.

##### 4.2.1 Mutual exclusion

We consider a random execution of the algorithm, and let  $C_t$  denote the configuration reached at point  $t$ , and  $L_t$  denote the number of completed `lock()` calls at point  $t$ .

The idea of this part of the proof is that we define a potential function  $\Phi(C_t)$  that becomes exponentially large if more than one process enters the critical section simultaneously. We then show that the expected value of  $\Phi(C_t)$  is proportional to  $L_t$ , and in particular that it is small if few `lock()` calls have finished. This gives a bound on the probability that two processes are in the critical section using Markov's inequality.

Class Lock( $\Gamma$ ) .
<p><b>shared:</b>            Boolean Register <math>S_0, \dots, S_{\Gamma-1}</math>            Register <math>A</math> initially <math>(\perp, 0)</math></p> <p><b>local:</b>            Integers <math>i, seq = 0</math> (<math>seq</math> has global scope)</p>
<p><b>Method lock()</b></p> <pre style="margin: 0; padding-left: 20px;"> 1  i = 0 2  while i &lt; <math>\Gamma</math> do 3    Choose random <math>rnd \in \{R, W\}</math> s.t. <math>Prob(rnd = W) = \frac{1}{2}</math> 4    if <math>rnd = R</math> then 5      if <math>S_i = 1</math> then 6            <math>i = 0</math>; 7      end 8    else 9          <math>S_i.write(1); i = i + 1</math> 10   end 11   if <math>i = 0</math> then 12         <math>a = A</math> 13         if <math>S_0 = 1</math> then 14               while <math>A = a</math> do “nothing” done 15         end 16   end 17 end </pre>
<p><b>Method unlock()</b></p> <pre style="margin: 0; padding-left: 20px;"> 18 i = <math>\Gamma</math> 19 while i &gt; 0 do 20     <math>S_{i-1}.write(0); i := i - 1</math> 21 end 22 <math>A.write(myID, seq + 1); seq = seq + 1</math> </pre>

■ **Figure 1** Monte Carlo Mutual Exclusion.

To denote the value of a local variable of a process  $p$ , we add subscript  $p$  to the variable name. For example,  $i_p$  denotes the value of  $p$ 's local variable  $i$ . To bound the probability of error, we define a potential function. The potential of a process  $p \in \{1, \dots, n\}$  is

$$\alpha(p) = \begin{cases} -1 & \text{if } p \text{ is poised to read in lines 12-14} \\ & \text{and entered this section through line 6} \\ 2^{i_p} - 1 & \text{otherwise.} \end{cases} \quad (4.1)$$

Hence,  $\alpha(p) = -1$  if and only if  $p$  is poised in lines 12-14, and prior to entering that section it read  $S_{i_p} = 1$  in line 5. The potential of register index  $j \in \{0, \dots, \Gamma - 1\}$  is

$$\beta(j) = -S_j \cdot w^j \quad (4.2)$$

Finally, the potential of the system at time  $t$  is

$$\Phi(C_t) = \sum_{p=1}^n \alpha(p) + \sum_{j=0}^{\Gamma-1} \beta(j) + (n-1) \quad (4.3)$$

► **Lemma 4.** *Suppose at some point  $t_1$  process  $p$  reads  $S_{j_1} = 1$  in line 5, and at a point  $t_2 \geq t_1$  it reads  $S_{j_2} = 1$  in line 5. Then at some point  $t' \in [t_1, t_2]$  either the value of  $S_0$  changes from 0 to 1, or the value of  $A$  changes.*

**Proof.** After reading  $S_{j_1} = 1$  at point  $t_1$ , process  $p$  proceeds to execute line 13, and thus it executes lines 12-14 during  $[t_1, t_2]$ . Let  $t' \in [t_1, t_2]$  be the point when it reads  $S_0$  in line 13.

First assume  $S_0 = 1$  at point  $t'$ . Then  $p$  enters the while-loop in line 14, and does not leave the while-loop until  $A$  has changed at least once since  $p$ 's previous read of  $A$  in line 12. Hence, in that case  $A$  changes at some point between  $[t_1, t_2]$ , and the claim is true.

Now assume  $S_0 = 0$  at point  $t'$ . We show that  $S_0$  changes from 0 to 1 at some point in  $[t', t_2]$ , which proves the claim. If  $j_2 = 0$ , then at point  $t_2$  process  $p$  reads  $S_0 = 1$ , so this is obvious. Hence, assume  $j_2 > 0$ . Then before point  $t_2$  process  $p$  must increment its local variable  $i_p$  by at least one, which means it writes 1 to  $S_0$  in line 9. ◀

► **Lemma 5.** *For a random execution that ends at point  $t$  with  $L$  `lock()` calls completed,  $\mathbb{E}[\Phi(C_t)] \leq 2n(L+1)$ .*

**Proof.** Consider the initial configuration  $C_0$  where each process is poised to begin a `lock()` call and all registers are 0. Then for all processes  $p$ ,  $\alpha(p) = 0$ , and for all  $j \in \{0, \dots, \Gamma-1\}$ ,  $\beta(j) = 0$ . Hence,  $\Phi(C_0) = n-1 < n$ . We bound the expected value of  $\Phi(C_t)$  in subsequent steps by case analysis. Whenever the adversary schedules a process  $p$  that has a pending `lock()` or `unlock()` call,  $p$  will do one of the following:

- (1) Set  $S_{i_p} = 0$  in line 20;
- (2) Exit Lines 12–14 having entered from line 5;
- (3) Exit Lines 12–14 having entered from line 6;
- (4) Stay in Lines 12–14;
- (5) Choose  $rnd_p$  at random in line 3 and then immediately either read  $S_{i_p}$  in line 5 or write  $S_{i_p}$  in line 9.

We will show that in cases (1), (2), (4), and (5) the expected change in  $\Phi$  is less than or equal to 0. In case (3)  $\Phi$  increases by 1. However, case (3) can only occur at most twice per process per successful lock call leading to our bound on  $\Phi(C_t)$ .

(1) Suppose  $p$  sets  $S_{i_p-1} = 0$  in line 20. Then  $\alpha(p)$  decreases by  $2^{i_p-1}$ . If  $S_{i_p-1}$  was 1, then  $\beta_{i_p-1}$  increases by  $S_{i_p-1}$  and  $\Phi$  does not change. If  $S_{i_p-1}$  was 0, then  $\Phi$  decreases.

(2) Next suppose  $p$  reads  $S_0 = 0$  in line 13 or reads some  $A \neq a_p$  in line 14 having entered from line 5 (i.e.,  $\alpha(p) = 0$ ). Then  $p$  becomes poised to execute line 3 and  $\Phi$  does not change.

(3) Next suppose  $p$  reads  $S_0 = 0$  in line 13 or  $p$  reads some  $A \neq a_p$  in line 14 having entered from line 6  $p = R$  (i.e., when  $\alpha(p) = -1$ ). Then no register gets written,  $p$ 's local variable  $i$  remains at value 0, and  $p$  ceases to be poised to execute a line in the range 12-14, so  $\alpha(p)$  increases from  $-1$  to 0. So  $\Phi$  increases by 1.

(4) Next, suppose  $p$  reads  $S_0 = 1$  in line 13, reads  $A$  in line 12, or reads  $A = a_p$  in line 14. Then no register gets written,  $\alpha(p)$  does not change, and  $p$ 's local variable  $i$  remains at 0, so  $\Phi$  stays the same.

(5) Finally, suppose that when  $p$  gets scheduled it chooses  $rnd_p$  at random, and then it either reads or writes  $S_{i_p}$ , depending on its random choice  $rnd_p$ . First assume  $S_{i_p} = 0$  when  $p$  gets scheduled. If  $rnd_p = R$ , then  $p$  reads  $S_{i_p}$  in line 5, and becomes poised to either



read  $A$  in line 12 (if  $i_p = 0$ ) entering this section from line 5 so  $\alpha(p)$  does not change, or becomes poised to choose  $rnd_p$  at random again in line 3. In either case  $\Phi$  does not change. If  $rnd_p = W$ , then  $p$  proceeds to write 1 to  $S_{i_p}$  in line 9, increments its local variable  $i$  to  $i'_p = i_p + 1$ , and either enters the critical section (if  $i'_p = \Gamma$ ), or becomes poised to make another random choice in line 3. Hence, the value of  $\alpha(p)$  increases by  $2^{i_p}$  (from  $2^{i_p} - 1$  to  $2^{i_p+1} - 1$ ). Since  $S_{i_p}$  changes from 0 to 1, the value of  $\beta(i_p)$  decreases by  $2^{i_p}$ . Therefore, the change of potential  $\Phi$  is 0.

Now suppose  $S_{i_p} = 1$  when  $p$  gets scheduled. If  $rnd_p = R$ , then  $p$  reads  $S_{i_p} = 1$  in line 5, and then immediately sets  $i$  to 0, and becomes poised to read  $A$  in line 12 entering from line 6. Thus,  $p$ 's new potential is  $-1$ . No register gets written, so  $\Phi$  changes by the same amount as  $\alpha(p)$ , which is  $-2^{i_p}$ . If  $rnd_p = W$ , then  $p$  writes 1 to  $S_{i_p}$  in line 5, then increments its local variable  $i$  to  $i'_p = i_p + 1$ , and either enters the critical section if  $i'_p = \Gamma$ , or become poised to make another random choice in line 3. Hence,  $p$ 's potential increases by  $2^{i_p}$ . To summarize, if  $S_{i_p} = 1$ , then with probability  $1/2$  the value of  $\Phi$  increases by  $2^{i_p}$ , and with probability  $1/2$  it decreases by  $2^{i_p}$ . Therefore the expected value of  $\Phi$  does not change in this case.

The only time  $\Phi$  can increase in expectation is in case (3), in which case it increases by 1. We will now show that for any process  $p$ , this case can happen at most twice per critical section. Case (3) can only occur by entering lines 12–14 by reading  $S_{i_p} = 1$  in line 5.

By Lemma 4 we have that if process  $p$  reads  $S_{i_p} = 1$  at  $t_1$  and  $t_2 > t_1$ , then the value of  $S_0$  changes from 0 to 1 or the value of  $A$  changes at some point  $t' \in [t_1, t_2]$ . Let  $U_t$  be the number of completed `unlock()` calls,  $L_t$  be the number of completed `lock()` calls, and  $A_t$  be the value of  $A_{seq}$  at time  $t$ . Since  $A_{seq}$  is only incremented at the end of a completed lock call,  $A_t \leq U_t$ . Since an unlock call is preceded by a successful `lock()` call,  $U_t \leq L_t$ . Hence  $A_t \leq L_t$ . The number of times  $S_0$  changes from 0 to 1 is also bounded by one more than the number of completed `lock()` calls at time  $t$ . Value 0 is written to  $S_0$  only once per `unlock()` call. Thus the number of times  $S_0$  changes from 0 to 1 is at most  $1 + U_t \leq 1 + L_t$ , and at any time  $t$ , the number of times a process  $p$  has taken a step of type (3) is at most  $1 + 2L_t$ . We thus have

$$\mathbb{E}[\Phi(C_t)] \leq \Phi(C_0) + n(1 + 2L) = (n - 1) + n + 2nL < 2n(L + 1). \quad (4.4)$$

◀

► **Lemma 6.** *In any execution, at any point there exists at least one process  $p_{max}$  with local variable  $i_{p_{max}}$  such that  $S_j = 0$  for all  $j \in \{i_{p_{max}}, \dots, \Gamma - 1\}$ .*

**Proof.** Consider any point  $t$  during an execution of the mutual exclusion algorithm. Let  $p_{max}$  be a process such that  $i_{p_{max}}$  is maximal at that point. For the purpose of contradiction assume there is an index  $j \in \{i_{p_{max}}, \dots, \Gamma - 1\}$ , such that  $S_j = 1$  at point  $t$ . Let  $p'$  be the last process that writes 1 to  $S_j$  at some point  $t' \leq t$ . I.e.,

$$S_j = 1 \text{ throughout } (t', t]. \quad (4.5)$$

Moreover, when  $p'$  writes 1 to  $S_j$  in line 9 at point  $t'$ ,  $i_{p'} = j$ , and immediately after writing it increments  $i_{p'}$  to  $j + 1$ . Since  $i_{p'} \leq i_{p_{max}} \leq j$  at point  $t$ , process  $p'$  must at some later point  $t^* \in (t', t)$  decrement  $i_{p'}$  from  $j + 1$  to  $j$ . This can only happen when  $p'$  executes line 20 while  $i_{p'} = j + 1$ . But then  $p'$  also writes 0 to  $S_j$  at  $t^* \in (t', t)$ , which contradicts (4.5). ◀

► **Lemma 7.** *In any reachable configuration  $C$ ,  $\Phi(C)$  is non-negative.*

**Proof.** By Lemma 6 there exists a process  $p_{max}$  such that  $S_j = 0$  for all  $j \in \{i_{p_{max}}, \dots, \Gamma-1\}$ . Then

$$\alpha(p_{max}) = 2^{i_{p_{max}}} - 1 = \sum_{j=0}^{i_{p_{max}}-1} 2^j \geq \sum_{j=0}^{\Gamma-1} S_j \cdot 2^j = - \sum_{j=0}^{\Gamma-1} \beta(j).$$

Since  $\alpha(p) \geq -1$  for each other process  $p$ ,

$$\Phi(C) = n - 1 + \sum_p \alpha(p) + \sum_{j=0}^{\Gamma-1} \beta(j) \geq n - 1 + \sum_{p \neq p_{max}} \alpha(p) \geq n - 1 + \sum_{p \neq p_{max}} -1 = 0. \quad \blacktriangleleft$$

► **Lemma 8.** *If  $C$  is a configuration in which at least two processes are in the critical section,  $\Phi(C) \geq 2^\Gamma$ .*

**Proof.** Suppose that in  $C$ , distinct processes  $p_1$  and  $p_2$  are in the critical section. Then  $\alpha(p_1) = \alpha(p_2) = 2^\Gamma - 1$ . Since  $\alpha(p) \geq -1$  for each other process, and  $\beta(j) \geq -2^j$

$$\Phi(C) \geq (2(2^\Gamma - 1) + (n - 2) \cdot (-1)) + \left( \sum_{j=0}^{\Gamma-1} -2^j \right) + (n - 1) = 2^\Gamma \quad (4.6)$$

◀

► **Lemma 9.** *For  $\Gamma = c \log n$ , the probability that mutual exclusion is violated at any point before  $L \text{ lock}()$  calls finish is  $O(L^2 \cdot n^{-c+1})$ .*

**Proof.** Let  $t_j$  for  $j \in \{2, \dots, L\}$  be the point when the  $j$ -th  $\text{lock}()$  call completes. By Lemma 5,  $E[\Phi(C_{t_j})] = O(n \cdot j)$ , so by Lemmas 7, 8 and Markov's inequality,

$$\Pr [C_{t_j} \in \mathcal{C}_{fail}] \leq \Pr [\Phi(C_{t_j}) \geq 2^\Gamma] = O\left(\frac{n \cdot j}{2^\Gamma}\right).$$

Mutual exclusion is violated before  $L \text{ lock}()$  calls finish if and only if it is violated after  $\ell \text{ lock}()$  calls finish for some  $\ell \in \{2, \dots, L-1\}$ . The probability of that event is

$$\begin{aligned} \Pr [\exists j \in \{2, \dots, L-1\} : C_{t_j} \in \mathcal{C}_{fail}] &\leq \Pr [\exists j \in \{2, \dots, L-1\} : \Phi(C_{t_j}) \geq 2^\Gamma] \\ &\leq \sum_{j=2}^{L-1} \Pr [\Phi(C_{t_j}) \geq 2^\Gamma] = O\left(\sum_{j=2}^{L-1} \frac{n(j+1)}{2^\Gamma}\right) = O\left(\frac{n \cdot L^2}{n^c}\right) = O\left(\frac{L^2}{n^{c-1}}\right). \end{aligned}$$

◀

## 4.2.2 Deadlock-freedom

► **Lemma 10.** *The algorithm is randomized deadlock-free.*

**Proof.** Consider any point  $t$  in an infinite fair execution, in which at least one process has a pending  $\text{lock}()$  call. We will show that some process enters the critical section after point  $t$  with probability 1.

Suppose no process enters the critical section in  $[t, \infty)$ . Since  $\text{unlock}()$  is wait-free, there is a point  $t_1 \geq t$  such that after  $t_1$  there are no more pending  $\text{unlock}()$  calls. Hence, throughout  $[t_1, \infty)$  no process writes 0 to any register  $S_j$ ,  $j \in \{0, \dots, \Gamma-1\}$ . In other words, only value 1 may get written to any register  $S_j$  after point  $t_1$ . Since there are only a finite

number of registers  $S_j$ , there is a point  $t_2$  such that no register  $S_j$ ,  $j \in \{0, \dots, \Gamma - 1\}$ , changes value after  $t_2$ . By Lemma 6 there is a process  $p_{max}$  such that at point  $t_2$  we have  $S_j = 0$  for all  $j \in \{i_{p_{max}}, \dots, \Gamma - 1\}$ . Let  $i^*$  be the value of  $i_{p_{max}}$  at point  $t_2$ . Thus,

$$S_{i^*} = \dots = S_{\Gamma-1} = 0 \text{ throughout } [t_2, \infty). \quad (4.7)$$

If  $i^* > 0$ , then at  $t_2$  process  $p_{max}$  is not poised to execute a shared memory operation in lines 12-14 (because  $i_{p_{max}} = i^*$  at that point). Hence,  $p_{max}$  is either poised to read in line 5 or to write in line 9. The latter is not possible, as  $p_{max}$  would eventually write 1 to  $S_{i^*}$ , contradicting (4.7). If  $p_{max}$  reads in line 5, then it reads 0 from  $S_{i_{p_{max}}}$ , where  $i_{p_{max}} = i^* > 0$ , and so it will begin another iteration of the while-loop with  $i_{p_{max}} = i^*$ . Repeating the argument,  $p_{max}$  will execute an infinite number of iterations of the outer while-loop, each time choosing at random  $rnd = R$ , and then reading  $S_{i^*}$  in line 5. This event has probability 0.

Hence, consider the case  $i^* = 0$ . First assume that at some point after  $t_2$  some process  $p$  is not poised to execute line 14. Then due to (4.7) the if-condition in line 13 remains false for  $p$  throughout  $[t_3, \infty)$ , so  $p$  executes an infinite number of iterations of the outer while-loop. With probability 1 process  $p$  will eventually in some iteration choose  $rnd = W$  in line 3 and then write 1 to some register  $S_j$ ,  $j \in \{0, \dots, \Gamma - 1\}$ . This contradicts (4.7) since we assumed  $i^* = 0$ .

Thus, throughout  $[t_2, \infty)$  all processes with pending `lock()` calls are stuck in the inner while-loop in line 14. Consider any process  $q$  stuck in the while-loop, and let  $T$  be the point when it read  $A$  for last time prior to becoming stuck. Let  $a^*$  be the value of  $A$  at point  $T$ . Register  $A$  gets only written in line 22, and due to the increasing sequence number, the same value never gets written twice. Hence, since  $q$  is stuck in line 14, it reads  $A = a^*$  infinitely many times, and thus

$$\text{no process writes } A \text{ throughout } [T, \infty). \quad (4.8)$$

But at some point  $T_1 > T$  and before  $q$  becomes stuck in the while-loop, it reads  $S_0 = 1$  in line 13. By (4.7), after  $T_1$  some process writes 0 to  $S_0$ , and then it will eventually write to  $A$ . This contradicts (4.8). ◀

### 4.2.3 RMR Bound

► **Lemma 11.** *In an execution with  $L$  invoked `lock()` calls, the expected total number of RMRs is  $O((n + \Gamma)L)$ .*

The remainder of this section is devoted to the proof of this lemma.

Let  $X_{p,\ell}$  denote the number of RMRs a process  $p$  incurs in line  $\ell$ , where  $\ell$  is one of 5, 9, 12, 13, 14, 20, and 22. These are the only lines where a process executes shared memory operations, so the total number of RMRs is obtained by summing over all  $X_{p,\ell}$ .

We now consider a random execution, and condition on the event that the random execution contains  $L$  `lock()` calls.

► **Lemma 12.** *For each  $j \in \{0, \dots, \Gamma - 1\}$ , each process incurs in total at most  $L + 1$  RMRs by reading value 0 from register  $S_j$ .*

**Proof.** Value 0 is written to  $S_j$  (in line 20) only once per `unlock()` call. Only the first read by a process in the execution, or the first read following such a write of value 0 can at the same time return 0 and incur an RMR. Now the claimed bound follows from the fact that there are at most  $L$  `lock()`, and thus at most  $L$  `unlock()` calls. ◀

► **Lemma 13.** *For any process  $p$  we have*

$$X_{p,12} + X_{p,14} \leq L + 1 \text{ and } X_{p,13} \leq 2(L + 1).$$

**Proof.** The value of  $A$  changes at most once per `unlock()` call, and thus at most  $L$  times during execution  $E$ . Hence, process  $p$  incurs at most  $L + 1$  RMRs by reading  $A$ . This proves the claimed upper bound on  $X_{p,12} + X_{p,14}$ .

By Lemma 12, the number of RMRs incurred by  $p$ 's reads of value 0 in line 13 is at most  $L + 1$ . If process  $p$  reads 1 from  $S_0$  in that line, then, due to the while-loop in line 14, it does not read  $S_0$  again until  $A$  changed at least once since  $p$ 's preceding read of  $A$  in line 12. In particular, for each read of value 1 from  $S_0$ , there is a distinct RMR incurred by  $p$  when reading  $A$  in line 14. Hence,  $X_{p,13} \leq L + 1 + X_{p,14} \leq 2(L + 1)$ . ◀

► **Lemma 14.**  $E[\sum_p X_{p,9}] = O((n + \Gamma)L)$

**Proof.** For  $b \in \{0, 1\}$  let  $Z_b$  denote the number of times a write in line 9 (by any process) overwrites value  $b$  with value 1. Thus,

$$\sum_p X_{p,9} = Z_0 + Z_1. \quad (4.9)$$

Since each register  $S_j$  is reset to 0 only once per `unlock()` call, it can change from 0 to 1 at most  $L + 1$  times. Accounting for  $\Gamma$  registers, we obtain

$$Z_0 \leq \Gamma(L + 1). \quad (4.10)$$

Now suppose  $S_j = 1$  when process  $p$  makes a random choice in line 3. With probability  $1 - 1/w$  process  $p$  decides to read, and if it does so, it reads  $S_j = 1$ . Hence,  $p$  overwrites in 9 a register that has value 1 in expectation at most  $1/(1 - 1/w) - 1 = 1/(w - 1)$  times before  $p$  reads a register with value 1 in line 5. By Lemma 4 between any two such reads, either  $S_0$  changes from 0 to 1 or  $A$  changes, and each of these events happens at most once per `unlock()` call. Thus, the expected number of times process  $p$  writes to a register  $S_j$  that has value 1 is at most  $1 + 1/(w - 1) \cdot L$ . Summing over all processes we obtain  $E[Z_1] = O(n \cdot L)$  (recall that  $Z_0 = Z_1 = 0$  if  $L = 0$ ). Now the claim follows from (4.9) and (4.10). ◀

► **Lemma 15.** *For any process  $p$  we have*

$$E[X_{p,5}] = O((n + \Gamma)L).$$

**Proof.** Let  $Y_p$  denote the number of times process  $p$  reads a value of 1 in line 5. By Lemma 4, between any two such consecutive reads, either the value of  $A$  changes, or  $S_0$  changes from 0 to 1. Since  $S_0$  can change from 1 to 0 at most  $L$  times (once for each `unlock()` call), it can change from 0 to 1 at most  $L + 1$  times. The value of  $A$  can also change at most once for each `unlock()` call, and thus at most  $L$  times. Hence,  $Y_p \leq 2L + 2$ .

By Lemma 12 process  $p$  incurs at most  $L + 1$  RMRs by reading value 0 from  $S_0$  in line 5. Now suppose  $j > 0$ . Then  $p$  reads  $S_j$  only after writing 1 to  $S_{j-1}$  in line 9, which contributes to  $X_{p,5}$ . Because  $p$  chooses to write  $S_j$  (instead of reading it) with probability  $1/w$ , the expected number of times  $p$  can read  $S_j$  in consecutive iterations of the while-loop (and thus before changing  $i_p$ ) is at most  $w - 1$ . Hence, for all  $x$ ,

$$E[X_{p,5} | X_{p,9} = x] \leq E[Y_p | X_{p,9} = x] + L + 1 + x(w - 1) \leq 3(L + 1) + x(w - 1)$$

Summing this conditional expectation weighted with  $\Pr[X_{p,9} = x]$  over all values of  $x$ , yields

$$E[X_{p,5}] \leq 3L + 3 + E[X_{p,9}] \cdot (w - 1).$$

Now the claim follows from Lemma 14. ◀

Lemma 11 now follows from Lemmas 12, 13, and 14:

**Proof of Lemma 11.** If  $L = 0$ , then no process calls `lock()` or `unlock()`, so the lemma is trivially true. Hence, we assume w.l.o.g. that  $L \geq 1$ . Since there are at most  $L$  `unlock()` calls in total, we have

$$\sum_p X_{p,20} \leq \Gamma \cdot L \text{ and } \sum_p X_{p,22} \leq L. \quad \blacktriangleleft$$

### 4.3 When the algorithm fails

Because the algorithm is randomized, there is a nonzero chance that it violates mutual exclusion even in short executions. We can guard against this using Lamport's fast mutual exclusion algorithm [15], which is now often abstracted in the form of a **splitter** object [16]. Lamport's fast mutual exclusion algorithm uses  $O(1)$  space and  $O(1)$  time to either allow a process into the critical section or deny it entry, and works as long as at most one process at a time invokes it. Because our algorithm guarantees mutual exclusion for polynomially many critical sections with high probability, in most executions we will not see multiple processes attempting to access the Lamport mutex, and so each process will successfully acquire this mutex. In the even that a process does not acquire the Lamport mutex, then our algorithm has failed; the process can then unlock the randomized algorithm and move over to a backup algorithm to attempt to acquire a mutex there. A 2-process mutex algorithm (using  $O(1)$  space and  $O(1)$  time) can then be used to choose between processes leaving the Lamport mutex and the backup mutex.

Because the combined mutex never uses more than  $O(n)$  objects, the high-probability  $O(\log n)$  space bound also gives a bound on expected space. The full result is:

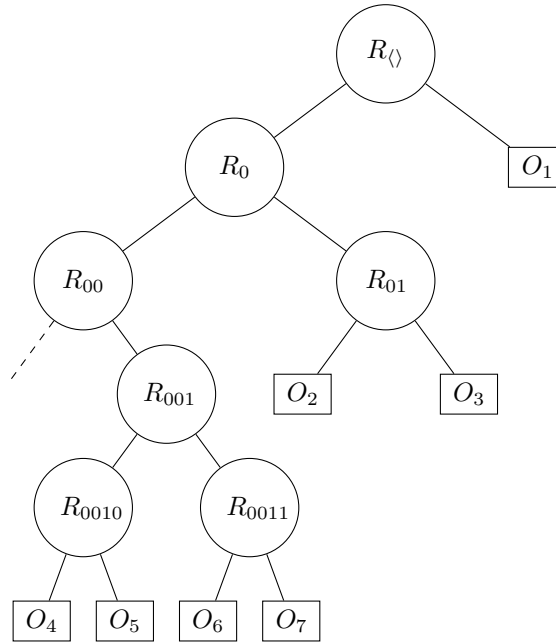
► **Corollary 16.** *There is a randomized mutual exclusion algorithm with expected amortized RMR complexity  $O(n)$ , such that the algorithm is randomized deadlock-free; satisfies mutual exclusion in all executions; uses at most  $O(n)$  objects in all executions; and, for a polynomial number of `lock()` calls, uses  $O(\log n)$  objects in expectation and with high probability.*

## 5 Simulating allocation on update

With a more refined space complexity measure comes the need to develop new tools for minimizing this measure. In this section, we describe a technique for designing protocols where the space complexity is proportional to the number of objects that are **updated** as opposed to all objects that are accessed. We distinguish between **update** operations that can change an object's state and **read** operations that cannot; an object is considered to be updated if an update operation is applied to it, even if its state is not changed by this particular application.

Counting only updates corresponds to an **allocate-on-update** model where merely reading an object costs nothing. We show that this model gives costs equivalent up to a factor logarithmic in size of the address space to the allocate-on-use model of Definition 1.

To obtain this result, we construct a data structure where the objects  $O_1, O_2, \dots$  are the leaves of a binary search tree whose internal nodes are one-bit registers that record if any object in their subtree has been updated. A read operation on some object  $O_i$  starts at the root of the tree and follows the path to  $O_i$  until it sees a 0, indicating that  $O_i$  can be treated as still being in its initial state, or reaches  $O_i$  and applies the operation to it. Conversely, an update operation starts by updating  $O_i$  and then sets all the bits in order along the path from  $O_i$  to the root.



■ **Figure 2** Tree derived from Elias gamma code.

The structure of the tree is based on the well-known correspondence between binary trees and prefix-free codes. Here the left edge leaving each node is labeled with a 0 and the right edge with a 1, the path to each leaf gives a code word, and the path to each internal node gives a proper prefix of a code word. The particular code we will use to construct the tree is the **Elias gamma code** [9]. This encodes each positive integer  $i$  as a sequence of bits, by first expressing  $i$  as its unique binary expansion  $1i_1i_2 \dots i_n$ , and then constructing a codeword  $\gamma(i) = 0^n1i_1i_2 \dots i_n$ . This gives a codeword for each positive integer  $i$  with length  $2\lceil \lg i \rceil + 1 = O(\log i)$ . The first few levels of the resulting tree are depicted in Figure 2.

Pseudocode for the simulation is given in Algorithm 4. Each register is labeled by a codeword prefix. The objects are labeled with their original indices.

► **Lemma 17.** *Algorithm 4 gives a linearizable implementation of  $O_1, O_2, \dots$ , such that in any execution in which update operations start on at most  $m$  objects, and the maximum index of these objects is  $s$ :*

1. *The space complexity is  $O(m \log s)$ .*
2. *The step complexity of an `apply( $\pi$ )` operation where  $\pi$  is an update to  $O_i$  is  $O(\log i)$ .*
3. *The step complexity of an `apply( $\pi$ )` operation where  $\pi$  is a read of  $O_i$  is  $O(\min(\log i, \log s))$ .*

**Proof.** We start by showing linearizability.

Given a concurrent execution  $H$  of Algorithm 4, we will construct an explicit linearization  $S$ . The first step in this construction is to assign a linearization point to each operation  $\pi$  in  $H$ . If  $\pi$  is an update operation on some object  $O_i$ , its linearization point is the first step in  $H$  at which (a)  $\pi$  has been applied to  $O_i$ , and (b) every bit in an ancestor of  $O_i$  is set. If  $\pi$  is a read operation, its linearization point is the step at which either  $\pi$  is applied to  $O_i$ , or the process executing  $\pi$  reads a 0 from an ancestor of  $O_i$ . In the case of an update operation  $\pi$ , the linearization point follows the step in which  $\pi$  is applied to  $O_i$  and precedes the return of  $\pi$  (since  $\pi$  cannot return without setting all ancestors of  $O_i$  to 1). In the case of a read operation  $\pi$ , the linearization point corresponds to an actual step of  $\pi$ . In both cases, the linearization point of  $\pi$  lies within  $\pi$ 's execution interval.

---

**Algorithm 4:** Applying operation  $\pi$  to object  $O_i$ .
 

---

```

procedure apply( $\pi$ )
  Let  $O_i$  be the object on which  $\pi$  is an operation
  Let  $x_1x_2\dots x_k$  be the encoding of  $i$ 
  if  $\pi$  is an update then
     $r \leftarrow \pi(O_i)$ 
    for  $j \leftarrow k - 1 \dots 0$  do
       $R_{x_1\dots x_j} \leftarrow 1$ 
    end
    return  $r$ 
  else
    for  $j \leftarrow 0 \dots k - 1$  do
      if  $R_{x_1\dots x_j} = 0$  then
        return  $\pi$  applied to the initial state of  $O_i$ 
      end
    end
    // Reached only if all nodes on path are 1
    return  $\pi(O_i)$ 
  end
end

```

---

If we declare  $\rho \leq \sigma$  whenever  $\rho$ 's linearization point precedes  $\sigma$ 's, we get a preorder on all operations in  $H$ . Because each operation's linearization point lies within its execution interval, this preorder is consistent with the observed execution order in  $H$ . But it is not necessarily a total order because update operations that are applied to the same object  $O_i$  may be assigned the same linearization point: the first step at which all ancestors of  $O_i$  are 1. Should this occur, we break ties by ordering such updates according to the order in which they were applied to  $O_i$ . We now argue that the resulting total order gives a sequential execution  $S$  on  $O_1, O_2, \dots$ . This requires showing that each operation that returns in  $H$  returns the same value in  $H$  as it would in  $S$ .

Fix some particular  $O_i$ . The operations on  $O_i$  can be divided into three groups:

1. Read operations that observe 0 in an ancestor of  $O_i$ .
2. Update operations that are applied to  $O_i$  before all ancestors of  $O_i$  are 1.
3. Read or update operations that are applied to  $O_i$  after all ancestors of  $O_i$  are 1.

That these groups include all operations follows from the fact that any update operation is applied either before or after all ancestors of  $O_i$  are 1, and any read operation that does not observe a 0 will eventually be applied to  $O_i$  after all ancestors of  $O_i$  are 1.

Now observe that all operations in the first group are assigned linearization points before the step at which all ancestors of  $O_i$  are 1; in the second group, at this step; and in the third group, after this step. So  $S$  restricted to  $O_i$  consists of a group of read operations that return values obtained from the initial state of  $O_i$ , consistent with having no preceding updates; followed by a sequence of updates linearized in the same order that they are applied to  $O_i$  in  $H$ ; followed by a sequence that may contain mixed updates and reads that are again linearized in the same order that they are applied to  $O_i$  in  $H$ . Since the first group of operations contain only read operations, the operations applied to  $O_i$  in  $H$  start with the same initial state as in  $S$ , and since they are the same operations applied in the the same order, they return the same values.

For space complexity, observe that any object accessed in the execution is either (a) an object  $O_i$  that is updated; (b) a register that is the ancestor of an object that is updated; or (c) a register or object all of whose ancestors are set to 1. Since a register is set to 1 only if it is an ancestor of an updated object, and since each such register has at least one child that is either in category (a) or (b), there is an injection from the set of registers and objects in category (c) to their parents in category (b). Category (a) requires  $m$  space; (b) requires  $O(m \log s)$  space; and thus (c) also requires  $O(m \log s)$  space. This gives  $O(m \log s)$  space total.

Time complexity of updates is immediate from the code;  $\text{apply}(\pi)$  traverses  $O(\log i)$  nodes to reach  $O_i$ . For reads,  $\text{apply}(\pi)$  follows a path of length  $O(\log i)$  that stops early if it reaches a node not on the path to an updated object; since any such path to an updated object has length  $O(\log s)$ , this gives the claimed bound. ◀

We believe that these overheads are the best possible using a binary tree structure. However, using a tree with higher arity (equivalent to using a code with a larger alphabet) could produce a lower time complexity overhead at the cost of more wasted space. We do not have a lower bound demonstrating that this particular trade-off is necessary, so the exact complexity of simulating allocate-on-update in the simpler allocate-on-access model remains open.

## 6 Open problems

While we have started a formal approach to analyzing allocate-on-use space complexity for shared-memory distributed algorithms, much remains to be done.

We have demonstrated that it is possible to solve mutual exclusion for a polynomial number of locks with logarithmic space complexity with high probability. Our algorithm pays for its low space complexity with linear RMR complexity. Curiously, it is possible to achieve both  $O(1)$  space and RMR complexity with high probability using very long random delays under the assumption that critical sections are not held for long; this follows from Lamport's fast mutual exclusion algorithm [15] and is essentially a randomized version of the delay-based algorithm of Fischer described by Lamport. However, this algorithm has poor step complexity even in the absence of contention. We conjecture that there exists a randomized algorithm for mutual exclusion that simultaneously achieves  $O(\log n)$  space complexity,  $O(\log n)$  RMR complexity, and  $O(\log n)$  uncontended step complexity, all with high probability assuming polynomially many passages through the critical section.

We have also shown that a system that assumes an allocate-on-update model can be simulated in the stricter allocate-on-access model with a logarithmic increase in the number of objects used. It is not clear whether this overhead is necessary, or whether it could be eliminated with a more sophisticated simulation.

---

### References

- 1 Dan Alistarh and James Aspnes. Sub-logarithmic test-and-set against a weak adversary. In *Distributed Computing: 25th International Symposium, DISC 2011*, volume 6950 of *Lecture Notes in Computer Science*, pages 97–109. Springer-Verlag, 2011.
- 2 Dan Alistarh, Hagit Attiya, Seth Gilbert, Andrei Giurgiu, and Rachid Guerraoui. Fast randomized test-and-set and renaming. In Nancy A. Lynch and Alexander A. Shvartsman, editors, *Distributed Computing, 24th International Symposium, DISC 2010, Cambridge, MA, USA, September 13-15, 2010. Proceedings*, volume 6343 of *Lecture Notes in Computer Science*, pages 94–108. Springer, 2010. doi:10.1007/978-3-642-15763-9\_9.



- 3 James Aspnes. Faster randomized consensus with an oblivious adversary. In *2012 ACM Symposium on Principles of Distributed Computing*, pages 1–8, 2012.
- 4 James Aspnes and Faith Ellen. Tight bounds for adopt-commit objects. *Theory of Computing Systems*, 55(3):451–474, 2014. doi:10.1007/s00224-013-9448-1.
- 5 Hagit Attiya, Danny Hendler, and Philipp Woelfel. Tight RMR lower bounds for mutual exclusion and other problems. In *Proceedings of the 40th Annual ACM Symposium on Theory of Computing, Victoria, British Columbia, Canada, May 17-20, 2008*, pages 217–226, 2008. doi:10.1145/1374376.1374410.
- 6 Hagit Attiya, Fabian Kuhn, C. Greg Plaxton, Mirjam Wattenhofer, and Roger Wattenhofer. Efficient adaptive collect using randomization. *Distributed Computing*, 18(3):179–188, 2006. doi:10.1007/s00446-005-0143-6.
- 7 Michael A. Bender and Seth Gilbert. Mutual exclusion with  $o(\log^2 \log n)$  amortized work. In Rafail Ostrovsky, editor, *IEEE 52nd Annual Symposium on Foundations of Computer Science, FOCS 2011, Palm Springs, CA, USA, October 22-25, 2011*, pages 728–737. IEEE Computer Society, 2011. doi:10.1109/FOCS.2011.84.
- 8 James E. Burns and Nancy A. Lynch. Bounds on shared memory for mutual exclusion. *Inf. Comput.*, 107(2):171–184, 1993. doi:10.1006/inco.1993.1065.
- 9 P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, March 1975. doi:10.1109/TIT.1975.1055349.
- 10 Rati Gelashvili. On the optimal space complexity of consensus for anonymous processes. In Yoram Moses, editor, *Distributed Computing: 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings*, pages 452–466, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. doi:10.1007/978-3-662-48653-5\_30.
- 11 George Giakkoupis and Philipp Woelfel. On the time and space complexity of randomized test-and-set. In Darek Kowalski and Alessandro Panconesi, editors, *ACM Symposium on Principles of Distributed Computing, PODC '12, Funchal, Madeira, Portugal, July 16-18, 2012*, pages 19–28. ACM, 2012. doi:10.1145/2332432.2332436.
- 12 George Giakkoupis and Philipp Woelfel. Randomized abortable mutual exclusion with constant amortized RMR complexity on the CC model. In Elad Michael Schiller and Alexander A. Schwarzmann, editors, *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017, Washington, DC, USA, July 25-27, 2017*, pages 221–229. ACM, 2017. doi:10.1145/3087801.3087837.
- 13 Danny Hendler and Philipp Woelfel. Randomized mutual exclusion with sub-logarithmic rmr-complexity. *Distributed Computing*, 24(1):3–19, 2011. doi:10.1007/s00446-011-0128-6.
- 14 Prasad Jayanti, King Tan, and Sam Toueg. Time and space lower bounds for non-blocking implementations. *SIAM J. Comput.*, 30(2):438–456, 2000. doi:10.1137/S0097539797317299.
- 15 Leslie Lamport. A fast mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, 5(1):1–11, 1987. doi:10.1145/7351.7352.
- 16 Mark Moir and James H. Anderson. Wait-free algorithms for fast, long-lived renaming. *Sci. Comput. Program.*, 25(1):1–39, 1995.
- 17 Jae-Heon Yang and James H. Anderson. A fast, scalable mutual exclusion algorithm. *Distributed Computing*, 9(1):51–60, 1995. doi:10.1007/BF01784242.
- 18 Leqi Zhu. A tight space bound for consensus. In Daniel Wichs and Yishay Mansour, editors, *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*, pages 345–350. ACM, 2016. doi:10.1145/2897518.2897565.



# Almost Global Problems in the LOCAL Model

**Alkida Balliu**

Aalto University, Finland  
alkida.balliu@aalto.fi

**Sebastian Brandt**

ETH Zürich, Switzerland  
brandts@ethz.ch

**Dennis Olivetti**

Aalto University, Finland  
dennis.olivetti@aalto.fi

**Jukka Suomela**

Aalto University, Finland  
jukka.suomela@aalto.fi

---

## Abstract

The landscape of the distributed time complexity is nowadays well-understood for subpolynomial complexities. When we look at deterministic algorithms in the LOCAL model and locally checkable problems (LCLs) in bounded-degree graphs, the following picture emerges:

- There are lots of problems with time complexities  $\Theta(\log^* n)$  or  $\Theta(\log n)$ .
- It is not possible to have a problem with complexity between  $\omega(\log^* n)$  and  $o(\log n)$ .
- In *general graphs*, we can construct LCL problems with infinitely many complexities between  $\omega(\log n)$  and  $n^{o(1)}$ .
- In *trees*, problems with such complexities do not exist.

However, the high end of the complexity spectrum was left open by prior work. In general graphs there are problems with complexities of the form  $\Theta(n^\alpha)$  for any rational  $0 < \alpha \leq 1/2$ , while for trees only complexities of the form  $\Theta(n^{1/k})$  are known. No LCL problem with complexity between  $\omega(\sqrt{n})$  and  $o(n)$  is known, and neither are there results that would show that such problems do not exist. We show that:

- In *general graphs*, we can construct LCL problems with infinitely many complexities between  $\omega(\sqrt{n})$  and  $o(n)$ .
- In *trees*, problems with such complexities do not exist.

Put otherwise, we show that any LCL with a complexity  $o(n)$  can be solved in time  $O(\sqrt{n})$  in trees, while the same is not true in general graphs.

**2012 ACM Subject Classification** Theory of computation  $\rightarrow$  Distributed computing models, Theory of computation  $\rightarrow$  Complexity classes

**Keywords and phrases** Distributed complexity theory, locally checkable labellings, LOCAL model

**Digital Object Identifier** 10.4230/LIPIcs.DISC.2018.9

**Related Version** The full version is available at <https://arxiv.org/pdf/1805.04776.pdf>.

**Funding** This work was supported in part by the Academy of Finland, Grant 285721.



© Alkida Balliu, Sebastian Brandt, Dennis Olivetti, and Jukka Suomela;  
licensed under Creative Commons License CC-BY

32nd International Symposium on Distributed Computing (DISC 2018).

Editors: Ulrich Schmid and Josef Widder; Article No. 9; pp. 9:1–9:16

Leibniz International Proceedings in Informatics



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

## 1 Introduction

Recently, in the study of *distributed graph algorithms*, there has been a lot of interest on *structural complexity theory*: instead of studying the distributed time complexity of specific graph problems, researchers have started to put more focus on the study of *complexity classes* in this context.

**LCL problems.** A particularly fruitful research direction has been the study of distributed time complexity classes of so-called LCL problems (locally checkable labellings). We will define LCLs formally in Section 2.2, but the informal idea is that LCLs are graph problems in which *feasible solutions can be verified by checking all constant-radius neighbourhoods*. Examples of such problems include vertex colouring with  $k$  colours, edge colouring with  $k$  colours, maximal independent sets, maximal matchings, and sinkless orientations.

LCLs play a role similar to the class NP in the centralised complexity theory: these are problems that would be easy to solve with a *nondeterministic* distributed algorithm – guess a solution and verify it in  $O(1)$  rounds – but it is not at all obvious what the distributed time complexity of solving a given LCL problem with *deterministic* distributed algorithms is.

**Distributed structural complexity.** In the classical (centralised, sequential) complexity theory one of the cornerstones is the *time hierarchy theorem* [12]. In essence, it is known that giving more time always makes it possible to solve more problems. Distributed structural complexity is fundamentally different: there are various *gap results* that establish that there are no LCL problems with complexities in a certain range. For example, it is known that there is no LCL problem whose deterministic time complexity on bounded-degree graphs is between  $\omega(\log^* n)$  and  $o(\log n)$  [7].

Such gap results have also direct applications: we can *speed up* algorithms for which the current upper bound falls in one of the gaps. For example, it is known that  $\Delta$ -colouring in bounded-degree graphs can be solved in  $\text{polylog } n$  time [17]. Hence 4-colouring in 2-dimensional grids can be also solved in  $\text{polylog } n$  time. But we also know that in 2-dimensional grids there is a gap in distributed time complexities between  $\omega(\log^* n)$  and  $o(\sqrt{n})$  [5], and therefore we know we can solve 4-colouring in  $O(\log^* n)$  time.

The ultimate goal here is to identify all such gaps in the landscape of distributed time complexity, for each graph class of interest.

**State of the art.** Some of the most interesting open problems at the moment are related to *polynomial complexities in trees*. The key results from prior work are:

- In bounded-degree trees, for each positive integer  $k$  there is an LCL problem with time complexity  $\Theta(n^{1/k})$  [8].
- In bounded-degree graphs, for each rational number  $0 < \alpha \leq 1/2$  there is an LCL problem with time complexity  $\Theta(n^\alpha)$  [1].

However, there is no separation between trees and general graphs in the polynomial region. Furthermore, we do not have any LCL problems with time complexities  $\Theta(n^\alpha)$  for any  $1/2 < \alpha < 1$ .

**Our contributions.** This work resolves both of the above questions. We show that:

- In bounded-degree graphs, for each rational number  $1/2 < \alpha < 1$  there is an LCL problem with time complexity  $\Theta(n^\alpha)$ .
- In bounded-degree trees, there is no LCL problem with time complexity between  $\omega(\sqrt{n})$  and  $o(n)$ .

Hence whenever we have a slightly sublinear algorithm, we can always speed it up to  $O(\sqrt{n})$  in trees, but this is not always possible in general graphs.

**Key techniques.** We use ideas from the classical centralised complexity theory – e.g. Turing machines and regular languages – to prove results in distributed complexity theory.

In the positive result, the key idea is that we can take any *linear bounded automaton*  $M$  (a Turing machine with a bounded tape), and construct an LCL problem  $\Pi_M$  such that the *distributed* time complexity of  $\Pi$  is a function of the *sequential* running time of  $M$ . Prior work [1] used a class of *counter machines* for a somewhat similar purpose, but the construction in the present work is much simpler, and Turing machines are more convenient to program than the counter machines used in the prior work.

To prove the gap result, we heavily rely on Chang and Pettie’s [8] ideas: they show that one can relate LCL problems in trees to regular languages and this way generate equivalent subtrees by “pumping”. However, there is one fundamental difference:

- Chang and Pettie first construct certain *universal* collections of tree fragments (that do not depend on the input graph), use the existence of a fast algorithm to show that these fragments can be labelled in a convenient way, and finally use such a labelling to solve any given input efficiently.
- We work directly with the *specific* input graph, expand it by “pumping”, and apply a fast algorithm there directly.

**Open problems.** Our work establishes a gap between  $\Theta(n^{1/2})$  and  $\Theta(n)$  in trees. The next natural step would be to generalise the result and establish a gap between  $\Theta(n^{1/(k+1)})$  and  $\Theta(n^{1/k})$  for all positive integers  $k$ .

## 2 Model and related work

As we study LCL problems, a family of problems defined on *bounded-degree graphs*, we assume that our input graphs are of degree at most  $\Delta$ , where  $\Delta = O(1)$  is a known constant. Each input graph  $G = (V, E)$  is simple, connected, and undirected; here  $V$  is the set of nodes and  $E$  is the set of edges, and we denote by  $n = |V|$  the total number of nodes in the input graph.

### 2.1 Model of computation

The model considered in this paper is the well studied LOCAL model [14, 18]. In the LOCAL model, each node  $v \in V$  of the input graph  $G$  runs the same deterministic algorithm. The nodes are labelled with unique  $O(\log n)$ -bit identifiers, and initially each node knows only its own identifier, its own degree, and the total number of nodes  $n$ .

Computation proceeds in synchronous rounds. At each round, each node

- sends a message to its neighbours (it may be a different message for different neighbours),
- receives messages from its neighbours,
- performs some computation based on the received messages.

In the LOCAL model, there is no restriction on the size of the messages or on the computational power of a node. Hence, after  $t$  rounds in the LOCAL model, each node has knowledge about the network up to distance  $t$  from him. The time complexity of an algorithm running in the LOCAL model is determined by this radius- $t$  that each node needs to explore in order to solve a given problem. It is easy to see that, in this setting, every problem can be solved in *diameter* time.

## 2.2 Locally checkable labellings

Locally checkable labelling problems (LCLs) were introduced in the seminal work of Naor and Stockmeyer [15]. These problems are defined on bounded degree graphs, so let  $\mathcal{F}$  be the family of such graphs. Also, let  $\Sigma_{\text{in}}$  and  $\Sigma_{\text{out}}$  be respectively input and output label alphabets. Each node  $v$  of a graph  $G \in \mathcal{F}$  has an input  $i(v) \in \Sigma_{\text{in}}$ , and must produce an output  $o(v) \in \Sigma_{\text{out}}$ . The output that each node must produce depends on the constraints defined with the LCL problem. Hence, let  $C$  be the set of legal configurations. A problem  $\Pi$  is an LCL problem if

- $\Sigma_{\text{in}}$  and  $\Sigma_{\text{out}}$  are of constant size;
- there exists an algorithm  $\mathcal{A}$  able to check the validity of a solution in constant time in the LOCAL model.

Hence, if the solution produced by the nodes is in the set  $C$  of valid configurations, then, by just looking at its local neighbourhood, each node must output ‘accept’, otherwise, at least one node must output ‘reject’. An example of an LCL problem is vertex colouring, where we have a constant size palette of colours; nodes can easily check in 1 round whether the produced colouring is valid or not.

## 2.3 Related work

**Cycles and paths.** LCL problems are fully understood in the case of cycles and paths. In these graphs it is known that there are LCL problems having complexities  $O(1)$ , e.g. trivial problems,  $\Theta(\log^* n)$ , e.g. 3 vertex-colouring, and  $\Theta(n)$ , e.g. 2 vertex-colouring [9, 14]. Chang, Kopelowitz, and Pettie [7] showed two automatic speedup results: any  $o(\log^* n)$ -time algorithm can be converted into an  $O(1)$ -time algorithm; any  $o(n)$ -time algorithm can be converted into an  $O(\log^* n)$ -time algorithm.

**Oriented grids.** Brandt et al. [5] studied LCL problems on oriented grids, showing that, as in the case of cycles and paths, the only possible complexities of LCLs are  $O(1)$ ,  $\Theta(\log^* n)$ , and  $\Theta(n)$ , on  $n \times n$  grids. However, while it is decidable whether a given LCL on cycles can be solved in  $t$ -rounds in the LOCAL model [5, 15], it is not the case for oriented grids [5].

**Trees.** Although well studied, LCLs on trees are not fully understood yet. Chang and Pettie [8] show that any  $n^{o(1)}$ -time algorithm can be converted into an  $O(\log n)$ -time algorithm. In the same paper they show how to obtain LCL problems on trees having deterministic and randomized complexity of  $\Theta(n^{1/k})$ , for any integer  $k$ . However, it is not known if there are problems of complexities between  $o(n^{1/k})$  and  $\omega(n^{1/(k+1)})$ .

**General graphs.** Another important direction of research is understanding LCLs on general (bounded-degree) graphs. Using the techniques presented by Naor and Stockmeyer [15], it is possible to show that any  $o(\log \log^* n)$ -time algorithm can be sped up to  $O(1)$  rounds. It is known that there are LCL problems with complexities  $\Theta(\log^* n)$  [2, 3, 10, 16] and  $\Theta(\log n)$  [4, 7, 11]. On the other hand, Chang et al. [7] showed that there are no LCL problems with deterministic complexities between  $\omega(\log^* n)$  and  $o(\log n)$ . It is known that there are problems (for example,  $\Delta$ -colouring) that require  $\Omega(\log n)$  rounds [4, 6], for which there are algorithms solving them in  $O(\text{polylog } n)$  rounds [17]. Until very recently, it was thought that there would be many other gaps in the landscape of complexities of LCL problems in general graphs. Unfortunately, it has been shown in [1] that this is not the case: it is possible to obtain LCLs with numerous different deterministic time complexities, including  $\Theta(\log^\alpha n)$  and  $\Theta(\log^\alpha \log^* n)$  for any  $\alpha \geq 1$ ,  $2^{\Theta(\log^\alpha n)}$ ,  $2^{\Theta(\log^\alpha \log^* n)}$ , and  $\Theta((\log^* n)^\alpha)$  for any  $\alpha \leq 1$ , and  $\Theta(n^\alpha)$  for any  $\alpha < 1/2$  (where  $\alpha$  is a positive rational number).

### 3 Near-linear complexities in general graphs

In this section we show the existence of LCL problems having complexities in the spectrum between  $\omega(\sqrt{n})$  and  $o(n)$ . We first give the definition of a standard model of computation, that is Linear Bounded Automata, and we then show that it is possible to encode the execution of an LBA as a locally checkable labelling. We then define an LCL problem where interesting instances are those in which one encodes the execution of a specific LBA in a multidimensional grid. Depending on the number of dimensions of the grid, and on the running time of the LBA, we obtain different time complexities.

#### 3.1 Linear bounded automata

A Linear Bounded Automaton (LBA)  $M_B$  consists of a Turing machine with a tape of bounded size  $B$ , able to recognize the boundaries of the tape [13, p. 225]. We consider a simplified version of LBAs, where the machine is initialized with an empty tape (no input is present). We describe this simplified version of LBAs as a 5-tuple  $M = (Q, q_0, f, \Gamma, \delta)$ , where:

- $Q$  is a finite set of states;
- $q_0 \in Q$  is the initial state;
- $f \in Q$  is the final state;
- $\Gamma$  is a finite set of tape alphabet symbols, containing a special symbol  $b$  (*blank*), and two special symbols,  $L$  and  $R$ , called *left* and *right* markers;
- $\delta: Q \setminus \{f\} \times \Gamma \rightarrow Q \times \Gamma \times \{-, \leftarrow, \rightarrow\}$  is the transition function.

The tape (of size  $B$ ) is initialized in the following way:

- the first cell contains the symbol  $L$ ;
- the last cell contains the symbol  $R$ ;
- all the other cells contain the symbol  $b$ .

The head is initially positioned on the cell containing the symbol  $L$ . Then, depending on the current state and the symbol present on the current position of the tape head, the machine enters a new state, writes a symbol on the current position, and moves to some direction.

In particular, the transition function  $\delta$  is going to be described by a finite set of 5-tuples  $(s_0, t_0, s_1, t_1, d)$  where:

1. The first 2 elements specify the input:
  - $s_0$  indicates the current state;
  - $t_0$  indicates the tape content on the current head position.
2. The remaining 3 elements specify the output:
  - $s_1$  is the new state;
  - $t_1$  is the new tape content on the current head position;
  - $d$  specifies the new position of the head:
    - ‘ $\rightarrow$ ’ means that the head moves to the next cell;
    - ‘ $\leftarrow$ ’ indicates that the head moves to the previous cell;
    - ‘ $-$ ’ means the head does not move.

If  $\delta$  is not defined on the current state and tape content, the machine terminates. The *growth* of an LBA  $M_B$ , denoted with  $g(M_B)$ , is defined as the running time of  $M_B$ . For example, it is easy to design a machine  $M$  that implements a binary counter, counting from all-0 to all-1, and this gives a growth of  $g(M_B) = \Theta(2^B)$ .

Also, it is possible to define a *unary  $k$ -counter*, that is, a list of  $k$  unary counters (where each one counts from 0 to  $B - 1$  and then overflows and starts counting from 0 again) in which when a counter overflows, the next is incremented. It is possible to achieve a growth

of  $g(M_B) = \Theta(B^k)$  by carefully implementing these counters (for example by using a single tape of length  $B$  to encode all the  $k$  counters at the cost of using more machine states and tape symbols).

### 3.2 Grid structure

Each LCL problem we will construct in Section 3.4 is designed in a way that ensures that the hardest input graphs for the LCL problem, i.e., the graphs providing the lower bound instances for the claimed time complexity, have a (multidimensional) grid structure. In this section, we introduce a class of graphs with this structure.

Let  $i \geq 2$  and  $d_1, \dots, d_i$  be positive integers. The set of nodes of an *i-dimensional grid graph*  $\mathcal{G}$  consists of all  $i$ -tuples  $u = (u_1, \dots, u_i)$  with  $0 \leq u_j \leq d_j$  for all  $1 \leq j \leq i$ . We call  $u_1, \dots, u_i$  the *coordinates* of node  $u$  and  $d_1, \dots, d_i$  the *sizes* of the *dimensions*  $1, \dots, i$ . Let  $u$  and  $v$  be two arbitrary nodes of  $\mathcal{G}$ . There is an edge between  $u$  and  $v$  if and only if  $\|u - v\|_1 = 1$ , i.e., all coordinates of  $u$  and  $v$  are equal, except one that differs by 1.

#### 3.2.1 Grid labels

In addition to the graph structure, we add *constant-size* labels to each grid graph. Each edge  $e = \{u, v\}$  is assigned two labels  $L_u(e)$  and  $L_v(e)$ , one for each endpoint. Label  $L_u(e)$  is chosen as follows:

- $L_u(e) = \text{Next}_j$  if  $v_j - u_j = 1$ ;
- $L_u(e) = \text{Prev}_j$  if  $u_j - v_j = 1$ .

Label  $L_v(e)$  is chosen analogously. If we want to focus on a specific label of some edge  $e$  and it is clear from the context which of the two edge labels is considered, we may refer to it simply as the label of  $e$ .

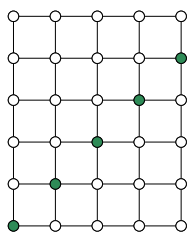
The labelling of the edges here is just a matter of convenience. We could equally well assign the labels to nodes instead of edges, satisfying the formal criteria of an LCL problem (and, for that matter, combine all input labels, and later output labels, of a node into a single input, resp. output, label). Furthermore, we could also equally well encode the labels in the graph structure. Hence all new time complexities presented in Section 3.4 can also be achieved by LCL problems without input labels.

In the full version of this paper we prove that, assuming that the considered graph contains a node not having any edge labelled with  $\text{Prev}_j$ , for all dimensions  $j$ , then nodes can locally check if they are in a valid grid graph.

#### 3.2.2 Unbalanced grid graphs

In Section 3.2.1, we saw the basic idea behind ensuring that non-grid graphs are not among the hardest instances for the LCL problems we construct. In this section, we will study the ingredient of our LCL construction that guarantees that grid graphs where the dimensions have “wrong” sizes are not worst-case instances. More precisely, we want that the hardest instances for our LCL problems are grid graphs with the property that there is at least one dimension  $2 \leq j \leq i$  whose size is not larger than the size of dimension 1. In the following, we will show how to make sure that *unbalanced* grid graphs, i.e., grid graphs that do not have this property, allow nodes to find a valid output without having to see too far. In a sense, in any constructed LCL, a locally checkable *proof* (of a certain well-specified kind) certifying that the input graph is an unbalanced grid graph constitutes a valid (global) output.





■ **Figure 1** An example of an unbalanced grid with 2 dimensions; nodes in green are labelled with Unbalanced, while white nodes are labelled with Exempt.

Consider a grid graph with  $i$  dimensions of sizes  $d_1, \dots, d_i$ . If  $d_1 < d_j$  for all  $2 \leq j \leq i$ , the following output labelling is regarded as correct in any constructed LCL problem:

- For all  $0 \leq t \leq d_1$ , node  $v = (v_1, \dots, v_i)$  satisfying  $v_1 = \dots = v_i = t$  is labelled Unbalanced.
- All other nodes are labelled Exempt.

This labelling is clearly locally checkable, i.e., it can be described as a collection of local constraints: Each node  $v$  labelled Unbalanced checks that it has exactly two “diagonal neighbours” and that their positions relative to  $v$  are consistent with the above output specification. Node  $v$  also may have only one diagonal neighbour, but only if it has no incident edge labelled  $\text{Prev}_j$ , or if it has an incident edge labelled  $\text{Next}_j$  for all  $2 \leq j \leq i$ , but no incident edge labelled  $\text{Next}_1$ . The latter condition ensures that the described diagonal chain of labels terminates at the end of dimension 1, but not at the end of any other dimension, thereby guaranteeing that grid graphs that are not unbalanced do not allow the output labelling specified above. Finally, the unique node without any incident edge labelled  $\text{Prev}_j$  checks that it is labelled Unbalanced, in order to prevent the possibility that each node simply outputs Exempt. We refer to Figure 1 for an example of an unbalanced 2-dimensional grid and its labelling.

### 3.3 Machine encoding

After examining the cases of the input graph being a non-grid graph or an unbalanced grid graph, in this section, we turn our attention towards the last remaining case: that is the input graph is actually a grid graph for which there is a dimension with size smaller than or equal to the size of dimension 1. In this case, we require the nodes to work together to create a global output that is determined by some LBA. Essentially, the execution of the LBA has to be written (as node outputs) on a specific part of the grid graph. In order to formalise this relation between the desired output and the LBA, we introduce the notion of an LBA *encoding graph* in the following.

#### 3.3.1 Labels

Let  $M_B$  be an LBA, where  $B$  denotes the size of the tape. Let  $S_\ell = (s_\ell, h_\ell, t_\ell)$  be the whole state of  $M_B$  after step  $\ell$ , where  $s_\ell$  is the machine internal state,  $h_\ell$  is the position of the head, and  $t_\ell$  is the whole tape content. The content of the cell in position  $y \in \{0, \dots, B-1\}$  after step  $\ell$  is denoted by  $t_\ell[y]$ . We denote by  $(x, y)_k$  the node  $v = (v_1, \dots, v_i)$  having  $v_1 = x$ ,  $v_k = y$ , and  $v_j = 0$  for all  $j \notin \{1, k\}$ . An (output-labelled) grid graph of dimension  $i$  is an LBA *encoding graph* if there exists a dimension  $2 \leq k \leq i$  satisfying the following.

- $d_k + 1$  is equal to  $B$ .
- For all  $0 \leq x \leq \min\{g(M_B), d_1\}$  and all  $0 \leq y \leq B - 1$ , it holds that:
  - Node  $(x, y)_k$  is labelled with  $\text{Tape}(t_x[y])$ .
  - Node  $(x, y)_k$  is labelled with  $\text{State}(s_x)$ .
  - Node  $(x, h_x)_k$  is labelled with  $\text{Head}$ .
  - Node  $(x, y)_k$  is labelled with  $\text{Dimension}(k)$ .
- All other nodes are labelled with  $\text{Exempt}$ .

Intuitively, the 2-dimensional surface expanding in dimensions 1 and  $k$  (having all the other coordinates equal to 0), encodes the execution of the LBA. The described labelling is locally checkable, see the full version of this paper for details.

### 3.4 LCL construction

Fix an integer  $i \geq 2$ , and let  $M$  be an LBA with growth  $g$ . As we do not fix a specific size of the tape,  $g$  can be seen as a function that maps the tape size  $B$  to the running time of the LBA executed on a tape of size  $B$ . We now construct an LCL problem  $\Pi_M$  with complexity related to  $g$ . Note that  $\Pi_M$  depends on the choice of  $i$ . The general idea of the construction is that nodes can either:

- produce a valid LBA encoding, or
- prove that dimension 1 is too short, or
- prove that there is an error in the (grid) graph structure.

We need to ensure that on balanced grid graphs it is not easy to claim that there is an error, while allowing an efficient solution on invalid graphs, i.e., graphs that contain a local error (some invalid label), or a global error (a grid structure that wraps, or dimension 1 too short compared to the others).

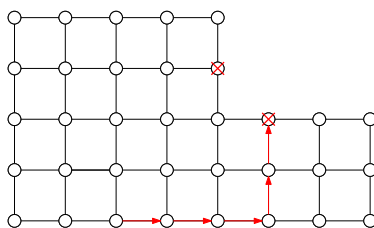
#### 3.4.1 LCL Problem $\Pi_M$

Denote by  $\mathcal{L}$  the set of output labels used for producing an LBA encoding graph. Formally, we specify the LCL problem  $\Pi_M$  as follows. The input label set for  $\Pi_M$  is the set of labels used in the grid labelling. The possible output labels are the following:

1. the labels from  $\mathcal{L}$ ;
2. an *unbalanced label*,  $\text{Unbalanced}$ ;
3. an *exempt label*,  $\text{Exempt}$ ;
4. an *error label*  $\text{Error}$ ;
5. *error pointers*, i.e., all possible pairs  $(s, r)$ , where  $s$  is either  $\text{Next}_j$  or  $\text{Prev}_j$  for some  $1 \leq j \leq i$ , and  $r \in \{0, 1\}$  is a bit whose purpose it is to distinguish between two different types of error pointers, *type 0* pointers and *type 1* pointers.

Note that the separate mention of  $\text{Exempt}$  in this list is not strictly necessary since  $\text{Exempt}$  is contained in  $\mathcal{L}$ , but we want to recall the fact that  $\text{Exempt}$  can be used in both a proof of unbalance and an LBA encoding.

Intuitively, nodes that notice that there is/must be an error in the grid structure, but are not allowed to output  $\text{Error}$  because the grid structure is valid *in their local neighborhood*, can point in the direction of an error. However, the nodes have to make sure that the error pointers form a chain that actually ends in an error. In order to make the proofs in this section more accessible, we distinguish between the two types of error pointers mentioned above; roughly speaking, type 0 pointers will be used by nodes that (during the course of the algorithm) cannot see an error in the grid structure, but notice that the grid structure



■ **Figure 2** An example of an error pointer chain (shown in red). Nodes that are marked with a red cross are those who actually see an error in the grid structure. The output of only some of the depicted nodes is shown.

wraps around in some way, while type 1 pointers are for nodes that can actually see an error. If the grid structure wraps around, then there must be an error somewhere (and nodes that see that the grid structure wraps around know where to point their error pointer to), except in the case that the grid structure wraps around “nicely” (e.g., along one dimension). This exceptional case is the only scenario where, deviating from the above, an error pointer chain does not necessarily end in an error, but instead may form a cycle; however, since the constraints we put on error pointer chains are *local* constraints (as we want to define an LCL problem), the global behaviour of the chain is irrelevant. We will not explicitly prove the global statements made in this informal overview; for our purposes it is sufficient to focus on the local views of nodes.

Note that if a chain of type 0 error pointers does not cycle, then at some point it will turn into a chain of type 1 error pointers, which in turn will end in an error. Chains of type 1 error pointers cannot cycle. We refer to Figure 2 for an example of an error pointer chain.

An output labelling for problem  $\Pi_M$  is correct if the following conditions are satisfied.

1. Each node  $v$  produces at least one output label. If  $v$  produces at least two output labels, then all of  $v$ 's output labels are contained in  $\mathcal{L} \setminus \{\text{Exempt}\}$ .
2. Each node at which the input labelling does not satisfy the local grid graph constraints given in Section 3.2.1 outputs **Error**. All other nodes do not output **Error**.
3. If a node  $v$  outputs **Exempt**, then  $v$  has at least one incident edge  $e$  with input label  $L_v(e) \in \{\text{Prev}_1, \dots, \text{Prev}_i\}$ .
4. If the output labels of a node  $v$  are contained in  $\mathcal{L} \setminus \{\text{Exempt}\}$ , then either there is a node in  $v$ 's 2-radius neighbourhood that outputs an error pointer, or the output labels of all nodes in  $v$ 's 2-radius neighbourhood are contained in  $\mathcal{L}$ . Moreover, in the latter case  $v$ 's 2-radius neighbourhood has a valid grid structure and the local constraints of an LBA encoding graph, given in Section 3.1, are satisfied at  $v$ .
5. If the output of a node  $v$  is **Unbalanced**, then either there is a node in  $v$ 's  $i$ -radius neighbourhood that outputs an error pointer, or the output labels of all nodes in  $v$ 's  $i$ -radius neighbourhood are contained in  $\{\text{Unbalanced}, \text{Exempt}\}$ . Moreover, in the latter case  $v$ 's  $i$ -radius neighbourhood has a valid grid structure and the local constraints for a proof of unbalance, given in Section 3.2.2, are satisfied at  $v$ .
6. Let  $v$  be a node that outputs an error pointer  $(s, r)$ . Then  $z_v(s)$  is defined, i.e., there is exactly one edge incident to  $v$  with input label  $s$ . Let  $u$  be the neighbour reached by following this edge from  $v$ , i.e.,  $u = z_v(s)$ . Then  $u$  outputs either **Error** or an error pointer  $(s', r')$ , where in the latter case the following hold:
  - $r' \geq r$ , i.e., the type of the pointer cannot decrease when following a chain of error pointers;

## 9:10 Almost Global Problems in the LOCAL Model

- if  $r' = 0 = r$ , then  $s' = s$ , i.e., the pointers in a chain of error pointers of type 0 are consistently oriented;
- if  $r' = 1 = r$  and  $s \in \{\text{Prev}_j, \text{Next}_j\}$ ,  $s' \in \{\text{Prev}_{j'}, \text{Next}_{j'}\}$ , then  $j' \geq j$ , i.e., when following a chain of error pointers of type 1, the dimension of the pointer cannot decrease;
- if  $r' = 1 = r$  and  $s, s' \in \{\text{Prev}_j, \text{Next}_j\}$  for some  $1 \leq j \leq i$ , then  $s' = s$ , i.e., any two subsequent pointers in the same dimension have the same direction.

These conditions are clearly locally checkable, so  $\Pi_M$  is a valid LCL problem.

### 3.4.2 Time complexity

Let  $B$  be the smallest positive integer satisfying  $n \leq B^{i-1} \cdot g(M_B)$ . We will only consider LBAs with the property that  $B \leq g(M_B)$  and for any two tape sizes  $B_1 \geq B_2$  we have  $g(M_{B_1}) \geq g(M_{B_2})$ . The LCL problem  $\Pi_M$  has time complexity  $\Theta(n/B^{i-1}) = \Theta(g(M_B))$ . The following theorem is proved in the full version of this paper.

► **Theorem 1.** *Problem  $\Pi_M$  has time complexity  $\Theta(g(M_B))$ .*

### 3.4.3 Instantiating the LCL construction

Our construction is quite general and allows to encode a wide variety of LBAs to obtain many different LCL complexities. As a proof of concept, we show some complexities that can be obtained using some specific LBAs.

- By using a  $k$ -unary counter, for constant  $k$ , we obtain a growth of  $\Theta(B^k)$ .
- By using a binary counter, we obtain a growth of  $\Theta(2^B)$ .

► **Theorem 2.** *For any rational number  $0 \leq \alpha \leq 1$ , there exists an LCL problem with time complexity  $\Theta(n^\alpha)$ .*

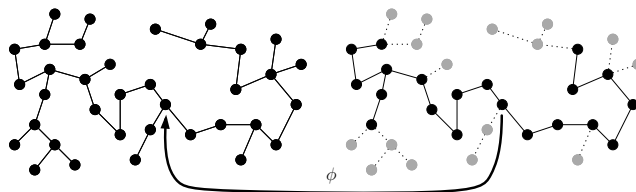
**Proof.** Let  $j > k$  be positive integers satisfying  $\alpha = k/j$ . Given an LBA with growth  $\Theta(B^k)$  and using a  $(j - k + 1)$ -dimensional grid graph, we obtain an LCL problem with complexity  $\Theta(n/B^{j-k})$ . We have that  $n = \Theta(B^{j-k} \cdot g(M_B)) = \Theta(B^j)$ , which implies  $B = \Theta(n^{1/j})$ . Thus the time complexity of our LCL problem is  $\Theta(n/n^{(j-k)/j}) = \Theta(n^\alpha)$ . ◀

► **Theorem 3.** *There exist LCL problems of complexities  $\Theta(\frac{n}{\log^i n})$ , for any positive integer  $i$ .*

**Proof.** Given an LBA with growth  $\Theta(2^B)$  and using an  $(i + 1)$ -dimensional grid graph, we obtain an LCL problem with complexity  $\Theta(n/B^i)$ . We have that  $n = \Theta(B^i \cdot g(M_B)) = \Theta(B^i \cdot 2^B)$ , which implies  $B = \Theta(\log n)$ . Thus the time complexity of our LCL problem is  $\Theta(n/\log^i n)$ . ◀

## 4 Complexity gap on trees

In this section we prove that, on trees, there are no LCLs having complexity  $T$  between  $\omega(\sqrt{n})$  and  $o(n)$ . We show that, given an algorithm  $\mathcal{A}$  that solves a problem in time  $T$ , it is possible to speed up its running time to  $O(\sqrt{n})$ , by first constructing a virtual tree  $S$  in which a ball of radius  $T$  corresponds to a ball of radius  $O(\sqrt{n})$  of the original graph, and then find a valid output for the original graph, having outputs for the virtual graph  $S$ .



■ **Figure 3** Example of a tree  $T$  and its skeleton  $T'$ ; nodes removed from  $T$  in order to obtain  $T'$  are shown in gray. In this example,  $\tau$  is 3.

## 4.1 Skeleton tree

We first describe how, starting from a tree  $T = (V, E)$ , nodes can distributedly construct a virtual tree  $T'$ , called the *skeleton* of  $T$ . Intuitively,  $T'$  is obtained by removing all subtrees of  $T$  having a height that is less than some threshold  $\tau$ .

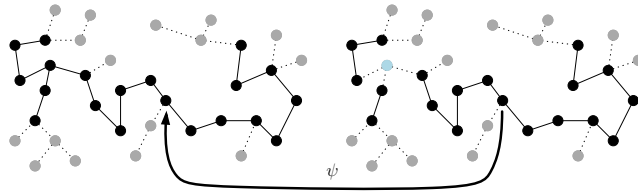
More formally, let  $\tau = c\sqrt{n}$ , for some constant  $c$  that will be fixed later. Each node  $v$  starts by gathering its  $\tau$ -radius neighbourhood,  $\text{Ball}_v$ . Also, let  $d_v$  be the degree of node  $v$  in  $T$ . We partition  $\text{Ball}_v$ ,  $\forall v \in V$ , in  $d_v$  components (one for each neighbour of  $v$ ), and let us denote these components with  $C_i(v)$ , where  $1 \leq i \leq d_v$ . Each component  $C_i(v)$  contains all nodes of  $\text{Ball}_v$  present in the subtree rooted at the  $i$ -th neighbour of  $v$ , excluding  $v$ .

Then, each node marks as *Del* all the components that have low depth and broadcasts this information. Informally, nodes build the skeleton tree by removing all the components that are marked as *Del* by at least one node. More precisely, each node  $v$ , for each  $C_i(v)$ , if  $\text{dist}(v, w) < \tau$  for all  $w$  in  $V(C_i(v))$ , marks all edges in  $E(C_i(v)) \cup \{\{v, u\}\}$  as *Del*, where  $u$  is the  $i$ -th neighbor of  $v$ . Then,  $v$  broadcasts  $\text{Ball}_v$  and the edges marked as *Del* to all nodes at distance at most  $\tau + 2c$ . Finally, when a node  $v$  receives messages containing edges that have been marked with *Del* by some node, then also  $v$  internally marks as *Del* those edges.

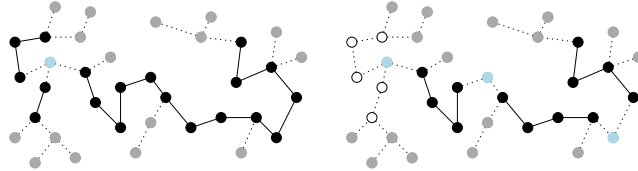
Now we have all the ingredients to formally describe how we construct the skeleton tree. The skeleton tree  $T' = (V', E')$  is defined in the following way. Intuitively, we keep only edges that have not been marked *Del*, and nodes with at least one remaining edge (i.e., nodes that have at least one incident edge not marked with *Del*). In particular,  $E' = \{e \in E(T) \mid e \text{ is not marked Del}\}$ , and  $V' = \{u \in V \mid \exists w \in V \text{ s.t. } \{u, w\} \in E'\}$ . Also, we want to keep track of the mapping from a node of  $T'$  to its original node in  $T$ ; let  $\phi$  be such a mapping. Finally, we want to keep track of deleted subtrees, so let  $\mathcal{T}_v$  be the subtree of  $T$  rooted at  $v \in V'$  containing all nodes of  $C_j(v)$ , for all  $j$  such that  $C_j(v)$  has been marked as *Del*. See Figure 3 for an example.

## 4.2 Virtual tree

We now show how to distributedly construct a new virtual tree, starting from  $T'$ , that satisfies some useful properties. Informally, the new tree is obtained by *pumping* all paths contained in  $T'$  having length above some threshold. More precisely, by considering only degree-2 nodes of  $T'$  we obtain a set of paths. We split these paths in shorter paths of length  $l$  ( $c \leq l \leq 2c$ ) by computing a  $(c + 1, c)$  ruling set. Then, we pump these paths in order to obtain the final tree. Recall a  $(\alpha, \beta)$ -ruling set  $R$  of a graph  $G$  guarantees that nodes in  $R$  have distance at least  $\alpha$ , while nodes outside  $R$  have at least one node in  $R$  at distance at most  $\beta$ . It can be distributedly computed in  $O(\log^* n)$  rounds using standard colouring algorithms [14].



■ **Figure 4** Example of the tree  $T''$  obtained from  $T'$ ; nodes with degree greater than 2 (in blue) are removed from  $T'$ .



■ **Figure 5** Blue nodes break the long paths  $\mathcal{P}$  of  $T''$  shown on the left into short paths  $\mathcal{Q}$  shown in black on the right; short paths (in the example, paths with length less than 4) are ignored.

More formally, we start by splitting the tree in many paths of short length. Let  $v$  a node in  $V'$  and  $d_v^{T'}$  its degree in  $T'$ . Let  $T''$  be the forest obtained by removing from  $T'$  each node  $v$  having  $d_v^{T'} > 2$ .  $T''$  is a collection  $\mathcal{P}$  of disjoint paths. Let  $\psi$  be the mapping from nodes of  $T''$  to their corresponding node in  $T'$ . See Figure 4 for an example.

We now want to split long paths of  $\mathcal{P}$  in shorter paths. In order to achieve this, nodes of the same path can efficiently find a  $(c + 1, c)$  ruling set in the path containing them. Nodes not in the ruling set form short paths of length  $l$ , such that  $c \leq l \leq 2c$ , except for some paths of  $\mathcal{P}$  that were already too short, or subpaths at the two ends of a longer path. Let  $\mathcal{Q}$  be the subset of the resulting paths having length  $l$  satisfying  $c \leq l \leq 2c$ . See Figure 5 for an example.

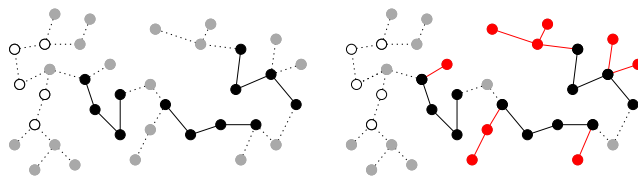
In order to obtain the final tree, we use the following function, called **Replace**. Informally, given a graph  $G$  and a subgraph  $H$  connected to the other nodes of  $G$  via a set of nodes  $F$ , called poles, and given another graph  $H'$ , it replaces  $H$  with  $H'$ . This function is a simplified version of the function **Replace** presented in [8] in Section 3.3.

► **Definition 4 (Replace)**. Let  $H$  be a subgraph of  $G$ . The poles of  $H$  are those vertices in  $V(H)$  adjacent to some vertex in  $V(G) \setminus V(H)$ . Let  $F = (v_1, \dots, v_p)$  be a list of the poles of  $H$ , and let  $F' = (v'_1, \dots, v'_p)$  be a list of nodes contained in  $H'$  (called poles of  $H'$ ). The graph  $G' = \text{Replace}(G, (H, F), (H', F'))$  is defined in the following way. Start with  $G$ , replace  $H$  with  $H'$ , and replace any edge  $\{u, v_i\}$ , where  $u \in V(G) \setminus V(H)$ , with  $\{u, v'_i\}$ .

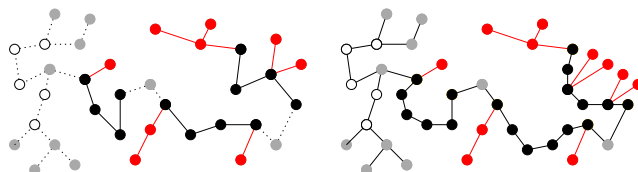
Informally, we will use the function **Replace** to substitute each path  $Q \in \mathcal{Q}$  with a longer version of it, that satisfies some useful properties. We will later define a function, **Pump**, that is used to obtain these longer paths. The function **Pump** is defined in an analogous way to the function **Pump** presented in [8] in Section 3.8. We now show which properties it satisfies.

► **Definition 5 (Properties of Pump)**. Given a path  $Q \in \mathcal{Q}$  of length  $l$  ( $c \leq l \leq 2c$ ), consider the subgraph  $Q^T$  of  $T$ , containing, for each  $v \in V(Q)$ , the tree  $\mathcal{T}_{\chi(v)}$ , where  $\chi(v) = \phi(\psi(v))$ , that is, the path  $Q$  augmented with all the nodes deleted from the original tree that are connected to nodes of the path. Let  $v_1, v_2$  be the endpoints of  $Q$ .

The function  $\text{Pump}(Q^T, B)$  produces a new tree  $P^T$  having two endpoints,  $v'_1$  and  $v'_2$ , satisfying that the path between  $v'_1$  and  $v'_2$  has length  $l'$ , such that  $cB \leq l' \leq c(B + 1)$ . The new tree is obtained by replacing a subpath of  $Q$ , along with the deleted nodes connected to it, with many copies of the replaced part, concatenated one after the other. Let



■ **Figure 6** Example of  $Q^T$ , obtained by merging the path nodes (in black) with previously removed trees connected to them (in red).



■ **Figure 7**  $S$  (on the right) is obtained by pumping the black paths.

$G' = \text{Replace}(G, (Q^T, (v_1, v_2)), (P^T, (v'_1, v'_2)))$ . Pump satisfies that nodes  $v'_1, v'_2 \in G'$  have the same view as  $v_1, v_2 \in G$  at distance  $2r$  (where  $r$  is the LCL checkability radius). Note that, in the formal definition of Pump, we will set  $c$  as a function of  $r$ .

See Figure 6 for an example of  $Q^T$ .

The final tree  $S$  is obtained from  $T$  by replacing each path  $Q \in \mathcal{Q}$  in the following way. Let  $\mathcal{Q}^T$  be the set containing all  $Q^T$ . Replace each subgraph  $Q^T$  with  $P^T = \text{Pump}(Q^T, B)$ . Note that a node  $v$  can not see the whole set  $\mathcal{Q}$ , but just all the paths  $Q \in \mathcal{Q}$  that end at distance at most  $\tau + 2c$  from  $v$ . Thus each node locally computes just a part of  $S$ , that is enough for our purpose. We call the subgraph of  $Q^T$  induced by the nodes of  $Q$  the *main path* of  $Q^T$ , and we define the main path of  $P^T$  in an analogous way. See Figure 7 for an example.

Finally, we want to keep track of the *real* nodes of  $S$ . Nodes of  $S$  are divided in two parts,  $S_o$  and  $S_p$ . The set  $S_o$  contains all nodes of  $T'$  that are not contained in any  $Q^T$ , and all nodes that are at distance at most  $2r$  from nodes not contained in any  $Q^T$ , while  $S_p = V(S) \setminus S_o$ . Let  $\eta$  be a mapping from real nodes of the virtual graph ( $S_o$ ) to their corresponding node of  $T$  (this is well defined, by the properties of Pump), and let  $T_o = \{\eta(v) \mid v \in S_o\}$  (note that also  $\eta^{-1}$  is well defined for nodes in  $T_o$ ). Informally,  $T_o$  is the subset of nodes of  $T$  that are far enough from pumped regions of  $S$ , and have not been removed while creating  $T'$ . Note that we use the function  $\eta$  to distinguish between nodes of  $S$  and nodes of  $T$ , but  $\eta$  is actually the identity function between a subset of shared nodes. Let Virt be the function that maps  $T$  to  $S$ , that is,  $S = \text{Virt}(T, B, c)$ . See Figure 8 for an example.

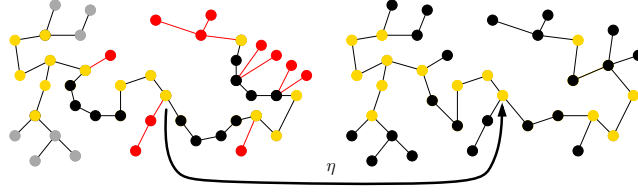
### 4.3 Properties of the virtual tree

The following lemma bounds the size of the graph  $S$ , compared to the size of  $T$ .

► **Lemma 6.** *The tree  $S$  has at most  $N = c(B + 1)n$  nodes, where  $n = |V(T)|$ , and  $S = \text{Virt}(T, B, c)$ .*

**Proof.**  $S$  is obtained by pumping  $T$ . The main path of the subtree obtained by pumping some  $Q^T \in \mathcal{Q}^T$  has length at most  $c(B + 1)$ . This implies that each node of the main path of  $Q^T$  is copied at most  $c(B + 1)$  times. Also, a deleted tree  $\mathcal{T}_v$  rooted at some path node  $v$  is not connected to more than one path node. Thus, *all* nodes of  $T$  are copied at most  $c(B + 1)$  times. ◀





■ **Figure 8** Nodes in yellow on the left are the ones in  $S_o$ , while the yellow ones on the right are nodes in  $T_o$ . Note that, for the sake of simplicity, we consider  $2r = 1$ .

The following lemma bounds the size of  $T'$  compared to the size of  $T''$ . Notice that, this is the exact point in which our approach stops working for time complexities of  $O(\sqrt{n})$  rounds. This is exactly what we expect, since we know that there are LCL problems on trees having complexity  $\Theta(\sqrt{n})$  [8].

► **Lemma 7.** *For any path  $P = (x_1, \dots, x_k)$  of length  $k \geq c\sqrt{n}$  that is a subgraph of  $T'$ , at most  $\frac{\sqrt{n}}{c}$  nodes in  $V(P)$  have degree greater than 2.*

**Proof.** If a node  $x_j \in P$  has  $d_v^{T'} > 2$ , it means that it has at least one neighbour  $z \notin \{x_{j-1}, x_{j+1}\}$  in  $T'$  such that there exists a node  $w$  satisfying  $\text{dist}(x_j, w) \geq \tau$  such that the shortest path connecting  $x_j$  and  $w$  contains  $z$ . Thus, for each node in  $P$  with  $d_v^{T'} > 2$ , we have at least other  $\tau$  nodes not in  $P$ . If at least  $\frac{\sqrt{n}}{c} + 1$  nodes of  $P$  have degree greater than 2, we would obtain a total of  $(\frac{\sqrt{n}}{c} + 1) \cdot \tau > n$  nodes, a contradiction. ◀

The following lemma compares distances in  $T$  with distances in  $S$ .

► **Lemma 8.** *There exists some constant  $c$  such that, if nodes  $u, v$  of  $T_o$  are at distance at least  $c\sqrt{n}$  in  $T$ , then their corresponding nodes  $\eta^{-1}(u)$  and  $\eta^{-1}(v)$  are at distance at least  $cB\sqrt{n}/3$  in  $S$ .*

**Proof.** Consider a node  $u$  at distance at least  $\tau$  from  $v$  in  $T$ . There must exist a path  $P$  in  $T'$  connecting  $\phi^{-1}(u)$  and  $\phi^{-1}(v)$ . By Lemma 7, at most  $\frac{\sqrt{n}}{c}$  nodes in  $P$  have degree greater than 2, call the set of these nodes  $X$ . We can bound the number of nodes of  $P$  that are not part of paths that will be pumped in the following way:

- At most  $\frac{c\sqrt{n}+1}{c+1} + \frac{\sqrt{n}}{c} + 1$  nodes can be part of the ruling set. To see this, order the nodes of  $P$  from left to right in one of the two canonical ways. The first summand bounds all the ruling set nodes whose right-hand short path is of length at least  $c$ , the second one bounds the ruling set nodes whose right-hand short path ends in a node  $x \in X$ , and the last one considers the path that ends in  $\phi^{-1}(u)$  or  $\phi^{-1}(v)$ .
- At most  $\frac{\sqrt{n}}{c}(1 + 2(c-1))$  nodes are either in  $X$  or in short paths of length at most  $c-1$  on the sides of a node in  $X$ .
- At most  $2(c-1)$  nodes are between  $\phi^{-1}(u)$  (or  $\phi^{-1}(v)$ ) and a ruling set node.

While pumping the graph, in the worst case we replace paths of length  $2c$  with paths of length  $cB$ , thus  $\text{dist}(\phi^{-1}(u), \phi^{-1}(v)) \geq (c\sqrt{n}+1 - (\frac{c\sqrt{n}+1}{c+1} + \frac{\sqrt{n}}{c} + 1 + \frac{\sqrt{n}}{c}(1+2(c-1)) + 2(c-1))) \cdot \frac{cB}{2c} - 1$ , which is greater than  $cB\sqrt{n}/3$  for  $c$  and  $n$  greater than a large enough constant. ◀

#### 4.4 Solving the problem faster

We now show how to speed up the algorithm  $\mathcal{A}$  and obtain an algorithm running in  $O(\sqrt{n})$ . First, note that if the diameter of the original graph is  $O(\sqrt{n})$ , every node sees the whole graph in  $O(\sqrt{n})$  rounds, and the problem is trivially solvable by brute force. Thus, in the following we assume that the diameter of the graph is  $\omega(\sqrt{n})$ . This also guarantees that  $T_o$  is not empty.



Informally, nodes can distributedly construct the virtual tree  $S$  in  $O(\sqrt{n})$  rounds, and safely execute the original algorithm on it. Intuitively, even if a node  $v$  sees just a part of  $S$ , we need to guarantee that this part has large enough radius, such that the original algorithm can't see outside the subgraph of  $S$  constructed by  $v$ .

More precisely, all nodes do the following. First, they distributedly construct  $S$ , in  $O(\sqrt{n})$  rounds. Then, each node  $v$  in  $T_o$  (nodes for which  $\eta^{-1}(v)$  is defined), simulates the execution of  $\mathcal{A}$  on node  $\eta^{-1}(v)$  of  $S$ , by telling  $\mathcal{A}$  that there are  $N = c(B+1)n$  nodes. Then, each node  $v$  in  $T_o$  outputs the same output assigned by  $\mathcal{A}$  to node  $\eta^{-1}(v)$  in  $S$ . Also, each node  $v$  in  $T_o$  fixes the output for all nodes in  $\mathcal{T}_v$  ( $\eta$  can be defined also for them,  $v$  sees all of them, and the view of these nodes is contained in the view of  $v$ , thus it can simulate  $\mathcal{A}$  in  $S$  for all of them). Let  $\Lambda$  be the set of nodes that already fixed an output, that is,  $\Lambda = \{\{u\} \cup V(\mathcal{T}_u) \mid u \in T_o\}$ . Intuitively  $\Lambda$  contains all the real nodes of  $S$  (nodes with a corresponding node in  $T$ ) and leaves out only nodes that correspond to pumped regions. Finally, nodes in  $V(T) \setminus \Lambda$  find a valid output via bruteforce.

We need to prove two properties, the first shows that a node can safely execute  $\mathcal{A}$  on the subgraph of  $S$  that it knows, while the second shows that it is always possible to find a valid output for nodes in  $V(T) \setminus \Lambda$  after having fixed outputs for nodes in  $\Lambda$ .

Let us choose a  $B$  satisfying  $\tau_{\text{orig}}(N) \leq cB\sqrt{n}/3$ , where  $\tau_{\text{orig}}(N)$  is the running time of  $\mathcal{A}$ . Note that  $B$  can be an arbitrarily large function of  $n$ . Such a  $B$  exists for all  $\tau_{\text{orig}}(x) = o(x)$ . We prove the following lemma.

► **Lemma 9.** *For nodes in  $T_o$ , it is possible to execute  $\mathcal{A}$  on  $S$  by just knowing the neighbourhood of radius  $2c\sqrt{n}$  in  $T$ .*

**Proof.** First, note that by Lemma 6, the number of nodes of the virtual graph,  $|V(S)|$ , is always at most  $N$ , thus, it is not possible that a node of  $S$  sees a number of nodes that is more than the number claimed when simulating the algorithm.

Second, since  $B$  satisfies  $\tau_{\text{orig}}(N) \leq cB\sqrt{n}/3$ , and since, by Lemma 8 and the bound of  $c\sqrt{n}$  on the depth of each deleted tree  $\mathcal{T}_u$ , the nodes outside a  $2c\sqrt{n}$  ball of nodes in  $T_o$  are at distance at least  $cB\sqrt{n}/3$  in  $S$ , the running time of  $\mathcal{A}$  is less than the radius of the subtree of  $S$  rooted at a node  $v$  that  $v$  distributedly computed and is aware of. This second part also implies that nodes in  $T_o$  do not see the whole graph, thus they cannot notice that the value of  $N$  is not the real size of the graph. ◀

## 4.5 Filling gaps by bruteforce

Using similar techniques presented in [8] we can show that, by starting from a tree  $T$  in which nodes of  $\Lambda$  have already fixed an output, we can find a valid output for all the other nodes of the graph, in constant time. See the full version for the details.

---

### References

- 1 Alkida Balliu, Juho Hirvonen, Janne H. Korhonen, Tuomo Lempinen, Dennis Olivetti, and Jukka Suomela. New classes of distributed time complexity. In *Proc. 50th Annual Symposium on the Theory of Computing (STOC 2018)*. ACM, 2018 (to appear). arXiv:1711.01871.
- 2 Leonid Barenboim. Deterministic  $(\Delta + 1)$ -coloring in sublinear (in  $\Delta$ ) time in static, dynamic, and faulty networks. *Journal of the ACM*, 63(5):47:1–47:22, 2016. doi:10.1145/2979675.
- 3 Leonid Barenboim, Michael Elkin, and Fabian Kuhn. Distributed  $(\Delta + 1)$ -coloring in linear (in  $\Delta$ ) time. *SIAM Journal on Computing*, 43(1):72–95, 2014. doi:10.1137/12088848X.

- 4 Sebastian Brandt, Orr Fischer, Juho Hirvonen, Barbara Keller, Tuomo Lempäinen, Joel Rybicki, Jukka Suomela, and Jara Uitto. A lower bound for the distributed Lovász local lemma. In *Proc. 48th Annual Symposium on the Theory of Computing (STOC 2016)*, pages 479–488. ACM, 2016. doi:10.1145/2897518.2897570.
- 5 Sebastian Brandt, Juho Hirvonen, Janne H. Korhonen, Tuomo Lempäinen, Patric R.J. Östergård, Christopher Purcell, Joel Rybicki, Jukka Suomela, and Przemysław Uznański. LCL problems on grids. In *Proc. 35th ACM Symposium on the Principles of Distributed Computing (PODC 2017)*, pages 101–110, 2017. doi:10.1145/3087801.3087833.
- 6 Yi-Jun Chang, Qizheng He, Wenzheng Li, Seth Pettie, and Jara Uitto. The complexity of distributed edge colouring with small palettes. In *Proc. 29th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2018)*. Society for Industrial and Applied Mathematics, 2018.
- 7 Yi-Jun Chang, Tsvi Kopelowitz, and Seth Pettie. An exponential separation between randomized and deterministic complexity in the LOCAL model. In *Proc. 57th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2016)*, pages 615–624. IEEE, 2016. arXiv:1602.08166.
- 8 Yi-Jun Chang and Seth Pettie. A time hierarchy theorem for the LOCAL model. In *Proc. 58th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2017)*, 2017. arXiv:1704.06297.
- 9 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.
- 10 Pierre Fraigniaud, Marc Heinrich, and Adrian Kosowski. Local conflict coloring. In *Proc. 57th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2016)*, pages 625–634, 2016. doi:10.1109/FOCS.2016.73.
- 11 Mohsen Ghaffari and Hsin-Hao Su. Distributed degree splitting, edge coloring, and orientations. In *Proc. 28th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2017)*, pages 2505–2523. Society for Industrial and Applied Mathematics, 2017. doi:10.1137/1.9781611974782.166.
- 12 Juris Hartmanis and Richard Edwin Stearns. On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, 117:285–306, 1965. doi:10.1090/S0002-9947-1965-0170805-7.
- 13 John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.
- 14 Nathan Linial. Locality in distributed graph algorithms. *SIAM Journal on Computing*, 21(1):193–201, 1992. doi:10.1137/0221015.
- 15 Moni Naor and Larry Stockmeyer. What can be computed locally? *SIAM Journal on Computing*, 24(6):1259–1277, 1995. doi:10.1137/S0097539793254571.
- 16 Alessandro Panconesi and Romeo Rizzi. Some simple distributed algorithms for sparse networks. *Distributed Computing*, 14(2):97–100, 2001. doi:10.1007/PL00008932.
- 17 Alessandro Panconesi and Aravind Srinivasan. The local nature of  $\Delta$ -coloring and its algorithmic applications. *Combinatorica*, 15(2):255–280, 1995. doi:10.1007/BF01200759.
- 18 David Peleg. *Distributed Computing: A Locality-Sensitive Approach*. SIAM Monographs on Discrete Mathematics and Applications. SIAM, Philadelphia, 2000.


# A Population Protocol for Exact Majority with $O(\log^{5/3} n)$ Stabilization Time and $\Theta(\log n)$ States

**Petra Berenbrink**

Universität Hamburg, Hamburg, Germany  
petra.berenbrink@uni-hamburg.de


**Robert Elsässer<sup>1</sup>**

University of Salzburg, Salzburg, Austria  
elsa@cs.sbg.ac.at

 <https://orcid.org/0000-0002-5766-8103>


**Tom Friedetzky**

Durham University, Durham, U.K.  
tom.friedetzky@dur.ac.uk

 <https://orcid.org/0000-0002-1299-5514>


**Dominik Kaaser**

Universität Hamburg, Hamburg, Germany  
dominik.kaaser@uni-hamburg.de

 <https://orcid.org/0000-0002-2083-7145>


**Peter Kling**

Universität Hamburg, Hamburg, Germany  
peter.kling@uni-hamburg.de

 <https://orcid.org/0000-0003-0000-8689>

**Tomasz Radzik<sup>2</sup>**

King's College London, London, U.K.  
tomasz.radzik@kcl.ac.uk

 <https://orcid.org/0000-0002-7776-5461>

---

## Abstract

A population protocol is a sequence of pairwise interactions of  $n$  agents. During one interaction, two randomly selected agents update their states by applying a deterministic transition function. The goal is to stabilize the system at a desired output property. The main performance objectives in designing such protocols are small number of states per agent and fast stabilization time.

We present a fast population protocol for the exact-majority problem, which uses  $\Theta(\log n)$  states (per agent) and stabilizes in  $O(\log^{5/3} n)$  parallel time (i.e., in  $O(n \log^{5/3} n)$  interactions) in expectation and with high probability. Alistarh et al. [SODA 2018] showed that exact-majority protocols which stabilize in expected  $O(n^{1-\Omega(1)})$  parallel time and have the properties of monotonicity and output dominance require  $\Omega(\log n)$  states. Note that the properties mentioned above are satisfied by all known population protocols for exact majority, including ours. They also showed an  $O(\log^2 n)$ -time exact-majority protocol with  $O(\log n)$  states, which, prior to our work, was the fastest exact-majority protocol with polylogarithmic number of states. The standard design framework for majority protocols is based on  $O(\log n)$  phases and requires that all

---

<sup>1</sup> Robert Elsässer's work has been supported by grant no. P 27613 of the Austrian Science Fund (FWF), "Distributed Voting in Large Networks".

<sup>2</sup> Tomasz Radzik's work has been supported by EPSRC grant EP/M005038/1, "Randomized algorithms for computer networks".



agents are well synchronized within each phase, leading naturally to upper bounds of the order of  $\log^2 n$  because of  $\Theta(\log n)$  synchronization time per phase. We show how this framework can be tightened with *weak synchronization* to break the  $O(\log^2 n)$  upper bound of previous protocols.

**2012 ACM Subject Classification** Theory of computation → Distributed computing models

**Keywords and phrases** Population Protocols, Randomized Algorithms, Majority

**Digital Object Identifier** 10.4230/LIPIcs.DISC.2018.10

**Related Version** See [9], <https://arxiv.org/abs/1805.05157>, for the full version of this article.

## 1 Introduction

We consider population protocols [4] for exact-majority voting. The underlying computation system consists of a population of  $n$  anonymous (i.e., identical) *agents*, or *nodes*, and a *scheduler* which keeps selecting pairs of nodes for interaction. A *population protocol* specifies how two nodes update their states when they interact. The computation is a (perpetual) sequence of interactions between pairs of nodes. The objective is for the whole system to eventually stabilize in configurations which have the output property defined by the considered problem. In the general case, the nodes can be connected according to a specified graph  $G = (V, E)$  and two nodes can interact only if they are joined by an edge. Following the scenario considered in most previous work on population protocols, we assume the complete communication graph and the random uniform scheduler. That is, each pair of (distinct) nodes has equal probability to be selected for interaction in any step and each selection is independent of the previous interactions.

The model of population protocols was proposed in Angluin et al. [4] and has subsequently been extensively studied to establish its computational power and to design efficient solutions for fundamental tasks in distributed computing such as various types of *consensus-reaching voting*. The survey from Aspnes and Ruppert [6] includes examples of population protocols, early computational results, and variants of the model. The main design objectives for population protocols are small number of states and fast stabilization time. The original definition of the model assumes that the agents are copies of the same finite-state automaton, so the number of states (per node) is constant. This requirement has later been relaxed by allowing the number of states to increase (slowly) with the population size, to study trade-offs between the memory requirements and the running times.

The (two-opinion) *exact-majority voting* is one of the basic settings of consensus voting [3, 4, 5]. Initially each node is in one of two distinct states  $q_A$  and  $q_B$ , which represent two distinct opinions (or votes)  $A$  and  $B$ , with  $a_0$  nodes holding opinion  $A$  (starting in the state  $q_A$ ) and  $b_0$  nodes holding opinion  $B$ . We assume that  $a_0 \neq b_0$  and denote the initial imbalance between the two opinions by  $\epsilon = |a_0 - b_0|/n \geq 1/n$ . The desired output property is that all nodes have the opinion of the initial majority. An *exact* majority protocol should guarantee that the correct answer is reached, even if the difference between  $a_0$  and  $b_0$  is only 1 (cf. [3]). In contrast, *approximate* majority would require correct answer only if the initial imbalance is sufficiently large. In this paper, when we refer to “majority” protocol/voting/problem we always mean the exact-majority notion.

**Formal Model.** We will now give further formalization of a population protocol and its time complexity. Let  $S$  denote the set of states, which can grow with the size  $n$  of the population (but keeping it low remains one of our objectives). A *configuration* of the system is an assignment of states to nodes. Let  $q(v, t) \in S$  denote the state of a node  $v \in V$  at step  $t$  (that is, after  $t$  individual interactions);  $(v, q(v, t))_{v \in V}$  is the configuration of the system at this step. Two interacting nodes change their states according to a common deterministic transition function  $\delta: S \times S \rightarrow S \times S$ . A population protocol also has an *output function*  $\gamma: S \rightarrow \Gamma$ , which is used to specify the desired output property of the computation. For majority voting,  $\gamma: S \rightarrow \{A, B\}$ , which means that a node in a state  $q \in S$  assumes that  $\gamma(q)$  is the majority opinion. The system is in an (output) correct configuration at a step  $t$  if for each  $v \in V$ ,  $\gamma(q(v, t))$  is the initial majority opinion. We consider undirected individual communications, that is, the two interacting nodes are not designated as initiator and responder, so the transition functions must be symmetric. Thus if  $\delta(q', q'') = (r', r'')$ , then  $\delta(q'', q') = (r'', r')$ , implying, for example, that  $\delta(q, q) = (r, r)$ .

We say that the system is in a *stable configuration* if no node will ever again change its output in any configuration that can be reached. The computation continues (since it is perpetual) and nodes may continue updating their states, but if a node changes from a state  $q$  to another state  $q'$  then  $\gamma(q') = \gamma(q)$ . Thus a majority protocol is in a correct stable configuration if all nodes output the correct majority opinion and will do so in all possible subsequent configurations. Two main types of output guarantee categorize population protocols as either *always correct*, if they reach the correct stable configuration with probability 1, or *w.h.p. correct*. A protocol of the latter type reaches a correct stable configuration *w.h.p.*<sup>3</sup>, allowing that with some low but positive probability an incorrect stable configuration is reached or the computation does not stabilize at all.

The notion of time complexity of population protocols which has lately been used to derive lower bounds on the number of states [1, 2], and the notion which we use also in this paper, is the *stabilization time*  $T_S$  defined as the first *round* when the system enters a correct stable configuration<sup>4</sup>. We follow the common convention of defining the *parallel time* as the number of interactions divided by  $n$ . Equivalently, we group the interactions in rounds of length  $n$ , called also (*parallel*) *steps*, and take the number of rounds as the measure of time. In our analysis we also use the term *period*, which we define as a sequence of  $n$  consecutive interactions, but not necessarily aligned with rounds.

The main result of this paper is a majority protocol with stabilization time  $O(\log^{5/3} n)$  *w.h.p.* and in expectation while using logarithmically many states. According to [2] this number of states is asymptotically optimal for protocols with  $\mathbf{E}(T_S) = O(n^{1-\epsilon})$ , and to the best of our knowledge this is the first result that offers stabilization in time  $O(\log^{2-\Omega(1)} n)$  with poly-logarithmic state space.

**Related Literature.** Draief and Vojnović [12] and Mertzios et al. [17] analyzed two similar four-state majority protocols. Both protocols are based on the idea that the two opinions have *weak* versions  $a$  and  $b$  in addition to the main *strong* versions  $A$  and  $B$ . The strong opinions are viewed as tokens moving around the graph. Initially each node  $v$  has a strong opinion  $A$  or  $B$ , and during the computation it has always one of the opinions  $a, b, A$  or  $B$  (so

<sup>3</sup> A property  $P(n)$ , e.g. that a given protocol reaches a stable correct configuration, holds *w.h.p.* (with high probability), if it holds with probability at least  $1 - n^{-\alpha}$ , where constant  $\alpha > 0$  can be made arbitrarily large by changing the constant parameters in  $P(n)$  (e.g. the constant parameters of a protocol).

<sup>4</sup> Some previous papers (e.g. [1, 11]) refer to this stabilization time as the convergence time.

is in one of these four states). Two interacting opposite strong opinions cancel each other and change into weak opinions. Such pairwise canceling ensures that the difference between the numbers of strong opinions  $A$  and  $B$  does not change throughout the computation (remaining equal to  $a_0 - b_0$ ) and eventually all strong opinions of the initial minority are canceled out. The surviving strong opinions keep moving around the graph, converting the weak opposite opinions. Eventually every node has the opinion (strong or weak) of the initial majority.

Mertzios et al. [17] called their protocol the *4-state ambassador protocol* (the strong opinions are ambassadors) and proved the expected stabilization time  $O(n^5)$  for any graph and  $O((n \log n)/|a_0 - b_0|)$  for the complete graph. Draief and Vojnović [12] called their 4-state protocol the *binary interval consensus*, viewing it as a special case of the *interval consensus* protocol of Bénézit et al. [7], and analyzed it in the continuous-time model. For the uniform edge rates (the continuous setting roughly equivalent to our setting of one random interaction per one time unit) they showed that the expected stabilization time for the complete graph is at most  $2n(\log n + 1)/|a_0 - b_0|$ . They also derived bounds on the expected stabilization time for cycles, stars and Erdős-Rényi graphs.

The appealing aspect of the four-state majority protocols is their simplicity and the constant-size local memory, but the downside is polynomially slow stabilization if the initial imbalance is small. The stabilization time decreases if the initial imbalance increases, so the performance would be improved if there were a way of boosting the initial imbalance. Alistarh et al. [3] achieved such boosting by multiplying all initial strong opinions by the integer parameter  $r \geq 2$ . The nodes keep the count of the number of strong opinions they currently hold. When eventually all strong opinions of the initial minority are canceled,  $|a_0 - b_0|r$  strong opinions of the initial majority remain in the system. This speeds up both the canceling of strong opinions and the converting of weak opinions of the initial minority, but the price is the increased number of states. Refining this idea, Alistarh et al. [1] obtained a majority protocol which has stabilization time  $O(\log^3 n)$  *w.h.p.* and in expectation and uses  $O(\log^2 n)$  states.

A suite of constant-state polylogarithmic-time population protocols for various functions, including exact majority, was proposed by Angluin et al. [5]. Their protocols are *w.h.p.* correct and, more significantly, require a unique leader to synchronize the progress of the computation. Their majority protocol *w.h.p.* reaches a correct stable configuration within  $O(\log^2 n)$  time (with the remaining low probability, it either needs more time to reach the correct output or it stabilizes with an incorrect output) and requires only a constant number of states, but the presence of the leader node is crucial.

The protocols developed in [5] introduced the idea of alternating *cancellations* and *duplications*, an idea that has been frequently used in subsequent majority protocols and also forms the basis of our new protocol. This idea has the following interpretation within the framework of canceling strong opinions. The canceling stops when it is guaranteed that *w.h.p.* the number of remaining strong opinions is less than  $\delta n$ , for some small constant  $\delta < 1/2$ . Now each remaining strong opinion duplicates (once): if a node with a strong opinion interacts with a node which does not hold a strong opinion then both nodes adopt the same strong opinion. This process of duplicating opinions lasts long enough to guarantee, again *w.h.p.*, that all strong opinions have been duplicated. One phase of (partial) cancellations followed by (complete) duplications takes *w.h.p.*  $O(\log n)$  time, and  $O(\log n)$  repetitions of this phase increases the difference between the numbers of strong opinions  $A$  and  $B$  to  $\Theta(n)$ . With such large imbalance between strong opinions, *w.h.p.* within additional  $O(\log n)$  time the minority opinion is completely eliminated and the majority opinion is propagated to all nodes.

Bilke et al. [11] showed that the cancellation-duplication framework from [5] can be implemented without a leader if the agents have enough states to count their interactions. They obtained a majority protocol which has stabilization time  $O(\log^2 n)$  *w.h.p.* and in expectation, and uses  $O(\log^2 n)$  states. Berenbrink et al. [10] generalized the previous results on majority protocols by working with  $k \geq 2$  opinions (*plurality voting*) and arbitrary graphs. Their protocol is based on the methodology introduced earlier for load balancing [18] and achieves  $O(\text{polylog } n)$  time using a polynomial number of states and assuming that the initial advantage of the most common opinion is  $\Omega(n/\text{polylog } n)$ . For the case of complete graphs and  $k = 2$ , their protocol runs *w.h.p.* in  $O(\log n)$  time.

Recently Alistarh et al. [2] showed that any majority protocol which has expected stabilization time of  $O(n^{1-\epsilon})$ , where  $\epsilon$  is any positive constant, and satisfies the conditions of monotonicity and output dominance<sup>5</sup>, requires  $\Omega(\log n)$  states. They also presented a protocol which uses only  $\Theta(\log n)$  states and has stabilization time  $O(\log^2 n)$  *w.h.p.* and in expectation. Their lower and upper bounds raised the following questions. Can exact majority be computed in poly-logarithmic time with  $o(\log n)$  states, if the time is measured in some natural and relevant way other than time until (correct) stabilization? Can exact majority be computed in  $o(\log^2 n)$  time with poly-logarithmically many states? (The protocol in [2] and all earlier exact majority protocols which use poly-logarithmically many states have time complexity at least of the order of  $\log^2 n$ .) Regarding the first question, one may consider the *convergence time* instead of the stabilization time. For a random (infinite) sequence  $\omega$  of interaction pairs, let  $T_C = T_C(\omega)$  denote the convergence time, defined as the first round when (at some interaction during this round) the system enters a correct configuration (all nodes correctly output the majority opinion) and remains in correct configurations in all subsequent interactions (of this sequence  $\omega$ ). Clearly  $T_C \leq T_S$ , since reaching a correct stable configuration implies that whatever the future interactions may be, the system will always remain in correct configurations.

Very recently Kosowski and Uznański [16] and Berenbrink et al. [8] have shown that the convergence time  $T_C$  can be poly-logarithmic while using  $o(\log n)$  states. In [16] the authors design a programming framework and accompanying compilation schemes that provide a simple way of achieving protocols (including majority) which are *w.h.p.* correct, use  $O(1)$  states and converge in expected poly-logarithmic time. They can make their protocols always-correct at the expense of having to use  $O(\log \log n)$  states per node, while keeping poly-logarithmic time, or increasing time to  $O(n^\epsilon)$ , while keeping a constant bound on the number of states. In [8] the authors show an always-correct majority protocol which converges *w.h.p.* in  $O(\log^2 n/\log s)$  time and uses  $\Theta(s + \log \log n)$  states and an always-correct majority protocol which stabilizes *w.h.p.* in  $O(\log^2 n/\log s)$  time and uses  $O(s \cdot \log n/\log s)$  states, where parameter  $s \in [2, n]$ .

The recent population protocols for majority voting often use similar technical tools (mainly efficient constructions of *phase clocks*) as protocols for another fundamental problem of *leader election*. There are, however, notable differences in computational difficulty of both problems, so advances in one problem do not readily imply progress with the other problem. For example, leader election admits always-correct protocols with poly-logarithmically fast stabilization and  $\Theta(\log \log n)$  states [13] (the lower bound here is only  $\Omega(\log \log n)$  [1]). Gasieniec and Stachowiak [14] have recently shown that leader election can be completed in

---

<sup>5</sup> Informally, the running time of a monotonic protocol does not increase if executed with a smaller number of agents. The output dominance means that if the positive counts of states in a stable configuration are changed, then the protocol will stabilize to the same output.



## 10:6 A Population Protocol for Exact Majority

expected time asymptotically significantly better than  $\log^2 n$ , but the best known time-bound for *w.h.p.*-correctness is  $O(\log^2 n)$ . The ideas in [14], however, are specific to leader election and we do not see how they could be applied to improve expected time of majority voting.

**Our Contributions.** We present a majority population protocol with stabilization time  $O(\log^{5/3} n)$  *w.h.p.* and in expectation and  $O(\log n)$  states. Since our protocol satisfies the conditions of monotonicity and output dominance, in view of the lower bound shown in [2], this implies the  $O(\log n)$  number of states being asymptotically optimal for this type of protocols. The main contribution of our protocol is that no majority protocol with  $O(\text{polylog } n)$  states and running time  $O(\log^{2-\alpha} n)$ , for any constant  $\alpha > 0$ , has been known before, not even if the weaker notions of the “convergence time” or “*w.h.p.* correctness” were considered.

All known fast majority protocols with a poly-logarithmic number of states are based in some way on the idea (introduced in [5]) of a sequence of  $\Omega(\log n)$  canceling-doubling phases. Each phase has length  $\Omega(\log n)$  and the nodes are synchronized when they proceed from phase to phase. Our new protocol still uses the overall canceling-doubling framework (as explained in Section 2) but with *shorter phases* of length  $\Theta(\log^{2/3} n)$  each, at the expense of weakening synchronization. We note that all existing majority protocols known to us cease to function properly with sub-logarithmic phases. Such phases are too short to synchronize nodes, resulting in there being, at the same time, nodes from different phases, and the computation potentially getting stuck (opposite opinions from different phases cannot cancel each other or we lose correctness). Moreover, we do not even have the guarantee that every node will be activated at all during a short phase – in fact, we know some nodes will not. The existing protocols require each node to be activated at least logarithmically many times during each phase.

Our main technical contributions are mechanisms to deal with the nodes which advance too slowly or too quickly through the short phases, that is, the nodes which are not in sync with the bulk. In a nutshell, we group  $\log^{1/3} n$  phases in one *epoch*, show that the configuration of the system remains reasonably tidy throughout one epoch even without explicit synchronization, and introduce “cleaning-up” and synchronization at the boundaries between epochs. We believe that some of our algorithmic and analytical ideas developed for fast majority voting may be of independent interest.

**Outline.** The remainder of the paper is organized as follows. We first, in Section 2, describe the  $O(\log^2 n)$ -time,  $O(\log^2 n)$ -state **Majority** protocol presented in [11], which we use as the baseline implementation of the canceling-doubling framework. We refer to the structure and the main properties of this protocol when describing and analysing our new faster protocols. In Section 3 we present our main protocol **FastMajority1**, which stabilizes in  $O(\log^{5/3} n)$  time and uses  $\Theta(\log^2 n)$  states, and in Section 4 we outline the analysis of this protocol. In Section 5 we outline how to modify protocol **FastMajority1** yielding protocol **FastMajority2**, which has the same  $O(\log^{5/3} n)$  bound on the running time but uses only  $\Theta(\log n)$  states. Further details of our protocols, including pseudocode and detailed proofs, are given in the full version of the paper [9].

### 2 Exact majority: the general idea of canceling-doubling phases

We view the  $A/B$  votes as tokens which can have different values (or magnitudes). Initially each node has one token of type  $A$  or  $B$  with value 1 (the base, or original, value of a token). Throughout the computation, each node either has one token or is *empty*. In the following we say that two tokens *meet* if their corresponding nodes interact.



- When two opposite tokens  $A$  and  $B$  of the same value meet, then they can cancel each other and the nodes become empty. Such an interaction is called *canceling* of tokens.
- When a token of type  $X \in \{A, B\}$  and value  $z$  interacts with an empty node, then this token can split into two tokens of type  $X$  and half the value  $z/2$ , and each of the two involved nodes takes one token. We call such an interaction *splitting, duplicating or doubling* of a token.

We also use the notion of the *age* of a token, the number of times it has undergone splitting. Thus the relation between the value  $z$  and the age  $g$  of a token is  $z = 1/2^g$ . Each node stores only the age of the token it possesses (if any), as its value can easily be computed from its age and vice-versa. Note that any sequence of canceling and splitting interactions preserves the difference between the sum of the values of all  $A$  and  $B$  tokens. This difference is always equal to the initial imbalance. The primary objective is to eliminate all minority tokens. When only majority tokens are left in the system (and this is recognized by at least of the the nodes), the majority opinion can be propagated to all nodes *w.h.p.* within additional  $O(\log n)$  time via a broadcast process. In the broadcast process, if two nodes interact and one of the nodes is in a final state, then the other node adopts the opinion of the first node and switches to the final state as well, except when a *conflict* occurs. In such a case some backup protocol is initiated that guarantees that the process always converges to the correct result. Since a conflict occurs with a small probability only, the running time of the overall protocol is  $O(\log^2 n)$  with high probability and in expectation. The details of this standard process of propagating the outcome will be omitted from our descriptions and analysis. That is, from now on we assume that the objective is to eliminate the minority tokens.

From a node's local point of view, the computation of the  $O(\log^2 n)$ -time,  $O(\log^2 n)$ -state Majority protocol consists of at most  $\log n + 2$  phases and each phase consists of at most  $C \log n$  interactions, where  $C$  is a suitably large constant. Each node keeps count of phases and steps (interactions) within the current phase, and maintains further information which indicates the progress of computation. More precisely, each node  $v$  keeps the following data, which require  $\Theta(\log^2 n)$  states.

- $v.token \in \{A, B, \emptyset\}$  – the type of token held by  $v$ . If  $v.token = \emptyset$  then the node is empty.
- $v.phase \in \{0, 1, 2, \dots, \log n + 2\}$  – the counter of phases.
- $v.phase\_step \in \{0, 1, 2, \dots, (C \log n) - 1\}$  – the counter of steps in the current phase.
- Boolean flags, which are initially false and indicate the following status when set to *true*:
  - $v.doubled$  –  $v$  has a token which has already doubled in the current phase;
  - $v.done$  – the node has made the decision on the final output;
  - $v.fail$  – the protocol has failed because of some inconsistencies.

If a node  $v$  is in neither of the two special states *done* and *fail*, then we say that  $v$  is in a *normal state*:  $v.normal \equiv \neg(v.done \vee v.fail)$ . A node  $v$  is in Phase  $i$  if  $v.phase = i$ . If  $v$  is in Phase  $i$  and is not empty, then the age of the token at  $v$  is either  $i$  if  $\neg v.doubled$  (the token has not doubled yet in this phase) or  $i + 1$  if  $v.doubled$ . Thus the phase of a token (more correctly, the token's host node) and the flag *doubled* indicate the age of this token. Throughout the whole computation, the pair  $(v.phase, v.phase\_step)$  can be regarded as the (combined) interaction counter  $v.time \in \{0, 1, 2, \dots, 2C \log^2 n\}$  of node  $v$ . This counter is incremented by 1 at the end of each interaction. Thus, for example, if  $v.phase\_step$  is equal to 0 after such an increment, then node  $v$  has just completed a phase. Each phase is divided into five parts defined below, where  $c$  is a constant discussed later.

- The beginning, the middle and the final parts of a phase are buffer zones, consisting of  $c \log n$  steps each. The purpose of these parts is to ensure that the nodes progress through the current phase in a synchronized way.

- The second part is the *canceling stage* and the fourth part is the *doubling stage*, each consisting of  $((C - 3c)/2) \log n$  steps. If two interacting nodes are in the canceling stage of the same phase and have opposite tokens then the tokens are canceled. If two interacting nodes are in the doubling stage of the same phase, one of them has a token which has not doubled yet in this phase and the other is empty, then this is a doubling interaction.

If nodes were simply incrementing their step counters by 1 at each interaction, then those counters would start diverging too much for the canceling-doubling process to work correctly. An important aspect of the **Majority** protocol, as well as our new faster protocols, is the following mechanism for keeping the nodes sufficiently synchronized. When two interacting nodes are in different phases then the node in the lower phase jumps up to (that is, sets its step counter to) the beginning of the next phase. The **Majority** protocol relies on this synchronization mechanism in the high-probability case when all nodes are in two adjacent parts of a phase (that is, either in two adjacent parts of the same phase, or in the final part of one phase and the beginning part of the next phase.) In this case the process of pulling all nodes up to the next phase follows the pattern of *broadcast*. The node, or nodes, which have reached the beginning of the next phase by way of normal one-step increments broadcast the message “*if you are not yet in my phase then proceed to the next phase.*” By the time the broadcast is completed (that is, by the time when the message has reached all nodes), all nodes are together in the next phase. It can be shown that there is a constant  $\beta_0$  such that *w.h.p.* the broadcast completes in  $\beta_0 n \log n$  random pairwise interactions (see, for example [5]; other papers may refer to this process as *epidemic spreading* or *rumor spreading*).

The constant  $c$  in the definition of the parts of a phase is suitably smaller than the constant  $C$ , but sufficiently large to guarantee the following two conditions: (a) the *broadcast* from a given node to all other nodes completes *w.h.p.* within  $(c/5)n \log n$  interactions; and (b) for a sequence of  $(C/2)n \log n$  consecutive interactions, *w.h.p.* for each node  $v$  and each  $0 < t \leq (C/2)n \log n$ , the number of times  $v$  is selected for interaction within the first  $t$  interactions differs from the expectation  $2t/n$  by at most  $(c/5) \log n$ . Condition (a) is used when the nodes reaching the end of the current phase  $i$  initiate broadcast to “pull up” the nodes lagging behind. Condition (a) implies that after  $(c/5)n \log n$  interactions, *w.h.p.* all nodes are in the next phase. Using Condition (b) with  $t = (c/5)n \log n$ , we can also claim that *w.h.p.* at this point all nodes are within the first  $(3/5)c \log n$  steps of the next phase (all nodes are in the next phase and no node interacted more than the expected  $(2/5)c \log n$  plus  $(1/5)c \log n$  times). Finally Condition (b) applied to all  $(c/5)n \log n \leq t \leq (C/2)n \log n$  implies that *w.h.p.* the differences between the individual counts of node interactions do not diverge by more than  $c \log n$  throughout this phase. We set  $c = C^{3/4}$  and take  $C$  large enough so that  $c \leq C/9$  (to have at least  $3c \log n$  steps in the canceling and doubling stages) and both Conditions (a) and (b) hold. This way we achieve the following synchronized progress of nodes through a phase: *w.h.p.* all nodes are in the same part of the same phase before they start moving on to the next part. Moreover, also *w.h.p.*, for each canceling or doubling stage there is a sequence of  $\Theta(n \log n)$  consecutive interactions when all nodes remain in this stage and each one of them is involved in at least  $c \log n$  interactions.

Thus throughout the computation of the **Majority** protocol, *w.h.p.* all nodes are in two adjacent parts of a phase. In particular, *w.h.p.* the canceling and doubling activities of the nodes are separated. This separation ensures that the cancellation of tokens creates a sufficient number of empty nodes to accommodate new tokens generated by token splitting in the subsequent doubling stage. If two interacting nodes are not in the same or adjacent parts of a phase (a low, but positive, probability), then their local times (step counters) are considered inconsistent and both nodes enter the special *fail* state. The details of the **Majority** protocol are given in pseudocode in the full version [9].

From a global point of view, *w.h.p.* each new phase  $p$  starts with all nodes in normal states in the beginning of this phase. We say that this phase completes successfully if all nodes are in normal states in the beginning part of the next phase  $p + 1$ . At this point all tokens have the same value  $1/2^{p+1}$ , and the difference between the numbers of opposite tokens is equal to  $2^{p+1}|a_0 - b_0|$ . The computation *w.h.p.* keeps successfully completing consecutive phases, each phase halving the value of tokens and doubling the difference between  $A$  tokens and  $B$  tokens, until the *critical phase*  $p_c$ , which is the first phase  $0 \leq p_c \leq \log n - 1$  when the difference between the numbers of opposite tokens is

$$2^{p_c}|a_0 - b_0| > n/3. \quad (1)$$

The significance of the critical phase is that the large difference between the numbers of opposite tokens means that *w.h.p.* all minority tokens will be eliminated in this phase, if they have not been eliminated yet in previous phases. More specifically, at the end of phase  $p_c$ , *w.h.p.* only tokens of the majority opinion are left and each of these tokens has value either  $1/2^{p_c+1}$  if the token has split in this phase, or  $1/2^{p_c}$  otherwise. If at least one token has value  $1/2^{p_c}$ , then this token has failed to double during this phase and assumes that the computation has completed. Such a node enters the *done* state and broadcasts its (majority) opinion to all other nodes. In this case phase  $p_c$  is the *final phase*.

If at the end of the critical phase all tokens have value  $1/2^{p_c+1}$  then no node knows yet that all minority tokens have been eliminated, so the computation proceeds to the next phase  $p_c + 1$ . Phase  $p_c + 1$  will be the final phase, since it will start with more than  $(2/3)n$  tokens and all of them of the same type, so at least one token will fail to double and will assume that the computation has completed and will enter the *done* state. The failure to double is taken as an indication that *w.h.p.* all tokens of opposite type have been eliminated. Some tokens may still double in the final phase and enter the next phase (later receiving the message that the computation has completed) but *w.h.p.* no node reaches the end of phase  $p_c + 2 \leq \log n + 1$ . Thus the *done* state is reached *w.h.p.* within  $O(\log^2 n)$  parallel time.

The computation may fail *w.l.p.*<sup>6</sup> when the step counters of two interacting nodes are not consistent, or a node reaches phase  $\log n + 2$ , or two nodes enter the *done* state with opposite-type tokens. Whenever a node realizes that any of these low-probability events has occurred, it enters the *fail* state and broadcasts this state.

It is shown in [11] that the **Majority** protocol stabilizes, either in the correct all-*done* configuration or in the all-*fail* configuration, within  $O(\log^2 n)$  time *w.h.p.* and in expectation. The standard technique of combining a fast protocol, which *w.l.p.* may fail, with a slow always-correct backup protocol gives an *extended Majority* protocol, which requires  $\Theta(\log^2 n)$  states per node and computes the exact majority within  $O(\log^2 n)$  time *w.h.p.* and in expectation. The idea is to run both the fast and the slow protocols in parallel and make the nodes in the *fail* state adopt the outcome of the slow protocol. The slow protocol runs in expected polynomial, say  $O(n^\alpha)$ , time, but its outcome is used only with low probability of  $O(n^{-\alpha})$ , so it contributes only  $O(1)$  to the overall expected time.

We omit the details of using a slow backup protocol (see, for example, [2, 11]), and assume that the objective of a canceling-doubling protocol is to use a small number of states  $s$ , to compute the majority quickly *w.h.p.*, say within a time bound  $T'(n)$ , and to also have low expected time of reaching the correct all-*done* configuration or the all-*fail* configuration, say within a bound  $T''(n)$ . If the bounds  $T'(n)$  and  $T''(n)$  are of the same order  $O(T(n))$ , then we get a corollary that the majority can be computed with  $O(s)$  states in  $O(T(n))$  time *w.h.p.* and in expectation.

<sup>6</sup> *w.l.p.* – with low probability – means that the opposite event happens *w.h.p.*

### 3 Exact majority in $O(\log^{5/3} n)$ time with $\Theta(\log^2 n)$ states

To improve on the  $O(\log^2 n)$  time of the Majority protocol, we reduce the length of a phase to  $\Theta(\log^{1-a} n)$ , where  $a = 1/3$ . The new **FastMajority1** protocol runs in  $O(\log^{1-a} n) \times O(\log n) = O(\log^{5/3} n)$  time and requires  $\Theta(\log^2 n)$  states per node. We will show in Section 5 that the number of states can be reduced to asymptotically optimal  $\Theta(\log n)$ . We keep the term  $a$  in the description and the analysis of our fast majority protocols to simplify notation and to make it easier to trace where a larger value of  $a$  would break the proofs.

Phases of sub-logarithmic length are too short to ensure that *w.h.p.* all tokens progress through the phases synchronously and keep up with required canceling and doubling, as they did in the Majority protocol. In the **FastMajority1** protocol, we have a small but *w.h.p.* positive number of *out-of-sync* tokens, which move to the next phase either too early or too late (with respect to the expectation) or simply do not succeed with splitting within a short phase. Such tokens stop contributing to the regular dynamics of canceling and doubling. The general idea of our solution is to group  $\log^a n$  consecutive phases (a total of  $\Theta(\log n)$  steps) into an *epoch*, to attach further  $\Theta(\log n)$  steps at the end of each epoch to enable the out-of-sync tokens to reach the age required at the end of this epoch, and to synchronize all nodes by the broadcast process at the boundaries of epochs. When analyzing the progress of tokens through the phases of the same epoch, we consider the tokens which remain synchronized and the out-of-sync tokens separately.

We now proceed to the details of the **FastMajority1** protocol. Each epoch consists of  $2C \log n$  steps, where  $C$  is a suitably large constant, and is divided into two equal-length parts. The first part is a sequence of  $\log^a n$  canceling-doubling phases, each of length  $C \log^{1-a} n$ . The purpose of the second part is to give sufficient time to out-of-sync tokens so that *w.h.p.* they all complete all splitting required for this epoch. Each node  $v$  maintains the following data, which can be stored using  $\Theta(\log^2 n)$  states. For simplicity of notation, we assume that expressions like  $\log^a n$  and  $C \log^{1-a} n$  have integer values if they refer to an index (or a number) of phases or steps.

- $v.token \in \{A, B, \emptyset\}$  – type of token held by  $v$ .
- $v.epoch \in \{0, 1, \dots, \log^{1-a} n + 2\}$  – the counter of epochs.
- $v.age\_in\_epoch \in \{0, 1, \dots, \log^a n\}$  – the age of the token at  $v$  (if  $v$  has a token) with respect to the beginning of the current epoch. If  $v.token$  is  $A$  or  $B$ , then the age of this token is  $g = v.epoch \cdot \log^a n + v.age\_in\_epoch$  and the value of this token is  $1/2^g$ .
- $v.epoch\_part \in \{0, 1\}$  – each epoch consists of two parts, each part has  $C \log n$  steps. The first part, when  $v.epoch\_part = 0$ , is divided into  $\log^a n$  canceling-doubling phases.
- $v.phase \in \{0, 1, \dots, (\log^a n) - 1\}$  – counter of phases in the first part of the current epoch.
- $v.phase\_step \in \{0, 1, \dots, (C \log^{1-a} n) - 1\}$  – counter of steps (interactions) in the current phase.
- Boolean flags indicating the status of the node, all set initially to *false*:
  - $v.doubled, v.done, v.fail$  – as in the Majority protocol;
  - $v.out\_of\_sync$  –  $v$  has a token which no longer follows the expected progress through the phases of the current epoch;
  - $v.additional\_epoch$  – the computation is in the additional epoch of  $3 \log^a n$  phases, with each of these phases consisting now of  $\Theta(\log n)$  steps.

We say that a node  $v$  is in epoch  $j$  if  $v.epoch = j$ , and in phase  $i$  (of the current epoch) if  $v.phase = i$ . We view the triplet  $(v.epoch\_part, v.phase, v.phase\_step)$  as the (combined) counter  $v.epoch\_step \in \{0, 1, 2, \dots, (2C \log n) - 1\}$  of steps in the current epoch,

and the pair  $(v.\text{epoch}, v.\text{epoch\_step})$  as the counter  $v.\text{time} \in \{0, 1, 2, \dots, (2C \log^{2-a} n) + O(\log n)\}$  of the steps of the whole protocol. If a node  $v$  is not in any of the special states  $\text{out\_of\_sync}$ ,  $\text{additional\_epoch}$ ,  $\text{done}$  or  $\text{fail}$ , then we say that  $v$  is in a *normal state*:

$$v.\text{normal} \equiv \neg(v.\text{out\_of\_sync} \vee v.\text{additional\_epoch} \vee v.\text{done} \vee v.\text{fail}).$$

A normal token is a token hosted by a normal node. Each phase is split evenly into the canceling stage (the first  $(C/2) \log^{1-a} n$  steps of the phase) and the doubling stage (the remaining  $(C/2) \log^{1-a} n$  steps).

The vast majority of the tokens are normal tokens progressing through the phases of the current epoch in a synchronized fashion. These tokens are simultaneously in the beginning part of the same phase  $j$  and have the same age  $j$  (w.r.t. the end of the epoch). They first try to cancel with tokens of the same age but opposite type during the canceling stage, and if they survive, they then split during the subsequent doubling stage. At some later time most of the tokens will still be normal, but in the beginning part of the next phase  $j + 1$  and having age  $j + 1$ . Thus the age of a normal token (w.r.t. the beginning of the current epoch) is equal to its phase if the token has not yet split in this phase (this is recorded by setting the flag *doubled*), or to its phase plus 1 otherwise.

As in the **Majority** protocol, we separate the canceling and the doubling activities to ensure that the canceling of tokens first creates a sufficient number of empty nodes to accommodate the new tokens obtained later from splitting. Unlike in the **Majority** protocol, the **FastMajority1** protocol does not have the buffer zones within a phase. Such zones would not be helpful in the context of shorter sublogarithmic phases when anyway we cannot guarantee that *w.h.p.* all nodes progress through a phase in a synchronously.

A token which has failed to split in one of the phases of the current epoch becomes an out-of-sync token (the *out\_of\_sync* flag is set). Such a token no longer follows the regular canceling-doubling phases of the epoch, but instead tries cascading splitting to break up into tokens of age  $\log^a n$  (relative to the beginning of the epoch) as expected by the end of this epoch. An out-of-sync token does not attempt canceling because there would be only relatively few opposite tokens of the same value, so only a small chance to meet them (too small to make a difference in the analysis). The tokens obtained from splitting out-of-sync tokens inherit the out-of-sync status. A token drops the out-of-sync status if it is in the second part of the epoch and has reached the age  $\log^a n$ . (Alternatively, out-of-sync tokens could switch back to the normal status as soon as their age coincides again with their phase, but this would complicate the analysis.) An *out-of-sync node* is a node hosting an out-of-sync token. While each normal node and token is in a specific phase of the first part of an epoch or is in the second part of an epoch, the out-of-sync nodes (tokens) belong to an epoch but not to any specific phase. The objective for a normal token is to split into two halves in each phase of the current epoch (if it survives canceling). The objective of an out-of-sync token is to keep splitting in the current epoch (disregarding the boundaries of phases) until it breaks into tokens as expected by the end of this epoch.

In our analysis we show that *w.h.p.* there are only  $O(n/2^{\Theta(\log^a n)})$  out-of-sync tokens in any one epoch. *W.h.p.* all out-of-sync tokens in the current epoch reach the age  $\log^a n$  (w.r.t. the beginning of the epoch) by the midpoint of the second part of the epoch (that is, by the step  $(3/2)C \log n$  of the epoch), for each but the *final epoch*  $j_f$ . In the final epoch at least one out-of-sync token completes the epoch without reaching the required age.

When the system completes the final epoch, the task of determining the majority opinion is not fully achieved yet. In contrast to the **Majority** protocol where on completion of the final phase *w.h.p.* only majority tokens are left, in the **FastMajority1** protocol there may

still be a small number of minority tokens at the end of the final epoch, so some further work is needed. A node which has failed to reach the required age by the end of the current epoch, identifying that way that this is the final epoch, enters the *additional\_epoch* state and broadcasts this state through the system to trigger an *additional epoch* of  $\Theta(\log^a n)$  phases. More precisely, the additional epoch consists of at most  $3 \log^a n$  phases corresponding to epochs  $j_f - 1$  (if  $j_f > 0$ ),  $j_f$  and  $j_f + 1$ , each phase now having  $\Theta(\log n)$  steps. *W.h.p.* these phases include the critical phase  $p_c$  and the phase  $p_c + 1$ , defined by (1). The computation of the additional epoch is as per the **Majority** protocol, taking  $O(\log^{1+a} n)$  time to reach the correct all-*done* configuration *w.h.p.* or the all-*fail* configuration *w.l.p.*

Two interacting nodes first check the consistency of their *time* counters (the counters of interactions) and switch to *fail* states if the difference between the counters is greater than  $(1/4)C \log n$ . If the counters are consistent but the nodes are in different epochs (so one near the end of an epoch with the other being near the beginning of the next) then the node in the lower epoch jumps up to the beginning of the next epoch. This is the synchronization mechanism at the boundaries of epochs, analogous to the synchronization by broadcast at the boundaries of phases in the **Majority** protocol. In the **FastMajority1** protocol, however, it is not possible to synchronize the nodes at the boundaries of (short) phases.

For details of the **FastMajority1** protocol we refer the reader to the full version [9].

#### 4 Analysis of the **FastMajority1** protocol

Ideally we would like for all tokens to progress through the phases of the current epoch synchronously, *w.h.p.*, that is, all tokens being roughly in the same part of the same phase, as in the **Majority** protocol. This would mean that *w.h.p.* at some (global) time all nodes are in the beginning part of the same phase, ensuring that all tokens have the same value  $x$ , and at some later point all nodes are in the end part of this phase and all surviving tokens have value  $x/2$ . This ideal behavior is achieved by the **Majority** protocol at the cost of having  $\Theta(\log n)$ -step phases. As discussed in Section 2, the logarithmic length of a phase also gives sufficient time to synchronize *w.h.p.* the local times of all nodes at the end of a phase so that they all end up together in the beginning part of the next phase.

Now, with phases having only  $\Theta(\log^{1-a} n)$  steps, we face the following two difficulties in the analysis. Firstly, while a good number of tokens split during such a shorter phase, *w.h.p.* there are also some tokens which do not split. Secondly, phases of length  $o(\log n)$  are too short to keep the local times of the nodes synchronized. We can again show that a good number of nodes proceed in synchronously, but *w.h.p.* there are nodes falling behind or rushing ahead and our analysis has to account for them.

Counting the phases across the epochs, we define the critical phase  $p_c$  as in (1). Similarly as in the  $O(\log^2 n)$ -time **Majority** protocol, the computation proceeds through the phases moving from epoch to epoch until it reaches the critical phase  $p_c$ . The computation will then get stuck in this phase or in the next phase  $p_c + 1$ . Some tokens do not split in that final phase, nor in any subsequent phase of the current epoch, because there are not enough empty nodes to accommodate new tokens. Almost all minority tokens have been eliminated, and so the creation of empty nodes by cancellations of opposite tokens has all but stopped. This is the final epoch  $j_f$  and the nodes which do not split to the value required by the end of this epoch trigger the additional epoch of  $O(\log^a n)$  phases, each having  $\Theta(\log n)$  steps. The additional epoch is needed because we do not have a high-probability guarantee that all minority tokens are eliminated by the end of the final epoch. The small number of remaining minority tokens may have various values which are inconsistent with the values of



the majority tokens, so further cancellations of tokens might not be possible. The additional epoch includes the phases of the three consecutive epochs  $j_f - 1$ ,  $j_f$  and  $j_f + 1$  to ensure that *w.h.p.* both phases  $p_c$  and  $p_c + 1$  are included. Phase  $p_c$  can be as early as the last phase in epoch  $j_f - 1$  and phase  $p_c + 1$  can be as late as the first phase in epoch  $j_f + 1$ .

The following conditions describe the regular configuration of the whole system at the beginning of epoch  $j$ , and the corresponding Lemma 1 summarizes the progress of the computation through this epoch. Recall that the **FastMajority1** protocol is parameterized by a suitably large constant  $C > 1$  and our analysis refers also to another smaller constant  $c = C^{3/4}$ . We refer to the first (last)  $c \log^{1-a} n$  steps of a phase or a stage as the beginning (end) part of this phase or stage. The (global) time steps count the number of interactions of the whole system.

*EpochInvariant(j)* :

1. At least  $n(1 - 1/2^{3 \log^a n})$  nodes are in normal states, are in epoch  $j$ , and their *epoch\_step* counters are at most  $c \log^a n$ .
2. For each remaining node  $u$ ,
  - a.  $u$  is in a normal state in epoch  $j - 1$  and  $u.\text{epoch\_step} \geq (3/2)C \log n$  (that is,  $u$  is in the last quarter of epoch  $j - 1$ ), or
  - b.  $u$  is in a normal or out-of-sync state in epoch  $j$  and  $u.\text{epoch\_step} \leq 4c \log n$ .

► **Lemma 1.** *Consider an arbitrary epoch  $j \geq 0$  such that phase  $p_c$  belongs to an epoch  $j' \geq j$  and assume that at some (global) step  $t$  the condition *EpochInvariant(j)* holds.*

1. *If phase  $p_c$  does not belong to epoch  $j$  (that is, phase  $p_c$  is in a later epoch  $j' > j$ ), then *w.h.p.* there is a step  $\tilde{t} \leq t + 2Cn \log n$  when the condition *EpochInvariant(j + 1)* holds.*
2. *If both phases  $p_c$  and  $p_c + 1$  belong to epoch  $j$ , then *w.h.p.* there is a step  $\tilde{t} \leq t + 2Cn \log n$  when*
  - (\*) *a node completes epoch  $j$  and enters the additional\_epoch state (because it has a token which has not split to the value required by the end of this epoch); and*  
*all other nodes are in normal or out-of-sync states in the second part of epoch  $j$  or the first part of epoch  $j + 1$ .*
3. *Otherwise, that is, if phase  $p_c$  is the last phase in epoch  $j$  (and  $p_c + 1$  is the first phase in epoch  $j + 1$ ), then *w.h.p.* either there is a step  $\tilde{t} \leq t + 2Cn \log n$  when the above condition (\*) for the end of epoch  $j$  holds, or all nodes eventually proceed to epoch  $j + 1$  and there is a step  $\hat{t} \leq t + 3Cn \log n$  when the condition analogous to (\*) but for the end of epoch  $j + 1$  holds.*

The condition *PhaseInvariant1(j, i)* given below describes the regular configuration of the whole system at the beginning of phase  $0 \leq i \leq \log^a n$  in epoch  $j \geq 0$ . We note that the last phase in an epoch is phase  $\log^a n - 1$  and the condition *PhaseInvariant1(j, \log^a n)* refers in fact to the beginning of the second part of the epoch. A normal token in the beginning of phase  $i$  in epoch  $j$  has (absolute) value  $2^{-(j \log^a n + i)}$  and relative values 1, 2,  $1/2^i$  and  $2^{\log^a n - i}$  w.r.t. the beginning of this phase, the end of this phase, the beginning of this epoch and the end of this epoch, respectively. It may also be helpful to recall that for a given node  $v$ , phase  $i$  starts at  $v$ 's epoch step  $Ci \log^{1-a} n$ . Observe that *EpochInvariant(j)* implies *PhaseInvariant1(j, 0)*.

*PhaseInvariant1(j, i)* :

1. The set  $W$  of nodes which are normal and in the beginning part of phase  $i$  in epoch  $j$  has size at least  $n(1 - (i+1)/2^{2 \log^a n})$ . That is, a node  $v$  is in  $W$  if and only if  $v.\text{normal}$  is true,  $v.\text{phase\_step} \leq c \log^{1-a} n$ ,  $v.\text{epoch} = j$ , and either  $v.\text{epoch\_part} = 0$  and  $v.\text{phase} = i$  if  $i < \log^a n$ , or  $v.\text{epoch\_part} = 1$  and  $v.\text{phase} = 0$  if  $i = \log^a n$ .

## 10:14 A Population Protocol for Exact Majority

2. Let  $U = V \setminus W$  denote the set of the remaining nodes.
  - a. For each  $u \in U$ :
    - $u$  is a normal node in epoch  $j - 1$ ,  $u.\text{epoch\_step} \geq (3/2)C \log n$  and  $i < (c/C) \log^a n$ ;
    - or  $u$  is in a normal or out-of-sync state in epoch  $j$  and  $|u.\text{epoch\_step} - Ci \log^{1-a} n| \leq 4c \log n$ .
  - b. The total value of the tokens in  $U$  w.r.t. the end of epoch  $j$  is at most  $n(i+1)/2^{2 \log^a n}$ .

For an epoch  $0 \leq j$  and a phase  $0 \leq i < \log^a n$  in this epoch, let  $p(j, i) = j \log^a n + i$  denote the global index of this phase. We show that *w.h.p.* the condition  $\text{PhaseInvariant1}(j, i)$  holds at the beginning of each phase  $p(j, i) \leq p_c$ .

► **Lemma 2.** *For arbitrary  $0 \leq j$  and  $0 \leq i \leq \log^a n - 1$  such that  $p(j, i) \leq p_c$ , assume that the condition  $\text{EpochInvariant}(j)$  holds at some (global) time step  $t$  and the condition  $\text{PhaseInvariant1}(j, i)$  holds at the step  $t_i = t + i(C/2)n \log^{1-a} n$ . Then the following conditions hold, where  $t_{i+1} = t + (i+1)(C/2)n \log^{1-a} n$ .*

1. *If  $p(j, i) < p_c$ , then *w.h.p.* at step  $t_{i+1}$  the condition  $\text{PhaseInvariant1}(j, i+1)$  holds.*
2. *If  $p(j, i) = p_c$ , then *w.h.p.* at step  $t_{i+1}$  the total value, w.r.t. the end of epoch  $j$ , of the minority-opinion tokens is  $O(n \log n / 2^{2 \log^a n})$ .*

Lemma 2 is proven by analyzing the cancellations and duplications of tokens in one phase. This lemma heavily uses Claim 6, in which it is essential that  $a \leq 1/3$ . Lemma 1 is proven by inductively applying Lemma 2. In turn, Theorem 3 below, which states the  $O(\log^{5/3} n)$  bound on the completion time of the `FastMajority1` protocol, can be proven by inductively applying Lemma 1 and by choosing  $a = 1/3$ .

► **Theorem 3.** *The `FastMajority1` protocol uses  $\Theta(\log^2 n)$  states, computes the majority *w.h.p.* within  $O(\log^{5/3} n)$  time, and reaches the correct all-done configuration or the all-fail configuration within expected  $O(\log^{5/3} n)$  time.*

► **Corollary 4.** *The majority can be computed with  $\Theta(\log^2 n)$  states in  $O(\log^{5/3} n)$  time *w.h.p.* and in expectation.*

We now give some further explanation of the structure of our analysis, referring the reader to the full version [9] for the formal proofs. Lemma 5 and Claim 6 show the synchronization of the nodes which we rely on in our analysis. Lemma 5 is used in the proof of Lemma 1, where we analyze the progress of the computation through one epoch consisting of  $O(n \log n)$  interactions ( $O(\log n)$  parallel steps). Lemma 5 can be easily proven using first Chernoff bounds for a single node and then the union bound over all nodes. The proof of Claim 6 is considerably more involved, but we need this claim in the proof of Lemma 2, where we look at the finer scale of individual phases and have to consider intervals of  $\Theta(\log^{1-a} n)$  interactions of a given node. This claim shows, in essence, that most of the nodes stay tightly synchronized when they move from phase to phase through one epoch. The  $\text{epoch\_step}$  counters of these nodes stay in a range of size at most  $c \log^{1-a} n$ .

► **Lemma 5.** *For any constant  $C$  and for  $c = C^{3/4}$ , during a sequence of  $t \leq 2Cn \log n$  interactions, with probability at least  $1 - n^{-\alpha(C)}$  (for a suitable function  $\alpha = \omega(1)$ ) the number of interactions of each node is within  $c \log n$  of the expectation of  $2t/n$  interactions.*

► **Claim 6.** *For a fixed  $j \geq 0$ , assume that  $\text{EpochInvariant}(j)$  holds at a time step  $t$ . Let  $W \subseteq V$  be the set of  $n(1 - o(1))$  nodes which satisfy at this step the condition 1 of  $\text{EpochInvariant}(j)$  (that is,  $W$  is the set of nodes which are in epoch  $j$  with  $\text{epoch\_step}$  counters at most  $c \log^a n$ ). Then at an arbitrary but fixed time step  $t < t' \leq t + (3/4)Cn \log n$ ,*



*w.h.p.* all nodes in  $W$  are in epoch  $j$  and all but  $O(n/2^{6 \log^a n})$  of them have their `epoch_step` counters within  $c/2 \cdot \log^{1-a} n$  from  $2(t' - t)/n$ .

Note that this claim only holds if  $a \leq 1/3$ , otherwise one can not guarantee that *w.h.p.* all but  $O(n/2^{6 \log^a n})$  of the nodes in  $W$  have their `epoch_step` counters within  $c/2 \cdot \log^{1-a} n$  of  $2(t' - t)/n$ .

Lemmas 7 and 8 describe the performance of the broadcast process in the population-protocol model. Lemma 7 has been used before and is proven, for example, in [11]. Lemma 8 is a more detailed view at the dynamics of the broadcast process, which we need in the context of Lemma 1 to show that the synchronization at the end epoch  $j$  gives *w.h.p.* `EpochInvariant`( $j + 1$ ).

► **Lemma 7.** *For any constant  $c$ , the broadcast completes with probability at least  $1 - n^{-\alpha(c)}$  (for a suitable function  $\alpha = \omega(1)$ ) within  $cn \log n$  interactions.*

► **Lemma 8.** *Let  $b \in (0, 1)$  and  $c > 0$  be arbitrary constants and let  $c_1$  be a sufficiently large constant. Consider the broadcast process and let  $t_1$  be the first step when  $n/2^{6 \log^b n}$  nodes are already informed and  $t_2 = t_1 + c_1 n \log^b n$ . Then the following conditions hold.*

1. *With probability at least  $1 - n^{-\omega(1)}$ ,  $n - O(n/2^{6 \log^b n})$  nodes receive the message for the first time within the  $c_1 n \log^b n$  consecutive interactions  $\{t_1 + 1, t_1 + 2, \dots, t_2\}$ .*
2. *With probability at least  $1 - n^{-\alpha(c)}$  (for a suitable function  $\alpha = \omega(1)$ ),  $t_1 \leq cn \log n$  and each node interacts at most  $4c \log n$  times in interval  $[1, t_2]$ .*
3. *With probability at least  $1 - n^{-\omega(1)}$ , there are  $n - O(n/2^{6 \log^b n})$  nodes which interact within interval  $[t_1 + 1, t_2]$  at least  $c_1 \log^b n$  times but not more than  $3c_1 \log^b n$  times.*

## 5 Reducing the number of states to $\Theta(\log n)$

Our `FastMajority1` protocol described in Section 3 requires  $\Theta(\log^2 n)$  states per node. Using the idea underlying the constructions of *leaderless phase clocks* in [15] and [2], we now modify `FastMajority1` into the protocol `FastMajority2`, which still works in  $O(\log^{5/3} n)$  time but has only the asymptotically optimal  $\Theta(\log n)$  states per node.<sup>7</sup> The general idea is to separate from the whole population a subset of *clock nodes*, whose only functionality is to keep the time for the whole system. The other nodes work on computing the desired output and check whether they should proceed to the next stage of the computation when they interact with clock nodes. We note that while we use similar general structure and terminology as in [2], the meaning of some terms and the dynamics of our phase clock are somewhat different. A notable difference is that in [2] the clock nodes keep their time counters synchronized on the basis of the power of two choices in load balancing: when two nodes meet, only the lower counter is incremented. In contrast, we keep the updates of time counters as in the `Majority` and `FastMajority1` protocols: both interacting clock nodes increment their time counters, with the exception that the slower node is pulled up to the next  $\Theta(\log n)$ -length phase or epoch, if the faster node is already there.

The nodes in the `FastMajority2` protocol are partitioned into two sets with  $\Theta(n)$  nodes in each set. One set consists of *worker nodes*, which may carry opinion tokens and work through canceling-doubling phases to establish the majority opinion. These nodes maintain only information on whether they carry any token, and if so, then the value of the token

<sup>7</sup> It may be possible to use instead the ideas underlying other phase clocks, e.g. the  $\Theta(\log \log n)$ -state phase clock from [13], but this would not result in fewer states being needed for our protocol.

## 10:16 A Population Protocol for Exact Majority

(equivalently, the age of the token, that is, the number of times this token has been split). Each worker node has also a constant number of flags which indicate the current activities of the node (for example, whether it is in the canceling stage of a phase), but it does not maintain a detailed step counter. The other set consists of *clock nodes*, which maintain their detailed epoch-step counters, counting interactions with other clock nodes modulo  $2C \log n$ , for a suitably large constant  $C$ , and synchronizing with other clocks by the broadcast mechanism at the end of epoch. Thus the clock nodes update their counters in the same way as all nodes would update their counters in the **FastMajority1** protocol.

The worker nodes interact with each other in a similar way as in **FastMajority1**, but now to progress orderly through the computation they rely on the relatively tight synchronization of clock nodes. A worker node  $v$  advances to the next part of the current phase (or to the next phase, or the next epoch), when it interacts with a clock node whose clock indicates that  $v$  should progress. There is also a third type of nodes, the *terminator nodes*, which will appear later in the computation. A worker or clock node becomes a terminator node when it enters a *done* or *fail* state. The meaning and function of these special states are as in protocols **Majority** and **FastMajority1**.

A standard input instance, when each node is a worker with a token of value 1, is converted into a required initial workers-clocks configuration during the following  $O(\log n)$ -time pre-processing. When two value-1 tokens of opposite type interact they cancel out and one of the two involved nodes, say the one which has had the token  $B$ , becomes a clock node. If two value-1 tokens of the same type interact and their step counters have different parity, then the tokens are combined into one token of value 2. The combined token is taken by one node, while the other node, say the one with the odd counter, becomes a clock node. All nodes count their interactions during the pre-processing, but the  $O(\log n)$  states needed for this are re-used when the pre-processing completes. At this point the worker nodes have an input instance with the base value of tokens equal to 2. Some tokens may have value 1 (one may view them as if already split in the first phase) and some nodes may be empty.

Referring to the state space of the **FastMajority1** protocol, in the **FastMajority2** protocol each worker node  $v$  maintains data fields  $v.token$ ,  $v.epoch$  and  $v.age\_in\_epoch$  to carry information about tokens and their ages, and a constant number of flags to keep track of the status of the node and its progress through the current epoch and the current phase. These include the status flags from the **FastMajority1** protocol  $v.doubled$ ,  $v.out\_of\_sync$  and  $v.additional\_epoch$ , and flags indicating the progress: the  $v.epoch\_part$  flag from **FastMajority1** and a new (multi-valued) flag  $stage \in \{beginning, canceling, middle, doubling, ending\}$ . The clock nodes maintain the  $epoch\_step$  counters. The nodes have constant number of further flags, for example to support the initialization to workers and clocks and the implementation of the additional epoch and the slow backup protocol. Thus in total each node has only  $\Theta(\log n)$  states.

Further details of **FastMajority2**, including pseudocodes, details of the pre-processing and outline of the proof of Theorem 9 which summarizes the performance of this protocol, are given in the full version [9].

► **Theorem 9.** *The **FastMajority2** protocol uses  $\Theta(\log n)$  states, computes the exact majority w.h.p. within  $O(\log^{5/3} n)$  parallel time and stabilizes (in the correct all-done configuration or in the all-fail configuration) within the expected  $O(\log^{5/3} n)$  parallel time.*

► **Corollary 10.** *The exact majority can be computed with  $\Theta(\log n)$  states in  $O(\log^{5/3} n)$  parallel time w.h.p. and in expectation.*

---

**References**

---

- 1 Dan Alistarh, James Aspnes, David Eisenstat, Rati Gelashvili, and Ronald L. Rivest. Time-space trade-offs in population protocols. In Philip N. Klein, editor, *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 2560–2579. SIAM, 2017. doi:10.1137/1.9781611974782.169.
- 2 Dan Alistarh, James Aspnes, and Rati Gelashvili. Space-optimal majority in population protocols. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 2221–2239, 2018. doi:10.1137/1.9781611975031.144.
- 3 Dan Alistarh, Rati Gelashvili, and Milan Vojnovic. Fast and exact majority in population protocols. In Chryssis Georgiou and Paul G. Spirakis, editors, *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastián, Spain, July 21 - 23, 2015*, pages 47–56. ACM, 2015. URL: <http://dl.acm.org/citation.cfm?id=2767386>, doi:10.1145/2767386.2767429.
- 4 Dana Angluin, James Aspnes, Zoë Diamadi, Michael J. Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed Computing*, 18(4):235–253, 2006. doi:10.1007/s00446-005-0138-3.
- 5 Dana Angluin, James Aspnes, and David Eisenstat. Fast computation by population protocols with a leader. *Distributed Computing*, 21(3):183–199, 2008.
- 6 James Aspnes and Eric Ruppert. An introduction to population protocols. In Benoît Garbinato, Hugo Miranda, and Luís Rodrigues, editors, *Middleware for Network Eccentric and Mobile Applications*, pages 97–120. Springer-Verlag, 2009.
- 7 Florence Bénézit, Patrick Thiran, and Martin Vetterli. Interval consensus: From quantized gossip to voting. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP 2009, 19-24 April 2009, Taipei, Taiwan*, pages 3661–3664. IEEE, 2009. doi:10.1109/ICASSP.2009.4960420.
- 8 Petra Berenbrink, Robert Elsässer, Tom Friedetzky, Dominik Kaaser, Peter Kling, and Tomasz Radzik. Majority & stabilization in population protocols. Unpublished manuscript, available on arXiv, May 2018.
- 9 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 asymptotically optimal number of states. Unpublished manuscript, available on arXiv, May 2018. arXiv:1805.05157.
- 10 Petra Berenbrink, Tom Friedetzky, Peter Kling, Frederik Mallmann-Trenn, and Chris Wastell. Plurality consensus via shuffling: Lessons learned from load balancing. *CoRR*, abs/1602.01342, 2016. URL: <http://arxiv.org/abs/1602.01342>.
- 11 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 Elad Michael Schiller and Alexander A. Schwarzmann, editors, *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017, Washington, DC, USA, July 25-27, 2017*, pages 451–453. ACM, 2017. Full version available at arXiv:1705.01146. doi:10.1145/3087801.3087858.
- 12 Moez Draief and Milan Vojnovic. Convergence speed of binary interval consensus. In *INFOCOM 2010. 29th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies, 15-19 March 2010, San Diego, CA, USA*, pages 1792–1800. IEEE, 2010. doi:10.1109/INFCOM.2010.5461999.
- 13 Leszek Gasiñec and Grzegorz Stachowiak. Fast space optimal leader election in population protocols. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete*

- Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 2653–2667, 2018. doi:10.1137/1.9781611975031.169.
- 14 Leszek Gasiñec, Grzegorz Stachowiak, and Przemyslaw Uznanski. Almost logarithmic-time space optimal leader election in population protocols. *CoRR*, abs/1802.06867, 2018. arXiv:1802.06867.
  - 15 Mohsen Ghaffari and Merav Parter. A polylogarithmic gossip algorithm for plurality consensus. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing PODC*, pages 117–126, 2016.
  - 16 A. Kosowski and P. Uznański. Population Protocols Are Fast. *ArXiv e-prints*, 2018. arXiv:1802.06872v2.
  - 17 George B. Mertzios, Sotiris E. Nikolettseas, Christoforos L. Raptopoulos, and Paul G. Spirakis. Determining majority in networks with local interactions and very small local memory. In Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias, editors, *Automata, Languages, and Programming*, volume 8572 of *Lecture Notes in Computer Science*, pages 871–882. Springer Berlin Heidelberg, 2014. doi:10.1007/978-3-662-43948-7\_72.
  - 18 Thomas Sauerwald and He Sun. Tight bounds for randomized load balancing on arbitrary network topologies. In *53rd Annual IEEE Symposium on Foundations of Computer Science, FOCS 2012, New Brunswick, NJ, USA, October 20-23, 2012*, pages 341–350. IEEE Computer Society, 2012. doi:10.1109/FOCS.2012.86.

# Integrated Bounds for Disintegrated Storage

**Alon Berger**

Viterbi Department of Electrical Engineering, Technion, Haifa, Israel

**Idit Keidar**

Viterbi Department of Electrical Engineering, Technion, Haifa, Israel

**Alexander Spiegelman**

VMware Research, Israel

---

## Abstract

We point out a somewhat surprising similarity between non-authenticated Byzantine storage, coded storage, and certain emulations of shared registers from smaller ones. A common characteristic in all of these is the inability of reads to safely return a value obtained in a single atomic access to shared storage. We collectively refer to such systems as *disintegrated storage*, and show integrated space lower bounds for asynchronous regular wait-free emulations in all of them. In a nutshell, if readers are invisible, then the storage cost of such systems is inherently exponential in the size of written values; otherwise, it is at least linear in the number of readers. Our bounds are asymptotically tight to known algorithms, and thus justify their high costs.

**2012 ACM Subject Classification** Theory of computation → Distributed computing models, Computing methodologies → Distributed algorithms

**Keywords and phrases** storage, coding, lower bounds, space complexity, register emulations

**Digital Object Identifier** 10.4230/LIPIcs.DISC.2018.11

**Acknowledgements** We thank Yuval Cassuto, Gregory Chockler, Rati Gelashvili, and Yuanhao Wei for many insightful discussions on space bounds for coded storage and emulations of large registers from smaller ones.

## 1 Introduction

### 1.1 Space bounds for encoded, multi-register, and Byzantine storage

In many data sharing solutions, information needs to be read from multiple sources in order for a single value to be reconstructed. One such example is coded storage where multiple storage blocks need to be obtained in order to recover a single value that can be returned to the application [5, 9, 10, 16–18, 22, 23]. Another example arises in shared memory systems, where the granularity of atomic memory operations (such as load and store) is limited to a single word (e.g., 64 bits) and one wishes to atomically read and write larger values [22]. A third example is replicating data to overcome Byzantine faults (without authentication) or data corruption, where a reader expects to obtain the same block from multiple servers in order to validate it [1, 2, 19].

We refer to such systems collectively as *disintegrated storage* systems. We show that such a need to read data in multiple storage accesses inherently entails high storage costs: exponential in the data size if reads do not modify the storage, and otherwise linear in the number of concurrent reads. This stands in contrast to systems that use non-Byzantine replication, such as ABD [6], where, although meta-data (e.g., timestamps) is read from several sources, the recovered value need only be read from a single source.



© Alon Berger, Idit Keidar, and Alexander Spiegelman;  
licensed under Creative Commons License CC-BY

32nd International Symposium on Distributed Computing (DISC 2018).

Editors: Ulrich Schmid and Josef Widder; Article No. 11; pp. 11:1–11:18

Leibniz International Proceedings in Informatics



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

## 1.2 Our results

We consider a standard shared storage model (see Section 2). We refer to shared storage locations (representing memory words, disks, servers, etc.) as *objects*. To strengthen our lower bounds, we assume that objects are responsive, i.e., do not fail; the results hold a fortiori if objects can also be unresponsive [19]. Objects support general read-modify-write operations by asynchronous *processes*. We study *wait-free* emulations of a shared *regular register* [20].

Section 3 formally defines *disintegrated storage*. We use a notion of *blocks*, which are parts of a value kept in storage – code blocks, segments of a longer-than-word value, or full copies of a replicated value. A key assumption we make is that each block in the shared storage pertains to a single write operation; a similar assumption was made in previous studies [11,23]. The disintegration property then stipulates that a reader must obtain some number  $\tau > 1$  of blocks pertaining to a value  $v$  before returning  $v$ . For example,  $\tau$  blocks are needed in  $\tau$ -out-of- $n$  coded storage, whereas  $\tau = f + 1$  in  $f$ -tolerant Byzantine replication. To strengthen our results, we allow the storage to hold unbounded meta-data (e.g., timestamps), and count only the storage cost for blocks. Note that the need to obtain  $\tau$  blocks implies that meta-data cannot be used instead of actual data.

In Section 4 we give general lower bounds that apply to all types of disintegrated storage – replicated, coded, and multi-register. We first consider *invisible reads*, which do not modify the shared storage. This is a common paradigm in storage systems and often essential where readers outnumber writers and have different permissions. In this case, even with one reader and one writer, the storage size can be exponential; specifically, if value sizes are  $D$  (taken from a domain of size  $2^D$ ), then we show a lower bound of  $\tau + (\tau - 1) \left\lceil \frac{2^D - 1}{L} \right\rceil$  blocks, where  $L$  is the number of blocks in a reader’s local storage. That is, either the local storage of the reader or the shared storage is exponential.

Section 5 studies a more restrictive flavor of disintegrated storage, called  $\tau$ -*common write*, where a reader needs to obtain  $\tau$  blocks produced by the *same*  $\text{write}(v)$  operation in order to return  $v$ . In other words, if the reader obtains blocks that originate from two different  $\text{writes}$  of the same value, then it cannot recognize that they pertain to the same value, as is the case when blocks hold parts of a value or code blocks rather than replicas. In this case, the shared storage cost is high independently of the local memory size. Specifically, we show a bound of  $\tau \cdot 2^D$  blocks with invisible readers. In systems that use symmetric coding (i.e., where all blocks are of the same size, namely at least  $D/\tau$  bits), this implies a lower bound of  $D \cdot 2^D$  bits. For a modest value size of 20 bytes, the bound amounts to  $2.66 \cdot 10^{37}$  TB, and for 1KB values it is a whopping  $1.02 \cdot 10^{2457}$  TB.

We further consider *visible reads*, which can modify the objects’ meta-data. Such readers may indicate to the writers that a read is ongoing, and signal to them which blocks to retain. Using such signals, the exponential bound no longer holds – there are emulations that store a constant number of values per reader [2, 5, 13, 22]. We show that such linear growth with the number of readers is inherent. Our results are summarized in Table 1.

These bounds are tight as far as regularity and wait-freedom go: relaxing either requirement allows circumventing our results [1, 19]. As for storage cost, our lower bounds are asymptotically tight to known algorithms, whether reads are visible [2, 5, 22] or not [7, 16, 18, 21].

We note that the study of the inherent storage blowup in asynchronous coded systems has only recently begun [11, 23] and is still in its infancy. In this paper, we point out a somewhat surprising similarity between coded storage and other types of shared memory/storage, and show unified lower bounds for all of them. Section 6 concludes the paper and suggests directions for future work.

■ **Table 1** Lower bounds on shared storage space consumption, in units of blocks;  $D$  is the value size,  $\tau > 1$  is the number of data blocks required in order to recover a value,  $L \geq 1$  is the maximal number of blocks stored in a reader’s local data, and  $R$  the number of readers.

	Invisible Reads	Visible Reads
General Case	$\tau + (\tau - 1) \left\lceil \frac{2^D - 1}{L} \right\rceil$	$\tau + (\tau - 1) \cdot \min \left( \left\lceil \frac{2^D - 1}{L} \right\rceil, R \right)$
Common Write (e.g., coded storage)	$\tau \cdot 2^D$	$\tau + (\tau - 1) \cdot \min(2^D - 1, R)$

### 1.3 Related work and applicability of our bounds

Several works have studied the space complexity of register emulations. Two recent works [11, 23] show a dependence between storage cost and the number of *writers* in crash-tolerant storage, identifying a trade-off between the cost of replication ( $f + 1$  copies for tolerating  $f$  faults) and that of  $\tau$ -out-of- $n$  coding (linear in the number of writers). Though they do not explicitly consider disintegrated storage, it is fairly straightforward to adapt the proof from [23] to derive a lower bound of  $\tau W$  blocks with  $W$  writers. Here we consider the case of single-writer algorithms, where this bound is trivial. Other papers [3, 15] show limitations of multi-writer emulations when objects do not support atomic read-modify-write, whereas we consider single-writer emulations that do use read-modify-write.

Chockler et al. [14] define the notion of *amnesia* for register emulations with an infinite value domain, which intuitively captures the fact that an algorithm “forgets” all but a finite number of values written to it. They show that a wait-free regular emulation tolerating non-authenticated Byzantine faults with invisible readers cannot be amnesic, but do not show concrete space lower bounds. In this paper we consider a family of disintegrated storage algorithms, with visible and invisible readers, and show concrete bounds for the different cases; if the size of the value domain is unbounded, then our invisible reader bounds imply unbounded shared storage.

Disintegrated storage may also correspond to emulations of large registers from smaller ones, where  $\tau$  is the size of the big register divided by the size of the smaller one. Some algorithms in this vein, e.g., [22], indeed have the disintegration property, as the writer writes  $\tau$  blocks to a buffer and a reader obtains  $\tau$  blocks of the same write. These algorithms are naturally subject to our bounds. Other algorithms, e.g., [12, 13, 20], do not satisfy our assumption that each block in the shared storage pertains to a single **write** operation, and a reader may return a value based on blocks written by *different write* operations. Thus, our bounds do not apply to them. It is worth noting that these algorithms nevertheless either have readers signal to the writers and use space linear in the number of readers, or have invisible readers but use space exponential in the value size. Following an earlier publication of our work in [8], Wei [25] showed that these costs – either linear in the number of visible readers or exponential in the value size with invisible ones – are also inherent in emulations of large registers from smaller ones that *do* share blocks among writes, albeit do not use meta-data at all. Several questions remain open in this context: first, Wei’s bound is not applicable to all types of storage we consider (in particular, Byzantine), and does not apply to algorithms that use timestamps. Second, we are not familiar with any regular register emulations where readers write-back data, and it is unclear whether our bound may be circumvented this way.

Non-authenticated Byzantine storage algorithms that tolerate  $f$  faults need to read a value  $f + 1$  times in order to return it, and are thus  $\tau$ -disintegrated for  $\tau = f + 1$ . Note that while our model assumes objects are responsive, it a fortiori applies to scenarios where objects



may be unresponsive. Some algorithms circumvent our bound either by providing only safe semantics [19], or by forgoing wait-freedom [1]. Others use channels with unbounded capacity to push data to clients [7, 21] or potentially unbounded storage with best-effort garbage collection [18].

As for coded storage, whenever  $\tau$  blocks are required to reconstruct a value, the algorithm is  $\tau$ -disintegrated. And indeed, previous solutions in our model require unbounded storage or channels [9, 10, 16–18], or retain blocks for concurrent visible readers, consuming space linear in the number of readers [5]. Our bounds justify these costs. Our assumption that each block in the shared storage pertains to a single value is satisfied by almost all coded storage algorithms we are aware of, the only exception is [24], which indeed circumvents our lower bound but does not conform to regular register semantics. Other coded storage solutions, e.g., [4], are not subject to our bound because they may recover a value from a single block.

## 2 Preliminaries

### Shared storage model

We consider an asynchronous shared memory system consisting of two types of entities: A finite set  $O = \{o_1, \dots, o_n\}$  of objects comprising *shared storage*, and a set  $\Pi$  of processes. Every entity in the system stores *data*: an object's data is a single block from some domain  $\mathbb{B}$ , whereas a process' data is an array of up to  $L$  blocks from  $\mathbb{B}$ . We assume a bound  $L$  on the number of blocks in the data array of each process. In addition, each entity stores potentially infinite meta-data, *meta*. We denote an entity  $e$ 's data as  $e.data$  and likewise for  $e.meta$ . A system's *storage cost* is the number of objects in the shared storage,  $n$ .

Objects support atomic **get** and **update** actions by processes. We denote by  $\mathbf{a}_p$  an action  $\mathbf{a}$  performed by  $p$  and by  $o.\mathbf{a}_p$  an  $\mathbf{a}_p$  action at  $o$ . An  $o.\mathbf{update}_p$  is an arbitrary read-modify-write that possibly writes a block from  $\mathbb{B}$  to  $o.data$  and modifies  $o.meta$ ,  $p.meta$ , and  $p.data$ . An  $o.\mathbf{get}_p$  may replace a block in  $p.data$  with  $o.data$  and may modify  $p.meta$ .

### Algorithms, configurations, and runs

An *algorithm* defines the behaviors of processes as deterministic state machines, where state transitions are associated with actions. A *configuration* is a mapping to states (data and meta) from all system components, i.e., processes and objects. In an *initial configuration* all components are in their initial states.

We study algorithms (executed by processes in  $\Pi$ ) that emulate a high-level functionality, exposing high-level operations, and performing low-level **gets/updates** on objects. We say that high-level operations are *invoked* and *return* or *respond*. Note that, for simplicity, we model **gets** and **updates** as instantaneous actions, because the objects are assumed to be atomic, and we do not explicitly deal with object failures in this paper.

A *run* of algorithm  $\mathcal{A}$  is a (finite or infinite) alternating sequence of configurations and actions, beginning with some initial configuration, such that configuration transitions occur according to  $\mathcal{A}$ . Occurrences of actions in a run are called *events*. The possible events are high-level operation invocations and responses and **get/update** occurrences. We use the notion of time  $t$  during a run  $r$  to refer to the configuration reached after the  $t^{\text{th}}$  event in  $r$ . For a finite run  $r$  consisting of  $t$  events we define  $t_r \triangleq t$ . Two operations are *concurrent* in a run  $r$  if both are invoked in  $r$  before either returns. If a process  $p$ 's state transition from state  $\mathcal{S}$  is associated with a low-level action  $\mathbf{a}_p \in \{\mathbf{get}_p, \mathbf{update}_p\}$ , we say that  $\mathbf{a}_p$  is *enabled* in  $\mathcal{S}$ . A run  $r'$  is an *extension* of a (finite) run  $r$  if  $r$  is a prefix of  $r'$ ; we denote by  $r' \setminus r$  the



suffix of  $r'$  that starts at  $t_r$ . If a high-level operation  $op$  has been invoked by process  $p$  but has not returned by time  $t$  in a run  $r$ , we say that  $op$ 's invocation is *pending* at  $t$  in  $r$ . We assume that each process' first action in a run is an invocation, and a process has at most one pending invocation at any time.

For  $e \in \Pi \cup O$ , we denote by  $e.data(r, t)$  the set of distinct blocks stored in  $e.data$  at time  $t$  in a run  $r$ . Since for an object  $o$ ,  $|o.data(r, t)| = 1$ , we sometimes refer to  $o.data(r, t)$  as the block itself, by slight abuse of notation. We say that  $p$  *obtains* a block  $b$  at time  $t$  in a run  $r$ , if  $b \notin p.data(r, t)$  and  $b \in p.data(r, t + 1)$ .

### Register emulations

We study algorithms that emulate a shared *register* [20], which stores a value  $v$  from some domain  $\mathbb{V}$ . We assume that  $|\mathbb{V}| = 2^D > 1$ , i.e., values can be represented using  $D > 0$  bits. For simplicity, we assume that each run begins with a dummy initialization operation that writes the register's initial value and does not overlap any operation. The register exposes high-level  $\text{read}_p$  and  $\text{write}_p(v)$  operations of values  $v \in \mathbb{V}$  to processes  $p \in \Pi$ . We consider single-writer (SW) registers where the application at only one process (the *writer*) invokes **writes**, and hence omit the subscript  $p$  from  $\text{write}(v)$ . The remaining  $R \triangleq |\Pi| - 1$  processes are limited to performing **reads**, and are referred to as *readers*. For brevity, we refer to the subsequence of a run where a specific invocation of a  $\text{write}(v)/\text{read}_p$  is pending simply as a  $\text{write}(v)/\text{read}_p$  operation.

We assume that whenever a  $\text{read}_p$  operation is invoked at time  $t$  in a run  $r$ ,  $p.data(r, t)$  is empty. We consider two scenarios: (1) *invisible reads*, where **reads** do not use **updates**, and (2) *visible reads*, where **reads** may perform **updates** that update meta-data (only) in the shared storage. Note that readers do *not* write actual data, which is usually the case in regular register emulations, defined below. In a single-reader (SR) register  $R = 1$ , and if  $R > 1$  the register is multi-reader (MR). If the states of the writer and the objects at the end of a finite run  $r$  are equal to their respective states at the end of a finite run  $r'$ , we say that  $t_r$  and  $t_{r'}$  are *indistinguishable* to the writer and objects, and denote:  $t_r \approx_w t_{r'}$ .

Our safety requirement is *regularity* [20]: a **read**  $rd$  must return the value of either the last **write**  $w$  that returns before  $rd$  is invoked, or some write that is concurrent with  $rd$ . For liveness, we require *wait-freedom*, namely that every operation invoked by a process  $p$  returns within a finite number of  $p$ 's actions. In other words, if  $p$  is given infinitely many opportunities to perform actions, it completes its operation regardless of the actions of other processes.

## 3 Disintegrated storage

As noted above, existing wait-free algorithms of coded and/or Byzantine-fault-tolerant storage with invisible readers may store all values ever written [7, 9, 11, 16–18, 21]. This is because if old values are erased, it is possible for a slow reader to never find sufficiently many blocks of the same value so as to be able to return it. If readers are visible, then a value per reader is retained. We want to prove that these costs are inherent. The challenge in proving such space lower bounds is that the aforementioned algorithms use unbounded timestamps. How can we show a space lower bound if we want to allow algorithms to use unbounded timestamps? We address this by allowing meta-data to store timestamps, etc., and by not counting the storage cost for meta-data. For example, the above algorithms store timestamps in meta-data alongside data blocks and use them to figure out which data is safe to return, but still need  $\tau$  actual blocks/copies of a value in order to return it. Note that for the sake of the lower

## 11:6 Integrated Bounds for Disintegrated Storage

bound, we do not restrict how meta-data is used; all we require is that the algorithm read  $\tau$  data blocks of the same value (or **write**), and we do not specify how the algorithm knows that they pertain to the same value (or **write**). To formalize the property that the algorithm returns  $\tau$  blocks pertaining to the same value or **write**, we need to track, for each block in the shared storage, which **write** produced it. To this end, we define *labels*. Labels are only an analysis tool, and do not exist anywhere. In particular, they are not timestamps, not meta-data, and not explicitly known to the algorithm. As an external observer, we may add them as abstract state to the blocks, and track how they change.

### Labels

We associate each block  $b$  in the shared or local storage with a set of labels,  $Labels(b)$ , as we now explain. For an algorithm  $\mathcal{A}$  and  $v \in \mathbb{V}$ , denote by  $\mathbb{W}_v^{\mathcal{A}}$  the set of **write**( $v$ ) operations invoked in runs of  $\mathcal{A}$ . For  $V \subseteq \mathbb{V}$ , we denote  $\mathbb{W}_V^{\mathcal{A}} \triangleq \bigcup_{v \in V} \mathbb{W}_v^{\mathcal{A}}$ , and let  $\mathbb{W}^{\mathcal{A}} \triangleq \mathbb{W}_V^{\mathcal{A}}$ . For clarity, we omit  $\mathcal{A}$  when obvious from the context, and refer simply to  $\mathbb{W}_v$ ,  $\mathbb{W}_V$ , and  $\mathbb{W}$ . We assume that the  $k^{th}$  **update** event occurring in a **write** operation  $w \in \mathbb{W}$  tags the block  $b$  it stores (if any) with a unique label  $\langle w, k \rangle$ , so  $Labels(b)$  becomes  $\{\langle w, k \rangle\}$ .

Whereas our assumption that each block in the shared storage pertains to a single write rules out associating multiple labels with such a block, we do allow the reader's meta-data to recall multiple accesses encountering the same block. For example, when blocks are copies of a replicated value, the reader can store one instance of the value in local memory and keep a list of the objects where the value was encountered. To this end, a block in a reader's *data* may be tagged with multiple labels: when a reader  $p$  obtains a block  $b$  from an object  $o$  at time  $t$  in a run  $r$ , the block  $b$  in  $p.data(r, t+1)$  is tagged with  $Labels(o.data(r, t))$ ; if at time  $t' > t$   $p.data$  still contains  $b$  and  $p$  performs an action on an object  $o'$  s.t.  $o'.data(r, t') = b$  and the latter is tagged with label  $\ell$ ,  $p$  adds  $\ell$  to  $Labels(b)$  (regardless of whether  $b$  is added to  $p.data$  once more). When all copies of a block are removed from  $p.data$ , all its labels are “forgotten”. We emphasize that labels are not stored anywhere, and are only used for analysis.

We track the labels of a value  $v \in \mathbb{V}$  at time  $t$  in a run  $r$  using the sets  $S$ -labels( $v, r, t$ ), of labels in the shared storage,  $L$ -labels $_p(v, r, t)$ , of labels in process  $p$ 's local storage, and  $All$ -labels $_p(v, r, t)$ , a combination of both. Formally,

- $S$ -labels( $v, r, t$ )  $\triangleq (\bigcup_{o \in O} Labels(o.data(r, t))) \cap (\mathbb{W}_v \times \mathbb{N})$ .
- $L$ -labels $_p(v, r, t)$   $\triangleq (\bigcup_{b \in p.data(r, t)} Labels(b)) \cap (\mathbb{W}_v \times \mathbb{N})$ .
- $All$ -labels $_p(v, r, t)$   $\triangleq L$ -labels $_p(v, r, t) \cup S$ -labels( $v, r, t$ ).

For a time  $t$  in a run  $r$  and  $p \in \Pi$ , we define  $values_p(r, t) \triangleq \{v \in \mathbb{V} \mid L$ -labels $_p(v, r, t) \neq \emptyset\}$ .

Similarly, we track labels associated with a particular **write**  $w \in \mathbb{W}$  accessible by process  $p \in \Pi$  at time  $t$  in a run  $r$ :

- $S$ -labels( $w, r, t$ )  $\triangleq (\bigcup_{o \in O} Labels(o.data(r, t))) \cap (\{w\} \times \mathbb{N})$ .
- $L$ -labels $_p(w, r, t)$   $\triangleq (\bigcup_{b \in p.data(r, t)} Labels(b)) \cap (\{w\} \times \mathbb{N})$ .
- $All$ -labels $_p(w, r, t)$   $\triangleq L$ -labels $_p(w, r, t) \cup S$ -labels( $w, r, t$ ).

We define  $writes_p(r, t) \triangleq \{w \in \mathbb{W} \mid L$ -labels $_p(w, r, t) \neq \emptyset\}$ . Note that for all  $v \in \mathbb{V}$  and  $w \in \mathbb{W}_v$ , (1)  $S$ -labels( $w, r, t$ )  $\subseteq S$ -labels( $v, r, t$ ), (2)  $L$ -labels $_p(w, r, t) \subseteq L$ -labels $_p(v, r, t)$ , and (3)  $All$ -labels $_p(w, r, t) \subseteq All$ -labels $_p(v, r, t)$ .

Since readers do not write-back:

► **Observation 1.** *If the  $t^{th}$  event in a run  $r$  is of a reader  $p \in \Pi$ , then for all  $v \in \mathbb{V}, w \in \mathbb{W}$ :  $All$ -labels $_p(v, r, t) \subseteq All$ -labels $_p(v, r, t-1)$  and  $All$ -labels $_p(w, r, t) \subseteq All$ -labels $_p(w, r, t-1)$ .*

### Disintegrated storage

Intuitively, in disintegrated storage register emulations, for a  $\text{read}_p$  to return  $v$ ,  $p$  must encounter  $\tau > 1$  blocks corresponding to  $v$  that were produced by separate  $\text{update}$  events. To formalize this, we use labels:

► **Definition 2** ( $\tau$ -disintegrated storage). If a return of  $v \in \mathbb{V}$  by a  $\text{read}_p$  invocation is enabled at time  $t$  in a run  $r$  then  $|L\text{-labels}_p(v, r, t)| \geq \tau$ .

Thus, a reader can only return  $v$  if it recalls (in its local memory) obtaining blocks of  $v$  with  $\tau$  different labels.

A more restrictive case of  $\tau$ -disintegrated storage occurs when readers cannot identify whether two blocks pertain to a common value unless they are produced by a common write that identifies them, e.g., with the same timestamp. This is the case when value parts or code words are stored in objects rather than full replicas.

To capture this case, for a block  $b \in \bigcup_{e \in O \cup \Pi} e.\text{data}$ , a value  $v \in \mathbb{V}$ , and a  $\text{write } w \in \mathbb{W}_v$ , if  $\exists k \in \mathbb{N}$  s.t.  $\langle w, k \rangle \in \text{Labels}(b)$ , we say that  $w$  is an *origin write* of  $b$  and  $v$  is an *origin value* of  $b$ . Common write  $\tau$ -disintegrated storage is then defined:

► **Definition 3** (common write  $\tau$ -disintegrated storage). If a return of  $v \in \mathbb{V}$  by a  $\text{read}_p$  invocation is enabled at time  $t$  in a run  $r$  then  $\exists w \in \mathbb{W}_v : |L\text{-labels}_p(w, r, t)| \geq \tau$ .

Note that we do not further require  $p.\text{data}$  to actually hold  $\tau$  blocks with a common write, because the weaker definition suffices for our lower bounds. For brevity, we henceforth refer to a common write  $\tau$ -disintegrated storage algorithm simply as  $\tau$ -common write.

### Permanence

Our lower bounds will all stem, in one way or another, from the observation that in wait-free disintegrated storage, every run must reach a point after which some values (and in the case of common write, also some  $\text{writes}$ ) must *permanently* have a certain number of blocks in the shared storage. This is captured by the following definition:

► **Definition 4** (permanence). Consider a finite run  $r$ ,  $k \in \mathbb{N}$ , a set  $S \subseteq \mathbb{V}$ , and a set of readers  $\Theta \subset \Pi$ . Let  $z \in \mathbb{V} \cup \mathbb{W}$  be a value or a  $\text{write}$  operation. We say that  $z$  is  $\langle k, \Theta, S \rangle$ -*permanent* in  $r$  if in every finite extension  $r'$  of  $r$  s.t. in  $r' \setminus r$  readers in  $\Theta$  do not take actions and  $\text{writes}$  are limited to values from  $S$ ,  $|S\text{-labels}(z, r', t_{r'})| \geq k$ .

Intuitively, this means that the shared storage continues to hold  $k$  blocks of  $z$  as long as readers in  $\Theta$  do not signal to the writer and only values from  $S$  are written. For brevity, when the particular sets  $S$  and  $\Theta$  are not important, we refer to the value shortly as  $k$ -permanent. The observation below follows immediately from the definition of permanence:

- **Observation 5.** Let  $v \in \mathbb{V}$ ,  $w \in \mathbb{W}_v$ ,  $k \in \mathbb{N}$ ,  $V_2 \subseteq V_1 \subseteq \mathbb{V}$ ,  $\Theta_1 \subseteq \Theta_2 \subset \Pi$ .
1. If  $w$  is  $\langle k, \Theta_1, V_1 \rangle$ -permanent in a finite run  $r$  then  $v$  is  $\langle k, \Theta_1, V_1 \rangle$ -permanent in  $r$ .
  2. If  $v$  is  $\langle k, \Theta_1, V_1 \rangle$ -permanent in a finite run  $r$  then  $v$  is  $\langle k, \Theta_2, V_2 \rangle$ -permanent in all finite extensions  $r'$  of  $r$  where in  $r' \setminus r$   $\text{writes}$  are limited to values from  $V_1$  and readers in  $\Theta_1$  do not take actions.

Since each object holds a single block associated with a single label:

► **Observation 6.** For time  $t$  in a run  $r$ , the number of objects is:  $n \geq |\bigcup_{v \in \mathbb{V}} S\text{-labels}(v, r, t)|$ .

Thus, if there are  $m$  different  $k$ -permanent values in a run, then  $n \geq mk$ . We observe that with invisible readers, the set  $\Theta$  is immaterial:

► **Observation 7.** Consider  $k \in \mathbb{N}$ ,  $V \subseteq \mathbb{V}$ , and a finite run  $r$  with an invisible reader  $p \in \Pi$ . If  $z \in \mathbb{V} \cup \mathbb{W}$  is  $\langle k, \{p\}, V \rangle$ -permanent in  $r$  then  $z$  is  $\langle k, \emptyset, V \rangle$ -permanent in  $r$ .

The specific lower bounds for the four scenarios we consider differ in the number of permanent values/writes and the number of blocks per value/write ( $k = \tau - 1$  or  $k = \tau$ ) we can force the shared storage to retain forever in each case. Interestingly, our notion of permanence resembles the idea that an algorithm is not *amnesic* introduced in [14] (see Section 1.3), but is more fine-grained in specifying the number of permanent blocks and restricting executions under which they are retained.

## 4 Lower bounds for disintegrated storage

In this section we provide lower bounds on the number of objects required for  $\tau$ -disintegrated storage regular wait-free register emulations. Section 4.1 proves two general properties of regular wait-free  $\tau$ -disintegrated storage algorithms. We show in Section 4.2 that with invisible reads, unless the readers' local storage size is exponential in  $D$ , the storage cost of such emulations is at least exponential in  $D$ . Finally, Section 4.3 shows that if reads are visible, then the storage cost increases linearly with the number of readers.

### 4.1 General properties

We first show that because readers must make progress even if the writer stops taking steps, at least  $2\tau - 1$  blocks are required regardless of the number of readers.

► **Claim 8.** Consider  $v_1, v_2 \in \mathbb{V}$  and a run  $r$  of a wait-free regular  $\tau$ -disintegrated storage algorithm with two consecutive responded writes  $w_1 \in \mathbb{W}_{v_1}$  followed by  $w_2 \in \mathbb{W}_{v_2}$ . Let  $p \in \Pi$  be a reader s.t. no  $\text{read}_p$  is pending in  $r$ . Then there is a time  $t$  between the returns of  $w_1$  and  $w_2$  when  $|S\text{-labels}(v_1, r, t)| \geq \tau$  and  $|S\text{-labels}(v_2, r, t)| \geq \tau - 1$ .

**Proof.** We first argue that at the time  $t_i$ ,  $i \in \{1, 2\}$  when  $w_i$  returns,  $|S\text{-labels}(v_i, r, t_i)| \geq \tau$ . Assume the contrary. We build a run  $r'$  identical to  $r$  up to  $t_i$ . In  $r'$ , only process  $p$  performs actions after time  $t_i$ . Next, invoke a  $\text{read}_p$  operation  $rd$ . By regularity and wait-freedom,  $rd$  must return  $v_i$ . Before performing actions on objects,  $p.\text{data}(r', t_i)$  is empty, thus, from  $\tau$ -disintegrated storage,  $p$  must encounter at least  $\tau$  blocks with an origin value of  $v_i$  in order to return it. Since no process other than  $p$  takes actions,  $|S\text{-labels}(v_i, r', t')| < \tau$  for all  $t' \geq t_i$  onward, so  $rd$  cannot find these blocks and does not return  $v_i$ , a contradiction. It follows that in  $r'$  at  $t_i$ , and hence also in  $r$  at  $t_i$ ,  $|S\text{-labels}(v_i, r, t_i)| \geq \tau$ .

Next, if at  $t_1$ ,  $|S\text{-labels}(v_2, r, t_1)| \geq \tau - 1$  then we are done. Otherwise, observe that objects are accessed one-at-a-time. Therefore, and since  $|S\text{-labels}(v_2, r, t_1)| < \tau - 1$ , there exists a time  $t$  between  $t_1$  and  $t_2$  when  $|S\text{-labels}(v_2, r, t)| = \tau - 1$ .

Finally, assume that  $|S\text{-labels}(v_1, r, t)| < \tau$ . Build a run  $r''$  identical to  $r$  up to  $t$ , where again only  $p$  takes actions after  $t$ . As above, it follows by regularity,  $\tau$ -disintegrated storage, and  $p.\text{data}(r'', t) = \emptyset$ , that  $rd$  never returns, in violation of wait-freedom. It follows that  $|S\text{-labels}(v_1, r'', t)| = |S\text{-labels}(v_1, r, t)| \geq \tau$ . ◀

The following lemma states that every non-empty set  $V$  can be split into two disjoint subsets, where one contains a value that is  $(\tau - 1)$ -permanent with respect to the other subset. The idea is to show that in the absence of such a value, a reader's accesses to the shared storage may be scheduled in a way that prevents the reader from obtaining  $\tau$  labels of the same value. The logic of the proof is the following: we restrict writes to a set of values  $V$ , and consider the set  $S$  of values with blocks in  $p.\text{data} \cap V$ . If no value in  $S$  is

$(\tau - 1)$ -permanent, then we can bring the shared storage to a state where none of the values in  $S$  have  $\tau$  labels, preventing the reader from obtaining the  $\tau$  labels required to return. By regularity, readers cannot return other values. The formal proof is slightly more subtle, because it needs to consider  $L$ -labels $_p$  as well as labels in the shared storage. It shows that the total number of labels of values in  $S$  (in both the shared and local storage) remains below  $\tau$  whenever  $p$  takes a step.

► **Lemma 9.** *Consider a non-empty set of values  $V \subseteq \mathbb{V}$ , a set of readers  $\Theta \subset \Pi$ , a reader  $p \in \Pi \setminus \Theta$ , and a finite run  $r$  of a wait-free regular  $\tau$ -disintegrated storage algorithm. Then there is a subset  $S \subseteq V$  of size  $1 \leq |S| \leq L$  and an extension  $r'$  of  $r$  where some value  $v \in S$  is  $\langle \tau - 1, \Theta \cup \{p\}, V \setminus S \rangle$ -permanent and s.t. in  $r' \setminus r$  writes are limited to values from  $V$  and readers in  $\Theta$  do not take steps.*

**Proof.** Assume by contradiction that the lemma does not hold. We construct an extension  $r'$  of  $r$  where a  $\text{read}_p$  operation includes infinitely many actions of  $p$  yet does not return. To this end, we show that the following property holds at specific times in  $r' \setminus r$ :

$$\varphi(\hat{r}, t) \triangleq \forall v \in \text{values}_p(\hat{r}, t) \cap V : |\text{All-labels}_p(v, \hat{r}, t)| < \tau.$$

First, extend  $r$  to  $r_0$  by returning any pending  $\text{read}_p$  and  $\text{write}$ , invoking and returning a  $\text{write}(v_0)$  for some  $v_0 \in \mathbb{V}$  (the operations eventually return, by wait-freedom), and finally invoking a  $\text{read}_p$  operation  $rd$  without allowing it to take actions. We now prove by induction that for all  $k \in \mathbb{N}$ , there exists an extension  $r'$  of  $r_0$  where (1)  $\varphi(r', t_{r'})$  holds and in  $r' \setminus r$ : (2) writes are restricted to values from  $V$ , (3)  $p$  performs  $k$  actions on objects following  $rd$ 's invocation, and (4)  $rd$ 's return is not enabled, and (5) processes in  $\Theta$  do not take steps.

Base: for  $k = 0$ , consider  $r' = r_0$ . (3,5) hold trivially. (2) holds since the only  $\text{write}$  in  $r' \setminus r$  is of  $v_0 \in V$ . Since  $p$  performs no actions following the invocation of  $rd$ ,  $p.\text{data}(r', t_{r'})$  is empty. Therefore, (1)  $\varphi(r', t_{r'})$  is vacuously true, and  $L$ -labels $_p(v, r, t)$  is empty for all  $v \in \mathbb{V}$ , thus (4)  $rd$ 's return is not enabled by  $\tau$ -disintegrated storage.

Step: assume inductively such an extension  $r_1$  of  $r_0$  with  $k \geq 0$  actions performed by  $p$  following  $rd$ 's invocation. Since  $rd$  cannot return, by wait-freedom, an action  $\mathbf{a}_p$  is enabled on some object. We construct an extension  $r_2$  of  $r_1$  by letting  $\mathbf{a}_p$  occur at time  $t_{r_1}$ . We consider two cases:

1.  $p$  does not obtain a block with an origin value in  $V \setminus \text{values}_p(r_1, t_{r_1})$  at  $\mathbf{a}_p$ , thus  $\text{values}_p(r_2, t_{r_2}) \cap V \subseteq \text{values}_p(r_1, t_{r_1}) \cap V$ . Then, by Observation 1 and the inductive hypothesis, (1)  $\varphi(r_2, t_{r_2})$  holds and thus, by  $\tau$ -disintegrated storage,  $rd$  cannot return any value  $v \in \text{values}_p(r_2, t_{r_2}) \cap V$  at  $t_{r_2}$ . (4) It cannot return any other value in  $\text{values}_p(r_2, t_{r_2})$  by regularity, and  $r_2$  satisfies the induction hypothesis for  $k + 1$ , as (2,3,5) trivially hold.

2.  $p$  obtains a block with origin value  $u \in V \setminus \text{values}_p(r_1, t_{r_1})$  at time  $t_{r_1}$ . Then  $|L\text{-labels}_p(u, r_2, t_{r_2})| = 1$ . By Observation 1 and the inductive hypothesis, for all  $v \in \text{values}_p(r_2, t_{r_2}) \setminus \{u\}$ ,  $|L\text{-labels}_p(v, r_2, t_{r_2})| < \tau$ , and thus  $rd$ 's return is not enabled at time  $t_{r_2}$  by  $\tau$ -disintegrated storage and regularity.

Let  $S = \text{values}_p(r_2, t_{r_2}) \cap V$ , and note that  $|S| \geq 1$  (since  $u \in S$ ) and that  $|S| \leq |p.\text{data}| \leq L$ . By the contradicting assumption,  $u$  is not  $\langle \tau - 1, \Theta \cup \{p\}, V \setminus S \rangle$ -permanent in  $r_2$ , thus there exists an extension  $r_3$  of  $r_2$  s.t.  $|S\text{-labels}(u, r_3, t_{r_3})| < \tau - 1$  and in  $r_3 \setminus r_2$  writes are limited to values from  $V \setminus S$  and no readers in  $\Theta \cup \{p\}$  take steps (3,5 hold). Since  $p$  takes no steps in  $r_3 \setminus r_2$ , we have that  $L\text{-labels}_p(u, r_3, t_{r_3}) = L\text{-labels}_p(u, r_2, t_{r_2})$ , yielding:

$$|\text{All-labels}_p(u, r_3, t_{r_3})| \leq |L\text{-labels}_p(u, r_2, t_{r_2})| + |S\text{-labels}(u, r_3, t_{r_3})| < 1 + (\tau - 1) = \tau. \quad (1)$$

All writes invoked after  $t_{r_2}$  are from  $\mathbb{W}_{V \setminus S}$  (2 holds), and therefore do not produce new labels associated with values in  $S$ . Since no values in  $S$  are written after  $t_{r_1}$  and

## 11:10 Integrated Bounds for Disintegrated Storage

readers' actions do not affect the sets  $S$ -labels, by Observation 1, we have that  $\forall v \in S$ ,  $All\text{-}labels_p(v, r_3, t_{r_3}) \subseteq All\text{-}labels_p(v, r_1, t_{r_1})$ , and since  $\varphi(r_1, t_{r_1})$  holds (inductively) and  $S \setminus \{u\} \subseteq values_p(r_1, t_{r_1}) \cap V$ ,

$$\forall v \in S \setminus \{u\} : |All\text{-}labels_p(v, r_3, t_{r_3})| \leq |All\text{-}labels_p(v, r_1, t_{r_1})| < \tau. \quad (2)$$

From Equations 1 and 2, and since  $values_p(r_3, t_{r_3}) \cap V = values_p(r_2, t_{r_2}) \cap V = S$ , we get  $\varphi(r_3, t_{r_3})$  (1). Since  $rd$ 's return was not enabled at time  $t_{r_2}$  and it took no actions since, its return is still not enabled (4), and we are done.  $\blacktriangleleft$

### 4.2 Invisible reads

We now consider a setting of a single reader and single writer where **reads** are invisible. To show the following theorem, we “blow up” the shared storage by repeatedly invoking Lemma 9, each time adding one more  $(\tau - 1)$ -permanent value, yielding the following bound:

**► Theorem 10.** *The storage cost of a regular  $\tau$ -disintegrated storage wait-free SRSW register emulation where **reads** are invisible is at least  $\tau + (\tau - 1) \left\lceil \frac{2^D - 1}{L} \right\rceil$  blocks.*

When readers are invisible, the set  $\Theta$  is of no significance, so we consider  $\emptyset$ . Given a set of values  $V$ , the value added by Lemma 9 is  $\langle \tau - 1, \emptyset, V \setminus S \rangle$ -permanent for a smaller set of values  $V \setminus S$  where  $|S| \leq L$ . Therefore, we can invoke Lemma 9  $m = \left\lceil \frac{2^D - 1}{L} \right\rceil - 1$  times before running out of values, showing the following:

**► Lemma 11.** *Let  $p \in \Pi$  be an invisible reader. There exist finite runs  $r_0, \dots, r_m$  and sets of values  $V_0 \supset V_1 \supset \dots \supset V_m$  and  $U_0 \subset U_1 \subset \dots \subset U_m$ , such that for all  $0 \leq k \leq m$ :*

- (1)  $|V_k| \geq 2^D - Lk$ ,  $|U_k| = k$ ,  $V_k \cap U_k = \emptyset$ , and
- (2) all elements of  $U_k$  are  $\langle \tau - 1, \emptyset, V_k \rangle$ -permanent in  $r_k$ .

**Proof.** By induction. Base:  $r_0$  is the empty run,  $V_0 = \mathbb{V}$  and  $U_0 = \emptyset$ . Assume inductively that the lemma holds for  $k < m$ . Since  $m < \frac{2^D - 1}{L}$ , we get:  $|V_k| > 2^D - L \frac{2^D - 1}{L} = 1$ . Since  $V_k$  is non-empty and  $|\emptyset| < R$ , by Lemma 9 there exist an extension  $r_{k+1}$  of  $r_k$  where **writes** in  $r_{k+1} \setminus r_k$  are limited to values from  $V_k$ , a set  $S \subset V_k$ ,  $1 \leq |S| \leq L$ , and a value  $v \in S$  that is  $\langle \tau - 1, \{p\}, V_k \setminus S \rangle$ -permanent in  $r_{k+1}$ .

Let  $V_{k+1} = V_k \setminus S$  and  $U_{k+1} = U_k \cup \{v\}$ . Note that, because  $V_k \cap U_k = \emptyset$  and  $v \in S \subset V_k$ , we get that  $V_{k+1} \cap U_{k+1} = \emptyset$  and  $|U_{k+1}| = |U_k| + 1 = k + 1$ . Since  $1 \leq |S| \leq L$  we have that  $V_k \supset V_{k+1}$  and  $|V_{k+1}| \geq |V_k| - |S| \geq 2^D - L(k + 1)$ . By the inductive assumption and Observation 5, all values in  $U_k$  are  $\langle \tau - 1, \emptyset, V_{k+1} \rangle$ -permanent in  $r_{k+1}$ . By Observation 7,  $v$  is also  $\langle \tau - 1, \emptyset, V_{k+1} \rangle$ -permanent in  $r_{k+1}$  and we are done.  $\blacktriangleleft$

Our bound combines the  $2\tau - 1$  blocks of Claim 8 with the  $(\tau - 1)m$  from Lemma 11:

**Proof (Theorem 10).** Consider an invisible reader  $p \in \Pi$  and construct  $r_m, V_m$ , and  $U_m$  as in Lemma 11. Note that  $V_m$  contains at least two distinct values that are not in  $U_m$ , since  $V_m \cap U_m = \emptyset$  and  $|V_m| \geq 2^D - Lm > 2^D - L \frac{2^D - 1}{L} = 1$ . Extend  $r_m$  to  $r_{m+1}$  by invoking and returning **write**( $v$ ) and **write**( $v'$ ) for  $v, v' \in V_m$ .

By Claim 8, there is a time  $t \geq t_{r_m}$  in  $r_{m+1}$  when there are  $2\tau - 1$  blocks in the shared storage with origin values of  $v$  or  $v'$ . In addition, by Lemma 11,  $U_m$  consists of  $m$  values that are  $\langle \tau - 1, \emptyset, V_m \rangle$ -permanent in  $r_m$ , and since **writes** in  $r_{m+1} \setminus r_m$  are of values from  $V_m$ , the values in  $U_m$  remain  $\langle \tau - 1, \emptyset, V_m \rangle$ -permanent in  $r_{m+1}$ . By Observation 6:

$$n \geq 2\tau - 1 + (\tau - 1)m = \tau + (\tau - 1)(m + 1) = \tau + (\tau - 1) \left\lceil \frac{2^D - 1}{L} \right\rceil. \quad \blacktriangleleft$$

### 4.3 Visible reads

We now consider systems where readers may write meta-data in the shared storage. We use a similar technique as in Lemma 11, except that due to readers' **updates**, the indistinguishability argument can no longer be used. Instead, we invoke a new reader for each extension, and therefore the number of runs might be limited by the number of readers,  $R$ :

► **Theorem 12.** *The storage cost of a regular  $\tau$ -disintegrated storage wait-free MRSW register emulation with  $R$  readers is at least  $\tau + (\tau - 1) \cdot \min\left(\left\lceil \frac{2^D - 1}{L} \right\rceil, R\right)$  blocks.*

To achieve this bound, we use Lemma 9 again to construct  $N = \min\left(\left\lceil \frac{2^D - 1}{L} \right\rceil, R\right) - 1$  extensions of the empty run (note that it does not assume invisible reads).

► **Lemma 13.** *There exist finite runs  $r_0, \dots, r_N$ , sets of values  $V_0 \supset V_1 \supset \dots \supset V_N$  and  $U_0 \subset U_1 \subset \dots \subset U_N$ , and sets of readers  $\Theta_0 \subset \Theta_1 \subset \dots \subset \Theta_N$ , such that for all  $0 \leq k \leq N$ :*

- (1)  $|V_k| \geq 2^D - Lk$ ,  $|U_k| = |\Theta_k| = k$ ,  $V_k \cap U_k = \emptyset$ , and
- (2) all elements of  $U_k$  are  $\langle \tau - 1, \Theta_k, V_k \rangle$ -permanent in  $r_k$ .

**Proof.** By induction. Base:  $r_0$  is the empty run,  $V_0 = \mathbb{V}$ ,  $\Theta_0 = U_0 = \emptyset$ . Assume inductively such  $r_k, V_k, U_k$ , and  $\Theta_k$  for  $k < N$ , and construct  $r_{k+1}$  as follows: since  $R - |\Theta_k| > 0$ , there is a reader  $p \in \Pi \setminus \Theta_k$ . Since  $N < \frac{2^D - 1}{L}$ , we get  $|V_k| > 2^D - LN > 1$ . Therefore, by Lemma 9, there exist an extension  $r_{k+1}$  of  $r_k$  where in  $r_{k+1} \setminus r_k$  writes are limited to values from  $V_k$  and readers in  $\Theta_k$  do not take steps, a set  $S \subseteq V_k$ ,  $1 \leq |S| \leq L$ , and a value  $v \in S$  that is  $\langle \tau - 1, \Theta_k \cup \{p\}, V_k \setminus S \rangle$ -permanent in  $r_{k+1}$ .

Let  $V_{k+1} = V_k \setminus S$  and  $U_{k+1} = U_k \cup \{v\}$ . Note that, because  $V_k \cap U_k = \emptyset$  and  $v \in S \subset V_k$ , it follows that  $V_{k+1} \cap U_{k+1} = \emptyset$  and  $|U_{k+1}| = k + 1$ . Furthermore, since  $1 \leq |S| \leq L$ , we get:  $V_k \supset V_{k+1}$  and  $|V_{k+1}| \geq |V_k| - |S| \geq 2^D - L(k + 1)$ . Finally, let  $\Theta_{k+1} = \Theta_k \cup \{p\}$ . By the inductive assumption and Observation 5, all values in  $U_k$  are  $\langle \tau - 1, \Theta_{k+1}, V_{k+1} \rangle$ -permanent in  $r_{k+1}$ , and so all of  $U_{k+1}$  is  $\langle \tau - 1, \Theta_{k+1}, V_{k+1} \rangle$ -permanent in  $r_{k+1}$ , as needed. ◀

From Lemma 13, in  $r_N$  there is a set of  $N$   $(\tau - 1)$ -permanent values, inducing a cost of  $(\tau - 1)N$ . We use Claim 8 to increase the bound by  $2\tau - 1$  additional blocks.

**Proof (Theorem 12).** Construct  $r_N, V_N, U_N$ , and  $\Theta_N$  as in Lemma 13. Note that, since  $R - N \geq 1$ , there exists  $p \in \Pi \setminus \Theta_N$ . Since  $V_N \cap U_N = \emptyset$  and  $|V_N| \geq 2^D - LN > 2^D - L \frac{2^D - 1}{L} = 1$ ,  $V_N \setminus U_N$  contains at least two values. Extend  $r_N$  to  $r_{N+1}$  by invoking and returning **write**( $v$ ) and **write**( $v'$ ) for  $v, v' \in V_N \setminus U_N$ .

By Claim 8, there is a time  $t \geq t_{r_N}$  in  $r_{N+1}$  when there are  $2\tau - 1$  blocks in the shared storage with origin values of  $v$  or  $v'$ .  $U_N$  consists of  $N$  additional values that are  $\langle \tau - 1, \Theta_N, V_N \rangle$ -permanent in  $r_N$ , and since in  $r_{N+1} \setminus r_N$  writes are of values from  $V_N$  and no reader in  $\Theta_N$  takes steps, the values in  $U_N$  remain  $\langle \tau - 1, \Theta_N, V_N \rangle$ -permanent in  $r_{N+1}$ . By Observation 6, the storage cost is:

$$n \geq 2\tau - 1 + (\tau - 1)N = \tau + (\tau - 1)(N + 1) = \tau + (\tau - 1) \cdot \min\left(\left\lceil \frac{2^D - 1}{L} \right\rceil, R\right). \quad \blacktriangleleft$$

## 5 Lower bounds for common write disintegrated storage

While the results of the previous section hold a fortiori for  $\tau$ -common write algorithms, in this case we are able to show stronger results, independent of the local storage size. Intuitively, this is because readers can no longer reuse blocks they obtained from previous writes of



## 11:12 Integrated Bounds for Disintegrated Storage

the same value, and so we can prolong the execution that blows up the shared storage by rewriting values. Section 5.1 proves a general attribute of  $\tau$ -common write algorithms. We show in Section 5.2 that even with a single reader (and a single writer), if **reads** are invisible, then the required storage cost is at least  $\tau \cdot 2^D$ . In Section 5.3 we prove a bound for visible reads.

### 5.1 General observation

In this section we define a property that is a special case of  $k$ -permanence, which additionally requires that the set of labels associated with a **write** does not change.

► **Definition 14** (Constancy). Consider a finite run  $r$ ,  $k \in \mathbb{N}$ , a set  $S \subseteq \mathbb{V}$ , and a set of readers  $\Theta \subset \Pi$ . We say that a **write**  $w \in \mathbb{W}$  is  $\langle k, \Theta, S \rangle$ -constant in  $r$  if in every finite extension  $r'$  of  $r$  s.t. in  $r' \setminus r$  readers in  $\Theta$  do not take actions and **writes** are limited to values from  $S$ ,  $S$ -labels( $w, r', t_{r'}$ ) =  $S$ -labels( $w, r, t_r$ ) and  $|S$ -labels( $w, r', t_{r'}$ )| =  $k$ .

Similarly to Observation 7, it can be shown that:

► **Observation 15.** Consider  $V \subseteq \mathbb{V}$ ,  $k \in \mathbb{N}$ , and a finite run  $r$  with an invisible reader  $p \in \Pi$ . If  $w \in \mathbb{W}$  is  $\langle k, \{p\}, V \rangle$ -constant in  $r$  then  $w$  is  $\langle k, \emptyset, V \rangle$ -constant in  $r$ .

We next prove a stronger variant of Lemma 9 that allows us to add a permanent **write** to the shared storage while some set  $C \subseteq \mathbb{W}$  of **writes** are constant. Note that since the number of **writes** of a value  $v$  is infinite and the number of constant **writes** in a finite run is finite, for any non-empty  $V \subseteq \mathbb{V}$ ,  $\mathbb{W}_V \setminus C$  is non-empty.

► **Lemma 16.** Consider a non-empty set of values  $V \subseteq \mathbb{V}$ , a set of readers  $\Theta \subset \Pi$ , a reader  $p \in \Pi \setminus \Theta$ , and a finite run  $r$  of a wait-free regular  $\tau$ -common write algorithm. Let  $C$  be a set of **writes** that are  $\langle \tau - 1, \Theta, V \rangle$ -constant in  $r$ . Then there is an extension  $r'$  of  $r$  where some  $w \in \mathbb{W}_V \setminus C$  returns and is  $\langle \tau - 1, \Theta \cup \{p\}, V \rangle$ -permanent, and s.t. in  $r' \setminus r$  **writes** are limited to  $\mathbb{W}_V$  and readers in  $\Theta$  do not take actions.

**Proof.** Assume by contradiction that the lemma does not hold. We build an extension  $r'$  of  $r$  where a  $\text{read}_p$  operation includes infinitely many actions of  $p$  yet does not return. To this end, we show that the following property holds at specific times in  $r' \setminus r$ :

$$\psi(\hat{r}, t) \triangleq \forall w \in \text{writes}_p(\hat{r}, t) \cap \mathbb{W}_V : |\text{All-labels}_p(w, \hat{r}, t)| < \tau.$$

Note that, by definitions of  $\tau$ -common write and of *All-labels*, whenever  $\psi(r', t)$  holds, no pending  $\text{read}_p$  invocation can return a value  $v \in \text{values}_p(r', t) \cap V$ .

First, extend  $r$  to  $r_0$  by returning any pending  $\text{read}_p$  and **write**, invoking and returning a  $\text{write}(v_0)$  for some  $v_0 \in \mathbb{V}$  (the operations eventually return, by wait-freedom), and finally invoking a  $\text{read}_p$  operation  $rd$  without allowing it to take actions. We now prove by induction that for all  $k \in \mathbb{N}$ , there exists an extension  $r'$  of  $r_0$  where (1)  $\psi(r', t_{r'})$  holds, (2) no **write** is pending at  $t_{r'}$ , and in  $r' \setminus r$ : (3) **writes** are restricted to  $\mathbb{W}_V$ , (4)  $p$  performs  $k$  actions on objects after invoking  $rd$ , (5)  $rd$ 's return is not enabled, and (6) processes in  $\Theta$  do not take steps.

Base: for  $k = 0$ , consider  $r' = r_0$ . (2,4,6) hold trivially. (3) holds since the only **write** in  $r' \setminus r$  is  $w_0 \in \mathbb{W}_V$ . Since  $p$  performs no actions following the invocation of  $rd$ ,  $p.\text{data}(r', t_{r'})$  is empty. Therefore, (1)  $\psi(r', t_{r'})$  is vacuously true, and  $L\text{-labels}_p(w, r', t_{r'})$  is empty for all  $w \in \mathbb{W}_V$ , thus (5)  $rd$ 's return is not enabled by  $\tau$ -common write.



Step: assume inductively such an extension  $r_1$  of  $r_0$  with  $k \geq 0$  actions by  $p$  following  $rd$ 's invocation. Since  $rd$  cannot return, by wait-freedom, an action  $\mathbf{a}_p$  is enabled on some object. We construct an extension  $r_2$  of  $r_1$  by letting  $\mathbf{a}_p$  occur at time  $t_{r_1}$ . We then consider three cases:

1.  $p$  does not obtain a block with an origin **write** in  $\mathbb{W}_V \setminus \text{writes}_p(r_1, t_{r_1})$  at  $\mathbf{a}_p$ , thus  $(\text{writes}_p(r_2, t_{r_2}) \cap \mathbb{W}_V) \subseteq (\text{writes}_p(r_1, t_{r_1}) \cap \mathbb{W}_V)$ . Then, by Observation 1 and the inductive hypothesis, (1)  $\psi(r_2, t_{r_2})$  holds and thus, by  $\tau$ -common write,  $rd$  cannot return any value  $v \in \text{values}_p(r_2, t_{r_2}) \cap V$  at  $t_{r_2}$ . (5) It cannot return any other value in  $\text{values}_p(r_2, t_{r_2})$  by regularity, and  $r_2$  satisfies the induction hypothesis for  $k + 1$  as (2,3,4,6) trivially hold.

2.  $p$  obtains a block with origin **write**  $w' \in C \cap \mathbb{W}_V \setminus \text{writes}_p(r_1, t_{r_1})$  at  $\mathbf{a}_p$ . Then  $|L\text{-labels}_p(w', r_1, t_{r_1})| = 0$ . Since  $w'$  is  $\langle \tau - 1, \Theta, V \rangle$ -constant in  $r$  and in  $r_1 \setminus r$  **writes** are restricted to  $\mathbb{W}_V$  and processes in  $\Theta$  do not take steps (inductively), then by definition of constancy,  $|S\text{-labels}(w', r_1, t_{r_1})| = \tau - 1$ . By Observation 1, for all  $w \in \text{writes}_p(r_2, t_{r_2}) \cap \mathbb{W}_V$ :  $All\text{-labels}_p(w, r_2, t_{r_2}) \subseteq All\text{-labels}_p(w, r_1, t_{r_1})$ . Therefore  $|All\text{-labels}_p(w', r_2, t_{r_2})| \leq |L\text{-labels}_p(w', r_1, t_{r_1})| + |S\text{-labels}(w', r_1, t_{r_1})| = \tau - 1$ . Together with the inductive hypothesis,  $\forall w \in \text{writes}_p(r_2, t_{r_2}) \cap \mathbb{W}_V \setminus \{w'\}$ ,  $|All\text{-labels}_p(w, r_2, t_{r_2})| \leq |All\text{-labels}_p(w, r_1, t_{r_1})| < \tau$ ;  $\psi(r_2, t_{r_2})$  follows, thus (5) follows, and (2,3,4,6) trivially hold.

3.  $p$  obtains a block with origin **write**  $w' \in \mathbb{W}_V \setminus (\text{writes}_p(r_1, t_{r_1}) \cup C)$  at  $\mathbf{a}_p$ . Then  $|L\text{-labels}_p(w', r_2, t_{r_2})| = 1$  and the number of labels of other **writes** in  $\text{writes}_p(r_2, t_{r_2})$  does not increase following  $\mathbf{a}_p$ , thus  $rd$ 's return is not enabled at  $t_{r_2}$  by  $\tau$ -common write and regularity.

By the contradicting assumption,  $w'$  is not  $\langle \tau - 1, \Theta \cup \{p\}, V \rangle$ -permanent in  $r_2$ , thus there is an extension  $r_3$  of  $r_2$  s.t.  $|S\text{-labels}(w', r_3, t_{r_3})| < \tau - 1$  and in  $r_3 \setminus r_2$  **writes** are limited to  $\mathbb{W}_V$  and no readers in  $\Theta \cup \{p\}$  take steps (3,4,6 hold). We further extend  $r_3$  to  $r_4$  by letting any pending **write** return (2).

Let  $S = \text{writes}_p(r_2, t_{r_2}) \cap \mathbb{W}_V$ . Since every  $w \in S$  returns before  $t_{r_2}$  by the inductive assumption, the **writes** in  $r_4 \setminus r_2$  do not produce new labels associated with  $w$ . Since readers do not affect the sets  $S\text{-labels}$ , it follows that  $\forall w \in S : S\text{-labels}(w, r_4, t_{r_4}) \subseteq S\text{-labels}(w, r_3, t_{r_3}) \subseteq S\text{-labels}(w, r_2, t_{r_2})$ . Next,  $p$  takes no steps in  $r_4 \setminus r_2$  (4 holds), thus  $\forall w \in S : L\text{-labels}_p(w', r_4, t_{r_4}) = L\text{-labels}_p(w', r_2, t_{r_2})$ . It follows that:

$$|All\text{-labels}_p(w', r_4, t_{r_4})| \leq |L\text{-labels}_p(w', r_2, t_{r_2})| + |S\text{-labels}(w', r_3, t_{r_3})| < 1 + (\tau - 1) = \tau. \quad (3)$$

Moreover, by Observation 1 and the inductive assumption that  $\psi(r_1, t_{r_1})$  holds,

$$\forall w \in S \setminus \{w'\} : |All\text{-labels}_p(w, r_4, t_{r_4})| \leq |All\text{-labels}_p(w, r_1, t_{r_1})| < \tau. \quad (4)$$

From Equations 3 and 4, and since  $\text{writes}_p(r_4, t_{r_4}) \cap \mathbb{W}_V = \text{writes}_p(r_2, t_{r_2}) \cap \mathbb{W}_V = S$ , we get (1)  $\psi(r_4, t_{r_4})$ . Since  $rd$ 's return is not enabled at  $t_{r_2}$  and (4) it took no actions since, its return is not enabled anywhere in  $r_4 \setminus r_1$  (5), and we are done.  $\blacktriangleleft$

## 5.2 Invisible reads

We prove the following theorem by constructing a run with an exponential number of  $\tau$ -permanent values. The idea is to show that if there is a value in the domain for which there is no  $\tau$ -permanent **write**, then infinitely many **writes** remain  $(\tau - 1)$ -constant, which is of course impossible.

► **Theorem 17.** *The storage cost of a regular  $\tau$ -common write wait-free SRSW register emulation where **reads** are invisible is at least  $\tau \cdot 2^D$  blocks.*

## 11:14 Integrated Bounds for Disintegrated Storage

► **Lemma 18.** *Consider a non-empty set of values  $V \subseteq \mathbb{V}$  and a finite run  $r$ . Let  $C$  be a set of **writes** that are  $\langle \tau - 1, \emptyset, V \rangle$ -constant in  $r$ . Then there exists an extension  $r'$  of  $r$  where **writes** in  $r' \setminus r$  are limited to  $\mathbb{W}_V$ , and some  $w \in \mathbb{W}_V \setminus C$  is either  $\langle \tau - 1, \emptyset, V \rangle$ -constant or  $\langle \tau, \emptyset, V \rangle$ -permanent in  $r'$ .*

**Proof.** Let  $p \in \Pi$  be a reader. By Lemma 16, there is an extension  $r'$  of  $r$  where **writes** in  $r' \setminus r$  are limited to  $\mathbb{W}_V$  and some  $w \in \mathbb{W}_V \setminus C$  returns and is  $\langle \tau - 1, \{p\}, V \rangle$ -permanent. By Observation 7, if  $w$  is  $\langle \tau, \{p\}, V \rangle$ -permanent in  $r'$ , then  $w$  is  $\langle \tau, \emptyset, V \rangle$ -permanent in  $r'$  and the lemma follows. Otherwise, there exists an extension  $r''$  of  $r'$  where in  $r'' \setminus r'$  **writes** are limited to  $\mathbb{W}_V$  and  $p$  takes no steps, and  $|S\text{-labels}(w, r'', t_{r''})| < \tau$ . Since  $w$  is  $\langle \tau - 1, \{p\}, V \rangle$ -permanent in  $r'$ ,  $|S\text{-labels}(w, r'', t_{r''})| = \tau - 1$ .

We show that  $w$  is  $\langle \tau - 1, \emptyset, V \rangle$ -constant in  $r''$ . Consider an extension  $r'''$  of  $r''$  where **writes** are limited to values from  $V$  and  $p$  takes no steps in  $r''' \setminus r''$ . Since  $w$  has already returned by time  $t_{r''}$ , no new blocks with an origin **write** of  $w$  can be added to the shared storage in  $r'''$  after  $t_{r''}$ . It follows that  $S\text{-labels}(w, r''', t_{r'''}) \subseteq S\text{-labels}(w, r'', t_{r''})$ . However, since  $w$  is  $\langle \tau - 1, \{p\}, V \rangle$ -permanent in  $r'$ , and in  $r''' \setminus r'$  **writes** are limited  $\mathbb{W}_V$  and  $p$  takes no steps, then  $|S\text{-labels}(w, r''', t_{r'''})| \geq \tau - 1 = |S\text{-labels}(w, r'', t_{r''})|$ , yielding that  $S\text{-labels}(w, r''', t_{r'''}) = S\text{-labels}(w, r'', t_{r''})$ . Thus,  $w$  is  $\langle \tau - 1, \{p\}, V \rangle$ -constant in  $r''$ . The lemma follows from Observation 15. ◀

► **Claim 19.** *Consider a finite run  $r$  and a non-empty  $V \subseteq \mathbb{V}$ . Then there is an extension  $r'$  of  $r$  s.t. **writes** in  $r' \setminus r$  are limited to  $\mathbb{W}_V$ , and some  $w \in \mathbb{W}_V$  is  $\langle \tau, \emptyset, V \rangle$ -permanent in  $r'$ .*

**Proof.** Consider an algorithm with storage cost  $n$ , and let  $m = \lceil n/(\tau - 1) \rceil + 1$ . Assume by contradiction that the claim does not hold. We get a contradiction by constructing  $m + 1$  extensions of  $r$ ;  $r_0, \dots, r_m$  with sets of **writes**  $C_0 \subset C_1 \subset \dots \subset C_m \subseteq \mathbb{W}_v$  s.t. for all  $0 \leq k \leq m$ :

- (1) **writes** in  $r_k \setminus r$  are limited to  $\mathbb{W}_V$ , and
- (2)  $C_k$  is a set of  $k$  **writes** that are  $\langle \tau - 1, \emptyset, V \rangle$ -constant in  $r_k$ .

Note that in  $r_m$ ,  $\lceil \frac{n}{\tau - 1} \rceil + 1$  **writes** are  $\langle \tau - 1, \emptyset, V \rangle$ -constant, implying a storage cost greater than  $n$  by Observation 6, a contradiction.

The construction is by induction. The base case vacuously holds for  $r_0 = r$ ,  $C_0 = \emptyset$ . Assume inductively such  $r_k$  and  $C_k$  for  $k < m$ . By Lemma 18 there exists an extension  $r_{k+1}$  of  $r_k$  where some  $w \in \mathbb{W}_V \setminus C_k$  is either  $\langle \tau, \emptyset, V \rangle$ -permanent or  $\langle \tau - 1, \emptyset, V \rangle$ -constant, and **writes** in  $r_{k+1} \setminus r_k$  are limited to  $\mathbb{W}_V$ . Since all **writes** in  $C_k$  are  $\langle \tau - 1, \emptyset, V \rangle$ -constant in  $r_k$  they are also  $\langle \tau - 1, \emptyset, V \rangle$ -constant in  $r_{k+1}$ . By the contracting assumption,  $w$  is not  $\langle \tau, \emptyset, V \rangle$ -permanent in  $r_{k+1}$  thus it is  $\langle \tau - 1, \emptyset, V \rangle$ -constant in the run. Let  $C_{k+1} = C_k \cup \{w\}$ , therefore  $|C_{k+1}| = k + 1$  and all **writes** in  $C_{k+1}$  are  $\langle \tau - 1, \emptyset, V \rangle$ -constant in  $r_{k+1}$ , as needed. ◀

We are now ready to prove our lower bound of  $\tau \cdot 2^D$  blocks:

**Proof (Theorem 17).** We show that there exist  $2^D + 1$  finite runs  $r_0, r_1, \dots, r_{2^D}$  and sets of values  $V_0 \supset V_1 \supset \dots \supset V_{2^D}$  and  $U_0 \subset U_1 \subset \dots \subset U_{2^D}$ , such that for all  $0 \leq k \leq 2^D$ :

- (1)  $|V_k| = 2^D - k$ ,  $|U_k| = k$ ,  $V_k \cap U_k = \emptyset$ , and
- (2) all elements of  $U_k$  are  $\langle \tau, \emptyset, V_k \rangle$ -permanent in  $r_k$ .

By induction. Base:  $r_0$  is the empty run,  $V_0 = \mathbb{V}$ ,  $U_0 = \emptyset$ . Assume inductively such  $r_k$ ,  $V_k$ , and  $U_k$  for  $k < 2^D$ , and construct  $r_{k+1}$  as follows: first, because  $|V_k| = 2^D - k > 0$ , by Claim 19 there is an extension  $r_{k+1}$  of  $r_k$  where **writes** in  $r_{k+1} \setminus r_k$  are limited to  $\mathbb{W}_{V_k}$  and some  $w \in \mathbb{W}_{V_k}$  is  $\langle \tau, \emptyset, V_k \rangle$ -permanent.

Consider the value  $v \in V_k$  written by  $w$ . By Observation 5,  $v$  is  $\langle \tau, \emptyset, V_k \rangle$ -permanent in  $r_{k+1}$ . Let  $V_{k+1} = V_k \setminus \{v\}$ , then  $|V_{k+1}| = |V_k| - 1 = 2^D - (k+1)$ . Further let  $U_{k+1} = U_k \cup \{v\}$ . Note that, because  $V_k \cap U_k = \emptyset$ , we get  $v \notin U_k$  and hence  $V_{k+1} \cap U_{k+1} = \emptyset$  and  $|U_{k+1}| = |U_k| + 1 = k + 1$ . Since  $V_k \supset V_{k+1}$ , then  $v$  is  $\langle \tau, \emptyset, V_{k+1} \rangle$ -permanent. Additionally, **writes** in  $r_{k+1} \setminus r_k$  are from  $\mathbb{W}_{V_k}$ , thus by the inductive assumption and Observation 5, values in  $U_k$  are  $\langle \tau, \emptyset, V_{k+1} \rangle$ -permanent in  $r_{k+1}$ , and so all of  $U_{k+1}$  are  $\langle \tau, \emptyset, V_{k+1} \rangle$ -permanent in  $r_{k+1}$ .

Finally,  $U_{2^D}$  holds  $2^D$  values that are  $\langle \tau, \emptyset, \emptyset \rangle$ -permanent in  $r_{2^D}$ . By Observation 6:

$$n \geq \tau \cdot 2^D. \quad \blacktriangleleft$$

### 5.3 Visible reads

To prove a lower bound on the cost of systems with visible **reads**, we create a similar construction, except that the number of extensions might be limited by the number of readers,  $R$ . Instead, the bound depends on  $\min(2^D - 1, R)$ :

► **Theorem 20.** *The storage cost of a regular  $\tau$ -common write wait-free MRSW register emulation is at least  $\tau + (\tau - 1) \cdot \min(2^D - 1, R)$  blocks.*

Let  $N = \min(2^D - 1, R) - 1$ . We build a run with  $N$   $(\tau - 1)$ -permanent values:

► **Lemma 21.** *There exist finite runs  $r_0, r_1, \dots, r_N$ , sets of values  $V_0 \supset V_1 \supset \dots \supset V_N$  and  $U_0 \subset U_1 \subset \dots \subset U_N$ , and sets of readers  $\Theta_0 \subset \Theta_1 \subset \dots \subset \Theta_N$ , s.t. for all  $0 \leq k \leq N$ :*

- (1)  $|V_k| = 2^D - k$ ,  $|U_k| = |\Theta_k| = k$ ,  $V_k \cap U_k = \emptyset$ , and
- (2) all elements of  $U_k$  are  $\langle \tau - 1, \Theta_k, V_k \rangle$ -permanent in  $r_k$ .

**Proof.** By induction. Base:  $r_0$  is the empty run,  $V_0 = \mathbb{V}$ ,  $\Theta_0 = U_0 = \emptyset$ . Assume inductively such  $r_k, V_k, U_k$ , and  $\Theta_k$  for  $k < N$ , and construct  $r_{k+1}$  as follows: since  $R - |\Theta_k| > 0$ , there is a reader  $p \in \Pi \setminus \Theta_k$ . Moreover,  $|V_k| > 2^D - N > 0$ . Therefore, by Lemma 16, there is an extension  $r_{k+1}$  of  $r_k$  where **writes** in  $r_{k+1} \setminus r_k$  are limited to  $\mathbb{W}_{V_k}$ , readers in  $\Theta_k$  do not take steps in  $r_{k+1} \setminus r_k$ , and some  $w \in W_{V_k}$  returns and is  $\langle \tau - 1, \Theta_k \cup \{p\}, V_k \rangle$ -permanent in  $r_{k+1}$ .

Let  $\Theta_{k+1} = \Theta_k \cup \{p\}$ , and consider the value  $v \in V_k$  written by  $w$ . By Observation 5,  $v$  is  $\langle \tau - 1, \Theta_{k+1}, V_k \rangle$ -permanent. Let  $V_{k+1} = V_k \setminus \{v\}$ , then  $|V_{k+1}| = 2^D - (k+1)$ . Further let  $U_{k+1} = U_k \cup \{v\}$ . Since  $V_k \cap U_k = \emptyset$ , we get that  $V_{k+1} \cap U_{k+1} = \emptyset$  and  $|U_{k+1}| = k + 1$ .

Since  $V_k \supset V_{k+1}$ ,  $v$  is  $\langle \tau - 1, \Theta_{k+1}, V_{k+1} \rangle$ -permanent. In addition, in  $r_{k+1} \setminus r_k$  **writes** are limited to  $\mathbb{W}_{V_k}$  and readers in  $\Theta_k$  do not take steps, and since  $\Theta_k \subset \Theta_{k+1}$ , then by the inductive assumption and Observation 5, all values in  $U_k$  are  $\langle \tau - 1, \Theta_{k+1}, V_{k+1} \rangle$ -permanent. Therefore all elements of  $U_{k+1}$  are  $\langle \tau - 1, \Theta_{k+1}, V_{k+1} \rangle$ -permanent in  $r_{k+1}$ , as needed. ◀

From Lemma 21, in  $r_N$  there is a set of  $N$   $(\tau - 1)$ -permanent values, inducing a cost of  $(\tau - 1)N$ . We use Claim 8 to increase the bound by  $2\tau - 1$  additional blocks.

**Proof (Theorem 20).** Construct  $r_N, V_N, U_N$ , and  $\Theta_N$  as in Lemma 21. Note that, since  $R - N \geq 1$ , there is a reader  $p \in \Pi \setminus \Theta_N$ . Since  $V_N \cap U_N = \emptyset$  and  $|V_N| = 2^D - N = 2^D - (\min(2^D - 1, R) - 1) \geq 2$ ,  $V_N$  contains two values, and they are not in  $U_N$ . Extend  $r_N$  to  $r_{N+1}$  by invoking and returning **write**( $v$ ) and **write**( $v'$ ) for  $v, v' \in V_N$ .

By Claim 8, there is a time  $t \geq t_{r_N}$  in  $r_{N+1}$  when there are  $2\tau - 1$  blocks in the shared storage with origin values of  $v$  or  $v'$ . In addition,  $U_N$  consists of  $N$  values that are  $\langle \tau - 1, \Theta_N, V_N \rangle$ -permanent in  $r_N$ , and since in  $r_{N+1} \setminus r_N$  **writes** are of values from  $V_N$  and no reader in  $\Theta_N$  takes steps, the values in  $U_N$  remain  $\langle \tau - 1, \Theta_N, V_N \rangle$ -permanent in  $r_{N+1}$ . By Observation 6, the storage cost amounts to at least:

$$n \geq 2\tau - 1 + (\tau - 1)N = \tau + (\tau - 1)(N + 1) = \tau + (\tau - 1) \cdot \min(2^D - 1, R). \quad \blacktriangleleft$$

## 6 Discussion

We have shown lower bounds on the space complexity of regular wait-free  $\tau$ -disintegrated storage algorithms. Although our bounds are stated in terms of blocks, there are scenarios where they entail concrete bounds in terms of bits. In replication, each block stores an entire value, thus the block sizes are  $D$  bits. Other applications use symmetric coding where all blocks are of equal size. Using a simple pigeonhole argument, it can be shown that in  $\tau$ -disintegrated storage emulations that use symmetric coding and that are not  $(\tau + 1)$ -disintegrated, the size of blocks is at least  $D/\tau$  bits, yielding bounds of  $D \cdot 2^D$  and  $D + D \frac{\tau-1}{\tau} \cdot \min(2^D - 1, R)$  with invisible and visible readers, respectively.

Our lower bounds for the common write case explain, for the first time, why previous coded storage algorithms have either had the readers write or consumed exponential (or even unbounded) space. Similarly, they establish why previous emulations of large registers from smaller ones have either had the readers write, had the writer share blocks among different writes, or consumed exponential space.

Our work leaves several open questions. First, when replication is used as a means to overcome Byzantine faults or data corruption, our results suggest that there might be an interesting trade-off between the shared storage cost and the size of local memory at the readers, and a possible advantage to systems that apply replication rather than error correction codes: we have shown that, with invisible readers, the former require  $\Omega(2^D/L)$  blocks, rather than the  $\Omega(2^D)$  blocks needed by the latter. Whether there are algorithms that achieve this lower cost remains an open question. Second, it is unclear how the bounds would be affected by removing our assumption that each block in the shared storage pertains to a single write. Wei [25] has provided a partial answer to this questions by showing that similar bounds hold without this assumption, but only in the case of emulating large registers from smaller ones *without* meta-data at all. Similarly, it would be interesting to study whether allowing readers to write data (and not only signals) impacts the storage cost. Finally, future work may consider additional sub-classes of disintegrated storage, e.g., with unresponsive objects, and show that additional costs are incurred in these cases.

---

## References

- 1 Ittai Abraham, Gregory Chockler, Idit Keidar, and Dahlia Malkhi. Byzantine disk paxos: optimal resilience with byzantine shared memory. *Distributed Computing*, 18(5):387–408, 2006.
- 2 Ittai Abraham, Gregory Chockler, Idit Keidar, and Dahlia Malkhi. Wait-free regular storage from byzantine components. *Information Processing Letters*, 101(2):60–65, 2007.
- 3 Marcos K. Aguilera, Burkhard Englert, and Eli Gafni. On using network attached disks as shared memory. In *Proceedings of the Twenty-second Annual Symposium on Principles of Distributed Computing*, PODC '03, pages 315–324, New York, NY, USA, 2003. ACM. doi:10.1145/872035.872082.
- 4 Marcos Kawazoe Aguilera, Ramaprabhu Janakiraman, and Lihao Xu. Using erasure codes efficiently for storage in a distributed system. In *2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 336–345, June 2005.
- 5 Elli Androulaki, Christian Cachin, Dan Dobre, and Marko Vukolić. Erasure-coded byzantine storage with separate metadata. In *International Conference on Principles of Distributed Systems*, pages 76–90. Springer, 2014.
- 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. doi:10.1145/200836.200869.

- 7 Rida A Bazzi and Yin Ding. Non-skipping timestamps for byzantine data storage systems. In *International Symposium on Distributed Computing*, pages 405–419. Springer, 2004.
- 8 Alon Berger, Idit Keidar, and Alexander Spiegelman. Integrated bounds for disintegrated storage. *arXiv preprint arXiv:1805.06265*, 2018.
- 9 Christian Cachin and Stefano Tessaro. Optimal resilience for erasure-coded byzantine distributed storage. In *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*, pages 115–124. IEEE, 2006.
- 10 Viveck R. Cadambe, Nancy Lynch, Muriel Medard, and Peter Musial. A coded shared atomic memory algorithm for message passing architectures. In *Network Computing and Applications (NCA), 2014 IEEE 13th International Symposium on*, pages 253–260. IEEE, 2014.
- 11 Viveck R. Cadambe, Zhiying Wang, and Nancy Lynch. Information-theoretic lower bounds on the storage cost of shared memory emulation. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, PODC '16, pages 305–313, New York, NY, USA, 2016. ACM. doi:10.1145/2933057.2933118.
- 12 Soma Chaudhuri, Martha J Kosa, and Jennifer L Welch. One-write algorithms for multi-valued regular and atomic registers. *Acta Informatica*, 37(3):161–192, 2000.
- 13 Tian Ze Chen and Yuanhao Wei. Step Optimal Implementations of Large Single-Writer Registers. 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 32:1–32:16, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.OPODIS.2016.32.
- 14 Gregory Chockler, Rachid Guerraoui, and Idit Keidar. Amnesic distributed storage. In *Distributed Computing*, pages 139–151. Springer, 2007.
- 15 Gregory Chockler and Alexander Spiegelman. Space complexity of fault-tolerant register emulations. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, PODC '17, pages 83–92, New York, NY, USA, 2017. ACM. doi:10.1145/3087801.3087824.
- 16 Dan Dobre, Ghassan Karame, Wenting Li, Matthias Majuntke, Neeraj Suri, and Marko Vukolić. Powerstore: proofs of writing for efficient and robust storage. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 285–298. ACM, 2013.
- 17 Partha Dutta, Rachid Guerraoui, and Ron R. Levy. Optimistic erasure-coded distributed storage. In *Proceedings of the 22nd International Symposium on Distributed Computing*, DISC '08, pages 182–196, Berlin, Heidelberg, 2008. Springer-Verlag. doi:10.1007/978-3-540-87779-0\_13.
- 18 Garth R Goodson, Jay J Wylie, Gregory R Ganger, and Michael K Reiter. Efficient byzantine-tolerant erasure-coded storage. In *Dependable Systems and Networks, 2004 International Conference on*, pages 135–144. IEEE, 2004.
- 19 Prasad Jayanti, Tushar Deepak Chandra, and Sam Toueg. Fault-tolerant wait-free shared objects. *Journal of the ACM (JACM)*, 45(3):451–500, 1998.
- 20 Leslie Lamport. On interprocess communication. *Distributed computing*, 1(2):86–101, 1986.
- 21 Jean-Philippe Martin, Lorenzo Alvisi, and Michael Dahlin. Minimal byzantine storage. In *International Symposium on Distributed Computing*, pages 311–325. Springer, 2002.
- 22 Gary L Peterson. Concurrent reading while writing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(1):46–55, 1983.
- 23 Alexander Spiegelman, Yuval Cassuto, Gregory Chockler, and Idit Keidar. Space bounds for reliable storage: Fundamental limits of coding. In *Proceedings of the 2016 ACM Symposium*

## 11:18 Integrated Bounds for Disintegrated Storage

*on Principles of Distributed Computing*, PODC '16, pages 249–258, New York, NY, USA, 2016. ACM. doi:10.1145/2933057.2933104.

- 24 Zhiying Wang and Viveck R. Cadambe. On multi-version coding for distributed storage. In *Communication, Control, and Computing (Allerton), 2014 52nd Annual Allerton Conference on*, pages 569–575. IEEE, 2014.
- 25 Yuanhao Wei. Space complexity of implementing large shared registers. *arXiv preprint arXiv:1808.00481*, 2018.

# Distributed Recoloring

**Marthe Bonamy**

CNRS, LaBRI, Université de Bordeaux, France  
marthe.bonamy@u-bordeaux.fr

**Paul Ouvrard**

LaBRI, CNRS, Université de Bordeaux, France  
paul.ouvrard@u-bordeaux.fr

**Mikaël Rabie**

Aalto University, Finland  
mikael.rabie@aalto.fi

**Jukka Suomela**

Aalto University, Finland  
jukka.suomela@aalto.fi

**Jara Uitto**

ETH Zürich, Switzerland; and University of Freiburg, Germany  
jara.uitto@inf.ethz.ch

---

## Abstract

Given two colorings of a graph, we consider the following problem: can we recolor the graph from one coloring to the other through a series of elementary changes, such that the graph is properly colored after each step?

We introduce the notion of *distributed recoloring*: The input graph represents a network of computers that needs to be recolored. Initially, each node is aware of its own input color and target color. The nodes can exchange messages with each other, and eventually each node has to stop and output its own recoloring schedule, indicating when and how the node changes its color. The recoloring schedules have to be globally consistent so that the graph remains properly colored at each point, and we require that adjacent nodes do not change their colors simultaneously.

We are interested in the following questions: How many communication rounds are needed (in the deterministic LOCAL model of distributed computing) to find a recoloring schedule? What is the length of the recoloring schedule? And how does the picture change if we can use *extra colors* to make recoloring easier?

The main contributions of this work are related to distributed recoloring with one extra color in the following graph classes: trees, 3-regular graphs, and toroidal grids.

**2012 ACM Subject Classification** Theory of computation → Distributed computing models, Theory of computation → Graph algorithms analysis

**Keywords and phrases** Distributed Systems, Graph Algorithms, Local Computations

**Digital Object Identifier** 10.4230/LIPIcs.DISC.2018.12

**Related Version** The full version of this work is available at arXiv:1802.06742.

**Funding** This work was supported in part by ERC Grant No. 336495 (ACDC) and ANR project DISTANCIA (ANR-17-CE40-0015).

**Acknowledgements** The authors would like to thank Nicolas Bousquet for helpful discussions regarding the proof of Lemma 14.



© Marthe Bonamy, Paul Ouvrard, Mikaël Rabie, Jukka Suomela, and Jara Uitto; licensed under Creative Commons License CC-BY

32nd International Symposium on Distributed Computing (DISC 2018).

Editors: Ulrich Schmid and Josef Widder; Article No. 12; pp. 12:1–12:17

Leibniz International Proceedings in Informatics



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



## 1 Introduction

In classical graph problems, we are given a graph and the task is to *find* a feasible solution. In *reconfiguration problems*, we are given two feasible solutions – an input configuration and a target configuration – and the task is to find a sequence of moves that turns the input configuration into the target configuration.

**Recoloring problems.** Perhaps the most natural example of a reconfiguration problem is *recoloring*: we are given a graph  $G$  and two proper  $k$ -colorings of  $G$ , let us call them  $s$  and  $t$ , and the task is to find a way to turn  $s$  into  $t$  by changing the color of one node at a time, such that each intermediate step is a proper coloring. More formally, the task is to find a sequence of proper  $k$ -colorings  $x_0, x_1, \dots, x_L$  such that  $x_0 = s$  and  $x_L = t$ , and  $x_{i-1}$  and  $x_i$  differ only at one node. Such problems have been studied extensively from the perspective of graph theory and classical centralized algorithms, but the problems are typically inherently *global* and solutions are long, i.e.,  $L$  is large in the worst case.

In this work we introduce recoloring problems in a *distributed* setting. We show that there are natural relaxations of the problem that are attractive from the perspective of distributed graph algorithms: they admit solutions that are short and that can be found *locally* (e.g., in sublinear number of rounds). Distributed recoloring problems are closely related to classical symmetry-breaking problems that have been extensively studied in the area of distributed graph algorithms, but as we will see, they also introduce new kinds of challenges.

**Distributed recoloring.** We will work in the usual LOCAL model of distributed computing: Each node  $v \in V$  of the input graph  $G = (V, E)$  is a computer, and each edge  $e \in E$  represents a communication link between two computers. Computation proceeds in synchronous rounds: each node sends a message to each of its neighbors, receives a message from each of its neighbors, and updates its local state. Eventually, all nodes have to announce their local outputs and stop; the running time of the algorithm is the number of communication rounds until all nodes stop. We assume that the algorithm is deterministic, and each node is labeled with a unique identifier.

In *distributed recoloring*, each node  $v \in V$  is given two colors, an *input color*  $s(v)$  and a *target color*  $t(v)$ . It is guaranteed that both  $s$  and  $t$  form a proper coloring of  $G$ , that is,  $s(u) \neq s(v)$  and  $t(u) \neq t(v)$  for all  $\{u, v\} \in E$ . Each node  $v \in V$  has to output a finite *recoloring schedule*  $x(v) = (x_0(v), x_1(v), \dots, x_\ell(v))$  for some  $\ell = \ell(v)$ . For convenience, we define  $x_i(v) = x_\ell(v)$  for  $i > \ell(v)$ . We say that the node *changes its color at time*  $i > 0$  if  $x_{i-1}(v) \neq x_i(v)$ ; let  $C_i$  be the set of nodes that change their color at time  $i$ . Define  $L = \max_v \ell(v)$ ; we call  $L$  the *length* of the solution. A solution is feasible if the following holds:

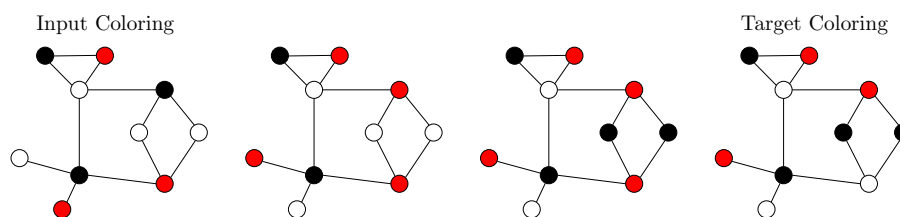
1.  $x_0 = s$  and  $x_L = t$ ,
2.  $x_i$  is a proper coloring of  $G$  for all  $i$ ,
3.  $C_i$  is an independent set of  $G$  for all  $i$ .

The key differences between distributed recoloring and classical recoloring are:

1. Input and output are given in a distributed manner: no node knows everything about  $G$ ,  $s$ , and  $t$ , and no node needs to know everything about  $x_i$  or the length of the solution  $L$ .
2. We do not require that only one node changes its color; it is sufficient that adjacent nodes do not change their colors simultaneously.

See Figure 1 for a simple example of distributed recoloring steps.





■ **Figure 1** Distributed recoloring: the input coloring  $s$  can be seen on the left and the target coloring  $t$  on the very right. The illustration shows one possible way to reach the target coloring in three steps by, in each step, changing the colors of an independent set of nodes.

Note that a solution to distributed recoloring is locally checkable in the following sense: to check that a solution is feasible, it is enough to check independently for each edge  $\{u, v\} \in E$  that the recoloring sequences  $x(u)$  and  $x(v)$  are compatible with each other, and for each node  $v \in V$  that  $x(v)$  agrees with  $s(v)$  and  $t(v)$ . However, distributed recoloring is not necessarily an LCL problem [21] in the formal sense, as the length of the output per node is not a priori bounded.

We emphasize that we keep the following aspects well-separated: what is the complexity of *finding* the schedule, and how *long* the schedules are. Hence it makes sense to ask, e.g., if it is possible to find a schedule of length  $O(1)$  in  $O(\log n)$  rounds (note that the physical reconfiguration of the color of the node may be much slower than communication and computation).

**Recoloring with extra colors.** Recoloring is computationally very hard, as solutions do not always exist, and deciding whether a solution exists is PSPACE-hard. It is in a sense analogous to problems such as finding an *optimal* node coloring of a given graph; such problems are not particularly interesting in the LOCAL model, as the complexity is trivially global. To make the problem much more interesting we slightly relax it.

We define a  $k + c$  recoloring problem (a.k.a.  $k$ -recoloring with  $c$  extra colors) as follows:

- We are given colorings with  $s(v), t(v) \in [k]$ .
- All intermediate solutions must satisfy  $x_i(v) \in [k + c]$ .

Here we use the notation  $[n] = \{1, 2, \dots, n\}$ .

The problem of  $k + c$  recoloring is meaningful also beyond the specific setting of distributed recoloring. For example, here is an example of a very simple observation:

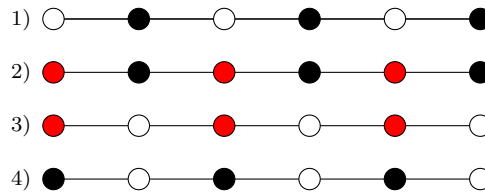
► **Lemma 1.** *Recoloring with 1 extra color is always possible in any bipartite graph, with a distributed schedule of length  $L = 3$ .*

**Proof.** Let the bipartition be  $V = V_1 \cup V_2$ . First each node  $v \in V_1$  switches to  $k + 1$ , then each  $v \in V_2$  switches to color  $t(v)$ , and finally each  $v \in V_1$  switches to color  $t(v)$ . ◀

Incidentally, it is easy to extend this result to show that  $k$ -recoloring with  $c = \chi - 1$  extra colors is always possible with a schedule of length  $O(c)$  in a graph with chromatic number  $\chi$ , and in particular  $k$ -recoloring with  $c = k - 1$  extra colors is trivial. Figure 2 gives an illustration of recoloring a bipartite graph with one extra color.

As a corollary, we can solve distributed  $k + 1$  recoloring in trees in  $O(n)$  rounds, with a schedule of length  $O(1)$ : simply find a bipartition and apply the above lemma. However, is this optimal? Clearly finding a bipartition in a tree requires  $\Omega(n)$  rounds, but can we solve recoloring with 1 extra color strictly faster?

These are examples of problems that we study in this work. We initiate the study of distributed complexity of recoloring, with the ultimate objective of finding a complete



■ **Figure 2** In the input graph, a bipartition is given. Therefore, the target coloring can be reached by using one extra color in three steps.

characterization of graph families and parameters  $k$ ,  $c$ , and  $L$  such that distributed  $k + c$  recoloring with schedules of length  $L$  can be solved efficiently in a distributed setting.

As we will see, the problem turns out to be surprisingly rich already in very restricted settings such as grids or 3-regular trees. Many of the standard lower bound techniques fail; in particular, known results on the hardness of graph coloring do not help here, as we are already given two proper colorings of the input graph.

**Contributions.** Our main contribution is a comprehensive study of the complexity of distributed recoloring in various graph families; the results are summarized in Tables 1–5. The highlights of this work are the following results:

1. **An algorithm for 3 + 1 recoloring on trees.** On trees, 3-recoloring is inherently global: it is easy to see that the worst-case running time is  $\Theta(n)$  and the worst-case schedule length is  $\Theta(n)$ . With one extra color, we can trivially find a schedule of length  $O(1)$  in time  $O(n)$ . However, we show that we can do much better: it is possible to find a schedule of length  $O(1)$  in time  $O(\log n)$ .

Here the key component is a new algorithm that solves the following sub-problem in  $O(\log n)$  rounds: given a tree, find an independent set  $I$  such that the removal of  $I$  splits the tree in components of size 1 or 2. This subroutine may find applications in other contexts as well.

These results are presented in Section 5.

2. **An algorithm for 3 + 1 recoloring for graphs of degree at most 3.** In general graphs, 3 + 1 recoloring is not necessarily possible; we can construct a small 4-regular graph in which 3 + 1 recoloring is not solvable. However, we will show that if the maximum degree of the graph is at most 3 (i.e., we have a *subcubic* graph), 3 + 1 recoloring is always possible. Moreover, we can find a schedule of length  $O(\log n)$  in time  $\text{polylog}(n)$ .

This result is presented in Section 6.

3. **Complexity of 3 + 1 recoloring on toroidal grids.** We also give a complete characterization of 3 + 1 recoloring in one particularly interesting family of 4-regular graphs: 2-dimensional toroidal grids (a.k.a. torus grid graphs, Cartesian graph products of two cycles). While the case of 1-dimensional grids (cycles) is easy to characterize completely, the case of 2-dimensional grids turns out to be much more interesting.

Here our main contribution is the following graph-theoretic result: in an  $h \times w$  toroidal grid, 3 + 1 recoloring is possible for any input if and only if (i) both  $h$  and  $w$  are even, or (ii)  $h = 4$ , or (iii)  $w = 4$ . In all other cases we can find 3-colorings  $s$  and  $t$  such that  $t$  is not reachable from  $s$  even if we can use 1 extra color.

As a simple corollary, 3 + 1 recoloring is inherently global from the perspective of distributed computing, and it takes  $\Theta(n)$  rounds to solve even if we have the promise that e.g.  $h$  and  $w$  are even (and hence a schedule of length  $\Theta(1)$  trivially exists).

This result is presented in Section 7.

Additionally, several simple upper and lower bounds and corollaries are given in Sections 4 and 8 and in the full version of the article.

**Motivation.** As a simple application scenario, consider the task of reconfiguring a system of unmanned aerial vehicles. Here each node is an aircraft, the color corresponds to an altitude range, and an edge corresponds to a pair of aircraft whose paths might cross and hence need to be kept at different cruising altitudes to avoid collisions.

For each aircraft there are designated areas in which they can safely change their altitude. To reconfigure the entire system, we could take all aircraft to these areas simultaneously. However, this may be a costly maneuver.

Another possibility is to reserve a longer timespan during which a set  $X$  of aircraft may change their altitudes, whenever they happen to be at convenient locations. Now if we let two aircraft  $u, v \in X$  change their altitudes during the same timespan, we need to ensure that any intermediate configuration is safe, regardless of whether  $u$  or  $v$  happens to change its altitude first. Furthermore, we would like to complete reconfiguration in minimal time (short schedule), and we would like to waste precious airspace as little as possible and hence keep as few altitude levels as possible in reserve for reconfiguration (few extra colors).

This scenario – as well as many similar scenarios, such as the task of reconfiguring the frequency bands of radio transmitters in a manner that never causes interference, even if the clocks are not perfectly synchronized – give rise to the following variant of distributed recoloring that we call *weak recoloring*: if two adjacent nodes  $u$  and  $v$  change their color simultaneously at time  $i$ , then  $\{x_{i-1}(u), x_i(u)\} \cap \{x_{i-1}(v), x_i(v)\} = \emptyset$ , that is, we have a proper coloring regardless of whether  $u$  or  $v$  changes its color first.

Let us now contrast weak recoloring with *strong recoloring*, in which adjacent nodes never change colors simultaneously. Trivially, strong recoloring solves weak recoloring. But the converse is also true up to constant factors: if we have  $k$  input colors and a solution to weak recoloring of length  $L$ , then we can also find a solution to strong recoloring of length  $kL$ . To see this, we can implement one weak recoloring step in  $k$  strong recoloring substeps such that in substep  $j$  nodes of input color  $j$  change their colors.

As our focus is on the case of a small number of input colors, we can equally well study strong or weak recoloring here; all of our results hold for either of them. While weak recoloring is closer to applications, we present our results using strong recoloring, as it has a more convenient definition.

## 2 Related work

**Reconfiguration and recoloring.** Recoloring, and more generally combinatorial reconfiguration has received attention over the past few years. Combinatorial reconfiguration problems consist of finding step-by-step transformations between two feasible solutions such that all intermediate results are also feasible. They model dynamic situations where a given solution is in place and has to be modified, but no disruption can be afforded. We refer the reader to the nice survey [24] for a full overview, and focus here on node coloring as a reference problem.

As mentioned earlier, we introduce distributed recoloring here, but centralized recoloring has been studied extensively before. Two main models are considered:

1. *Node recoloring*: at each step, we can recolor a node into a new color that does not appear on its neighborhood
2. *Kempe recoloring*: at each step, we can switch the colors in a bichromatic component (we operate a Kempe change).

The usual questions are of the form: Given a graph  $G$  and an integer  $k$ , are all its  $k$ -colorings equivalent (up to node or Kempe recolorings)? What is the complexity of deciding that? What is the maximum number of operations needed to go from the other?

All of those questions can also be asked for two specific  $k$ -colorings  $s$  and  $t$  of  $G$ . Are they equivalent (up to node or Kempe recolorings)? What is the complexity of deciding that? What is the maximum number of operations needed to go from  $s$  to  $t$  in  $G$ ?

While the complexity of questions related to Kempe recoloring remains elusive, the problems related to node recoloring are typically PSPACE-hard [6]. The related question of deciding equivalence when a bound on the length of an eligible recoloring sequence is given as part of the input has also been considered [7]. We know that the maximum number of operations needed to go from one 3-coloring to another in a tree is  $\Theta(n)$  [11]. While  $(\Delta + 1)$ -recoloring a graph with no node of degree more than  $\Delta$  is not always possible, having  $\Delta + 2$  colors always suffices [16], and there are also meaningful results to obtain for the problem of  $(\Delta + 1)$ -recoloring [14]. Two other settings have received special attention: characterizing fully when 3-recoloring is possible [11, 10], and guaranteeing short reconfiguration sequences in the case of sparse graphs for various notions of sparse [4, 8].

Kempe changes were introduced in 1879 by Kempe in his attempted proof of the Four Color Theorem [17]. Though this proof was fallacious, the Kempe change technique has proved useful in, for example, the proof of the Five Color Theorem and a short proof of Brooks' Theorem. Most works on the topic initially focused on planar graphs, but significant progress was recently obtained in more general settings. We know that all  $k$ -colorings of a graph with no node of degree more than  $k$  are equivalent (w.r.t. Kempe changes), except in the case of one very specific graph: the 3-prism [5, 15, 18].

Note that some other variants have also been studied, perhaps most notably the question of how many nodes to recolor at once so that the graph can be recolored [19].

While we will not discuss Kempe recoloring in our work, we point out that recoloring with extra colors is closely connected to Kempe recoloring: Kempe recolorability implies recolorability with one extra color (while the converse is not true). Hence the negative results related to one extra color also hold for Kempe recoloring.

**Distributed graph coloring.** Panconesi and Srinivasan [23] have used Kempe operations to design efficient distributed algorithms for graph coloring with  $\Delta$  colors. Other than that we are not aware of prior work on distributed recoloring. On the other hand, the literature on the standard distributed coloring is vast. The best overview on the topic is the book by Barenboim and Elkin [3]; the most important recent developments include the following results. There is a randomized  $O(\log^* n + 2\sqrt{\log \log n})$ -time algorithm for  $(\Delta + 1)$ -coloring by Chang et al. [13]. In the case of trees, the number of colors can be reduced to  $\Delta$  with the cost of increasing the runtime to  $O(\log_{\Delta} \log n)$  [12]. On the deterministic side, the best known  $(\Delta + 1)$ -coloring algorithm requires  $O(\Delta^{3/4} \log \Delta + \log^* n)$  communication rounds [2]. In the case of trees, the *rake-and-compress* -method by Miller and Reif gives a 3-coloring in time  $O(\log n)$  [20].

However, there seems to be surprisingly little technology that one can directly transfer between the coloring domain and recoloring domain. Toroidal grids are a good example: by prior work [9], 3-coloring is an inherently global problem, and by the present work,  $3 + 1$  recoloring is an inherently global problem, but the arguments that are used in these proofs are very different (despite the fact that both of them are related to the idea that a “parity” is preserved).

### 3 Preliminaries

In this article, each graph  $G = (V, E)$  is a simple undirected graph where  $V$  represents its node set and  $E$  its edge set. For a subset of nodes  $S \subseteq V$ , we denote by  $G[S]$  the subgraph induced by  $S$ . For a node  $u \in V$ , we denote by  $N(u)$  the *open neighborhood* of  $u$  that is the set of all the neighbors of  $u$  and by  $N[u]$  its *closed neighborhood* i.e. the set  $N(u) \cup \{u\}$ . For a subset  $S \subseteq V$ , its closed neighborhood corresponds to the set  $\bigcup_{u \in S} N[u]$ .

The *degree* of a node is the number of neighbors. A *k-regular graph* is a graph in which all nodes have degree  $k$ , a *cubic graph* is the same thing as a 3-regular graph, and a *subcubic graph* is a graph in which all nodes have degree at most 3. A *tree* is a connected acyclic graph, and a *k-regular tree* is a tree in which each node has degree 1 or  $k$ .

A *maximal independent set* (MIS)  $S \subseteq V$  is an independent set (i.e. a set of pairwise non-adjacent nodes) such that for each non-MIS node  $u \notin S$ ,  $N(u) \cap S \neq \emptyset$ .

Given a graph  $G = (V, E)$ , a *list-assignment* is a function which assigns to each node  $v \in V$  a list of colors  $L(v)$ . An *L-coloring* of  $G$  is a function  $c$  that assigns to each node  $v \in V$  a color  $c(v) \in L(v)$  such that for any two adjacent nodes  $u, v \in V$ , we have  $c(u) \neq c(v)$ . A graph  $G$  is *k-list-colorable* if it admits an *L-coloring* for every list-assignment where the list of each node is of size at least  $k$ . Therefore, list-coloring generalizes node-coloring if we consider the special case where each node receives the same input list. The notion of *L-recoloring* is the natural generalization of *k-recoloring*: the same elementary steps are considered, and every intermediate coloring must be an *L-coloring*.

In order to output a recoloring schedule, it is convenient to consider the question of recoloring a graph  $G$  from a coloring  $s$  to a coloring  $t$ , rather than the more symmetric question of whether the two colorings are equivalent in the given setting. We take this opportunity to note that we can reverse time and hence recoloring schedule from  $s$  to  $t$  also yields a recoloring schedule from  $t$  to  $s$ . In the rest of the paper, we therefore address the two questions as one.

### 4 Simple upper bounds

We will start by providing some simple upper bounds of recoloring problems:

► **Lemma 2.** *In any graph,  $k + c$  recoloring for  $c = k - 1$  is possible in 0 communication rounds, with a schedule of length  $O(k)$ .*

**Proof.** Generalize the idea of Lemma 1; note that the schedule of node  $v$  depends only on  $s(v)$  and  $t(v)$ , and not on the colors of any other node around it. ◀

► **Lemma 3.** *Let  $G$  be a graph of maximum degree at most  $\Delta$ , and let  $k \geq \Delta + 2$ . Then  $k$ -recoloring with  $c$  extra colors is at least as easy as  $(k - 1)$ -recoloring with  $c + 1$  extra colors.*

**Proof.** Given a  $k$ -coloring  $x$ , we can construct a  $(k - 1)$ -coloring  $x'$  as follows: all nodes of color  $k$  pick a new color from  $\{1, 2, \dots, k - 1\}$  that is not used by any of their neighbors. Note that  $x \rightarrow x'$  is a valid step in distributed recoloring (nodes of color  $k$  form an independent set), and by reversing the time, also  $x' \rightarrow x$  is a valid step.

Hence to recolor  $s \rightarrow t$  with  $c$  extra colors, it is sufficient to recolor  $s' \rightarrow t'$  with  $c + 1$  extra colors (color  $k$  no longer appears in the input and target colorings and can be used as an auxiliary color during recoloring). Then we can put everything together to form a recoloring schedule  $s \rightarrow s' \rightarrow t' \rightarrow t$ , with only constant overhead in the running time and schedule length. ◀

Please see the full version for more examples of simple upper and lower bounds.

## 5 Recoloring algorithm for trees

In this section, we provide two efficient algorithms for recoloring and list-recoloring trees. Note that Theorem 5 is tight; see the full version for more details.

► **Theorem 4.** *For any  $k \in \mathbb{N}$ , for every tree  $T$  on  $n$  nodes, for any two  $k$ -colorings  $\alpha, \beta$  of  $T$ , we can compute in  $O(\log n)$  rounds how to recolor  $T$  from  $\alpha$  to  $\beta$  with 1 extra color and a schedule of length  $O(1)$ .*

► **Theorem 5.** *For every tree  $T$  on  $n$  nodes and any list assignment  $L$  of at least 4 colors to every node of  $T$ , for any two  $L$ -colorings  $\alpha, \beta$  of  $T$ , we can compute in  $O(\log n)$  rounds how to  $L$ -recolor  $T$  from  $\alpha$  to  $\beta$  with schedule of length  $O(\log n)$ .*

We first discuss how to compute efficiently an independent set with some desirable properties. For this, we use a simple modification of the *rake and compress* method by Reif and Miller [20]. More precisely, we iterate rake and compress operations, and label nodes based on the step at which they are reached. We then use the labels to compute an independent set satisfying given properties. We finally explain how to make use of the special independent set to obtain an efficient recoloring algorithm, in each case.

► **Definition 6.** A *light  $h$ -labeling* is a labeling  $V \rightarrow [h]$  such that for any  $i \in [h]$ :

1. Any node labeled  $i$  has at most two neighbors with label  $\geq i$ , at most one of which with label  $\geq i + 1$ .
2. No two adjacent nodes labeled  $i$  both have a neighbor with label  $\geq i + 1$ .

► **Lemma 7.** *There is an  $O(\log n)$ -round algorithm that finds a light  $(2 \log n)$ -labeling of a tree.*

**Proof.** As discussed above, we merely use a small variant of the *rake and compress* method. At step  $i$ , we remove all nodes of degree 1 and all nodes of degree 2 that belong to a chain of at least three nodes of degree 2, and assign them label  $i$ .

One can check that this yields a light labeling. It remains to discuss how many different labels are used, i.e. how many steps it takes to delete the whole tree. Let us argue that no node remains after  $2 \log n$  rounds. Let  $T$  be a tree, let  $V_1$  (resp.  $V_2, V_3$ ) be the number of nodes of degree 1 (resp. 2,  $\geq 3$ ) in the tree, and let  $T'$  be the tree obtained from  $T$  by replacing any maximal path of nodes of degree 2 with an edge. Note that  $|V(T')| = |V_1| + |V_3|$ . Let  $W$  be the set of nodes in  $T$  that have degree 2 with both neighbors of degree 2. Note that  $|V_2 \setminus W| \leq 2|E(T')| = 2(|V_1| + |V_3| - 1)$ . Note also that  $|V_1| \geq |V_3|$ , simply by the fact that there are fewer edges than nodes in a tree. It follows that  $|W| \geq |V_2| - 2(|V_1| + |V_3| - 1) = |V(T)| - |V_1| - |V_3| - 2(|V_1| + |V_3| - 1) \geq |V(T)| - 6|V_1|$ . Consequently, we obtain  $|W| + |V_1| \geq \frac{|V|}{6}$ . In other words, at every step, we remove in particular  $W \cup V_1$ , hence at least a sixth of the nodes. It follows that after  $k$  steps, the number of remaining nodes is at most  $n \cdot (\frac{5}{6})^k$ . Note that this is less than 1 once  $k \geq 2 \log n$ . ◀

We now discuss how to make use of light  $h$ -labelings.

► **Lemma 8.** *For any graph  $T$ , any 3-coloring  $\alpha$  of  $T$ , and any integer  $h$ , let  $L$  be a light  $h$ -labeling of  $T$ . There is an  $O(h)$ -round algorithm that finds a maximal independent set  $S$  such that  $T \setminus S$  only has connected components on 1 or 2 nodes.*

**Proof.** In brief, we proceed as follows: at step  $i = h, h - 1, \dots, 1$ , we first add all nodes of label  $i$  which have a neighbor of label  $\geq i + 1$  that is not in  $S$  (they form an independent set by definition of a light label), then use the 3-coloring to obtain a fast greedy algorithm to make  $S$  maximal on the nodes of label  $\geq i$ . The detailed algorithm can be found in the full version.

The fact that the output  $S$  is an independent set follows directly from the construction, as does the fact that the running time is  $O(h)$  rounds. We note that no connected component of  $T \setminus S$  contains nodes of different labels, due to the first operation at step  $i$ .

It remains to argue that for any  $i$ , the nodes of label  $i$  that do not belong to  $S$  only form connected components of size 1 or 2. Assume for a contradiction that there is a node  $u$  of label  $i$  which has two neighbors  $v$  and  $w$ , also of label  $i$ , such that none of  $\{u, v, w\}$  belongs to  $S$ . By definition of a light label, the node  $u$  has no other neighbor of label  $\geq i$ , a contradiction to the fact that we build  $S$  to be an MIS among the nodes of label  $\geq i$ . ◀

Combining Lemmas 7 and 8, and observing that a 3-coloring of a tree can be obtained in  $O(\log n)$  rounds, we immediately obtain the following.

► **Lemma 9.** *There is an  $O(\log n)$ -round algorithm that finds an MIS in a tree, such that every component induced by non-MIS nodes is of size one or two.*

We are now ready to prove Theorem 4.

**Proof of Theorem 4.** First, we use Lemma 9 to obtain in  $O(\log n)$  rounds an MIS  $S$  such that  $T \setminus S$  only has connected components of size 1 or 2. We recolor each node in  $S$  with the extra color. Remove  $S$ , and recolor each component from  $\alpha$  to  $\beta$  without using any extra colors; this can be done in  $O(1)$  recoloring rounds. Each node in  $S$  can then go directly to its color in  $\beta$ . ◀

Moving on to the list setting, we have to use a more convoluted approach since there is no global extra color that we can use. Before discussing 4-list-recoloring, we discuss 3-list-recoloring. For the sake of intuition, we start by presenting an algorithm for 3-recoloring trees, and explain afterwards how to adapt it for the list setting.

► **Lemma 10.** *For every tree  $T$  with radius at most  $p$  and for any two 3-colorings  $\alpha, \beta$  of  $T$ , we can compute in  $O(p)$  rounds how to 3-recolor  $T$  from  $\alpha$  to  $\beta$  with a schedule of length  $O(p)$ .*

**Proof.** Let  $c: V \rightarrow [3]$  be a 3-coloring of  $T$ . We introduce an identification operation: Given a leaf  $u$  and a node  $v$  such that  $u$  and  $v$  have a common neighbor  $w$ , we recolor  $u$  with  $c(v)$ , and from then on we pretend that  $u$  and  $v$  are a single node. In other words, we delete  $u$  from the tree we are considering, and reflect any recoloring of  $v$  to the node  $u$ . Note that these operations can stack up: the recoloring of a single node might be reflected on an arbitrarily large independent set in the initial tree.

We now briefly describe an algorithm to recolor a 3-coloring into a 2-coloring  $c'$  in  $O(p)$  rounds, with schedule  $O(p)$ . First, root  $T$  on a node  $r$  which is at distance at most  $p$  of any node of  $T$ . Any node of  $T$  which is not adjacent to the root has a *grandparent*, which is defined as its parent's parent.

Then, at each step, we consider the set  $A$  of leaves of  $T$  which have a grandparent, if any. We identify each leaf in  $A$  with its grandparent (note that the notion of grandparent guarantees that this operation is well-defined, and that the operation results in  $A$  being deleted).



## 12:10 Distributed Recoloring

This process stops when  $T$  consists only of the root  $r$  and its children. We select one of the children arbitrarily and identify the others with it. This results in  $T$  being a single edge. Note that the color partition of  $c'$  is compatible with the identification operations, as we only ever identify nodes at even distance of each other.

We then recolor  $T$  into  $c'$ : this is straightforward in the realm of 3-recoloring.

We can now choose a 2-coloring of  $T$  (this can be done in  $O(p)$  rounds), and apply the above algorithm to 3-recolor both  $\alpha$  and  $\beta$  to that 2-coloring. This results in a 3-recoloring between  $\alpha$  and  $\beta$  with schedule  $O(p)$ . ◀

The same idea can be adapted to list coloring; we give a proof of the following result in the full version of the article:

► **Lemma 11.** *For every tree  $T$  with radius at most  $p$ , for any list assignment  $L$  of at least 3 colors to each node, for any two  $L$ -colorings  $\alpha, \beta$  of  $T$ , we can compute in  $O(p)$  rounds how to  $L$ -recolor  $T$  from  $\alpha$  to  $\beta$  with schedule  $O(p)$ .*

To prove Theorem 5, we first split the tree in small components. We slightly adapt the proof of Lemma 8; see the full version for the details:

► **Lemma 12.** *For any tree  $T$ , any 3-coloring  $\alpha$  of  $T$ , and any integer  $h$ , let  $L$  be a light  $h$ -label of  $T$ . There is a  $O(h)$ -round algorithm that finds a maximal independent set  $S$  such that no node has two neighbors in  $S$  and  $T \setminus S$  only has connected components of radius  $O(h)$ .*

Now we are ready to prove Theorem 5.

**Proof of Theorem 5.** Compute (in  $O(\log n)$  rounds) an independent set  $S$  such any two elements of  $S$  are at distance at least 2 of each other and every connected component of  $T \setminus S$  has radius  $O(\log n)$ . By Lemmas 7 and 12 and the fact that a 3-coloring of a tree can be computed in  $O(\log n)$  rounds, we compute (in  $O(\log n)$  rounds) an  $L$ -coloring  $\gamma$  of  $T \setminus S$  such that every node adjacent to an element  $u \in S$  has a color different from  $\alpha(u)$  and  $\beta(u)$ . Note that this coloring exists since any tree is 2-list-colorable. Use Lemma 11 to recolor each connected component of  $T \setminus S$  from  $\alpha$  to  $\gamma$ . Recolor every element of  $S$  with its color in  $\beta$ . Use Lemma 11 to recolor each connected component  $T \setminus S$  from  $\gamma$  to  $\beta$ . Note that this yields an  $L$ -recoloring of  $T$  from  $\alpha$  to  $\beta$  with schedule  $O(\log n)$ . ◀

Note that a direct corollary of Theorem 5 is that for any  $k$ -coloring  $\alpha, \beta$  of a trees with  $k \geq 4$ , a schedule of length  $\Theta(\log n)$  can be found in  $\Theta(\log n)$  rounds.

## 6 Recoloring algorithm for subcubic graphs

In this section we study recoloring in subcubic graphs (graphs of maximum degree at most 3); our main result is summarized in the following theorem:

► **Theorem 13.** *For every subcubic graph  $G$  on  $n$  nodes, for any two 3-colorings  $\alpha, \beta$  of  $G$ , we can compute in  $O(\log^2 n)$  rounds how to recolor  $G$  from  $\alpha$  to  $\beta$  with 1 extra color and a schedule of length  $O(\log n)$ .*

A *theta* is formed of three node-disjoint paths between two nodes. Note that in particular if a graph contains two cycles sharing at least one edge, then it contains a theta. We note  $B^k(u)$  the set of nodes at distance at most  $k$  to  $u$ .

We show here, roughly, that there is around every node a nice structure that we can use to design a valid greedy algorithm for the whole graph. This proof is loosely inspired by one in [1]. The proofs of Lemmas 14 and 15 are given in the full version.



**Algorithm 1** DECOMPOSING INTO A SMALL FOREST AND AN INDEPENDENT SET.**Require:** A subcubic graph  $G$ .**Ensure:** A decomposition  $(F, S)$  of  $V(G)$  such that  $G[S]$  is an independent set and every connected component of  $G[F]$  has radius at most  $\log n$ .

- 1: **for**  $u$  in  $V(G)$  (in parallel) **do**
- 2:   Acquire knowledge on  $B^{2\log n}(u)$
- 3:   Select in the node set of  $B^{2\log n}(u)$  a configuration  $C(u)$  that is a minimal theta or a node of degree 1 or 2
- 4: **end for**
- 5: Compute a  $(4\log n, 8\log n)$ -ruling set  $X$  in  $G$
- 6: Define  $\mathcal{A} = \cup_{u \in X} \{C(u)\}$
- 7: Compute the distance of every node in  $G$  to an element of  $\mathcal{A}$
- 8: Let  $F = S = \emptyset$
- 9: **for**  $i = 8\log n$  downto 1 **do**
- 10:   Extend the partition  $(F, S)$  to the nodes at distance  $i$  from  $\mathcal{A}$ , more precisely:
- 11:   Each connected component is a path or cycle where no internal node has an already assigned neighbor, let  $U_i$  be the set of the internal nodes
- 12:   Assuming a pre-computed MIS on each layer for the sets  $U_i$ , assign that MIS to  $S$
- 13:   Extend greedily on the remaining nodes (which form bounded-size components), assigning nodes to  $S$  when possible, to  $F$  when not
- 14: **end for**
- 15: Extend the partition  $(F, S)$  to the nodes belonging to an element of  $\mathcal{A}$  using Lemma 15

► **Lemma 14.** *For every subcubic graph  $G$  on  $n$  nodes, for every node  $u \in V(G)$ , there is a node  $v$  with degree at most 2 or a theta that is contained in  $B^{2\log n}(u)$ .*

► **Lemma 15.** *Let  $G$  be a subcubic graph, let  $p$  be an integer, and let  $\mathcal{A}$  be a collection of thetas and nodes of degree  $\leq 2$  in  $G$  each at distance at least 2 of each other. Let  $r \geq 1$  be such that no element of  $\mathcal{A}$  has diameter more than  $\frac{r}{2}$ . If the nodes of  $G \setminus (\cup_{A \in \mathcal{A}} A)$  can be partitioned into  $S$  and  $F$  such that  $G[S]$  is an independent set and  $G[F]$  is a forest of radius at most  $p$ , then there is a partition  $(S', F')$  of  $\cup_{A \in \mathcal{A}} A$  such that  $G[S \cup S']$  is an independent set and  $G[F \cup F']$  is a forest of radius at most  $p + r$ .*

► **Lemma 16.** *Let  $G$  be a subcubic graph on  $n$  nodes. We can compute in  $O(\log^2 n)$  rounds a partition  $(S, F)$  of the nodes of  $G$  that  $G[S]$  is an independent set and  $G[F]$  is a forest of radius  $O(\log n)$ .*

**Proof.** To that purpose, we combine the previous lemmas in Algorithm 1. The algorithm computes a decomposition as desired and runs in  $O(\log n) + RS(n)$  rounds, where  $RS(n)$  is the number of rounds necessary to compute a  $(4\log n, 8\log n)$ -ruling set in a subcubic graph. We derive from [22] that  $RS(n) = O(\log^2(n))$ , hence the conclusion. ◀

We are now ready to prove Theorem 13, which we do in a similar fashion as Theorem 4.

**Proof.** Use Lemma 16, and obtain a decomposition  $(S, F)$  as stated. Recolor all of  $S$  to the extra color, then use Lemma 11 on each connected component of  $G[F]$  so that all nodes of  $F$  reach their target color (remember that each connected component of  $G[F]$  has radius  $O(\log n)$ ). Finally recolor each node of  $S$  with its target color. ◀

## 7 Recoloring in toroidal grids

In this section we study toroidal grids (torus grid graphs). Throughout this section, an  $h \times w$  toroidal grid is the Cartesian graph product of cycles of lengths  $h$  and  $w$ ; we assume  $h \geq 3$  and  $w \geq 3$ . A toroidal grid can be constructed from an  $h \times w$  grid by wrapping both boundaries around into a torus. In the full version, we show that e.g.  $2 + 0$ ,  $3 + 0$ , and  $4 + 0$  recoloring is not always possible, and by Lemma 2 e.g.  $2 + 1$ ,  $3 + 2$ , and  $4 + 3$  recoloring is trivial. The first nontrivial case is  $3 + 1$  recoloring; in this section we give a complete characterization of  $3 + 1$  recolorability in toroidal grids:

► **Theorem 17.** *Let  $G$  be the  $h \times w$  toroidal grid graph. Then  $3 + 1$  recoloring is possible for any source and target coloring in the following cases: (i) both  $h$  and  $w$  are even, or (ii)  $h = 4$ , or (iii)  $w = 4$ . For all other cases it is possible to construct 3-colorings  $s$  and  $t$  such that  $t$  is not reachable from  $s$  by valid recoloring operations using 1 extra color.*

This also shows that  $3 + 1$  recoloring is an inherently global problem in toroidal grids, even if we have a promise that recoloring is possible. For example, if there was a sublinear-time distributed recoloring algorithm  $A$  for  $6 \times w$  grids for an even  $w$ , we could apply the same algorithm in a  $6 \times w$  grid with an odd  $w$  (the algorithm cannot tell the difference between these two cases in time  $o(w)$ ), and hence we could solve recoloring in  $6 \times w$  grids for all  $w$ , which contradicts Theorem 17. By a similar argument, distributed recoloring in non-toroidal grids is also an inherently global problem.

**Existence.** To prove Theorem 17, let us start with the positive results. If  $h$  and  $w$  are even, the graph is bipartite and recoloring is always possible by Lemma 1. The remaining cases are covered by the following lemma.

► **Lemma 18.** *Let  $G$  be a  $4 \times w$  toroidal grid for any  $w \geq 3$ , and let  $s$  and  $t$  be any 3-colorings. Then there exists a recoloring from  $s$  to  $t$  with one extra color.*

**Proof.** We first take an MIS  $S$  over pairs of consecutive columns, i.e. a set of indices of the form  $(i, i + 1)$  such that every column  $j \notin S$  is such that at least one of  $j - 1$  and  $j + 2$  belongs to  $S$ , every column  $i \in S$  is such that precisely one of  $i - 1$  and  $i + 1$  is in  $S$ . Note that indices are taken modulo  $w$ . For every pair in  $S$ , we select a maximal independent set of the corresponding columns. The resulting union yields an independent set  $R$ . We then greedily make  $R$  maximal columnwise away from  $S$ . We recolor  $R$  with the extra color. It remains to argue that  $G \setminus R$  can reach its targeted coloring. We note that since leaves are not problematic, removing  $R$  essentially boils down to removing the columns with index in  $S$ . Note that the remaining connected components are cycles of length 4. Cycles of length 4 can be always 3-recolored.

Note that the above proof yields in fact an  $O(\log n)$  rounds algorithm that outputs an  $O(1)$  schedule. We can improve it into an  $O(1)$ -round algorithm, simply by pointing out that there is only a finite number of possible colorings for a column, and two adjacent columns cannot have the same coloring. This allows us to compute  $S$  in constant time. ◀

**Non-existence.** Let us now prove the negative result. Our high-level plan is as follows. Let  $G$  be an  $h \times w$  toroidal grid. We will look at all *tiles* of size  $2 \times 2$ . If  $G$  is properly colored with  $k$  colors, so is each tile. The following two tiles are of special importance to us; we call these tiles of *type A*:

$$\begin{bmatrix} 2 & 3 \\ 3 & 1 \end{bmatrix}, \quad \begin{bmatrix} 1 & 3 \\ 3 & 2 \end{bmatrix}.$$

We are interested in the *number* of type-A tiles. For example, consider the following colorings of the  $3 \times 3$  toroidal grid:

$$s = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \\ 3 & 1 & 2 \end{bmatrix}, \quad t = \begin{bmatrix} 3 & 1 & 2 \\ 2 & 3 & 1 \\ 1 & 2 & 3 \end{bmatrix}.$$

Here  $s$  contains 3 tiles of type A (recall that we wrap around at the boundaries), while  $t$  does not have any tiles of type A. In particular,  $s$  has an odd number of type-A tiles and  $t$  has an even number of type-A tiles. In brief, we say that the *A-parity* of  $s$  is odd and the A-parity of  $t$  is even. It turns out that this is sufficient to show that recoloring from  $s$  to  $t$  with one extra color is not possible (see the full version of the article for the proof of this lemma):

► **Lemma 19.** *Let  $G$  be a toroidal grid, and let  $s$  and  $t$  be two 3-colorings. If  $s$  and  $t$  have different A-parities, then it is not possible to recolor  $G$  from  $s$  to  $t$  with 1 extra color.*

Hence the A-parity of a coloring partitions the space of colorings in two components that are not connected by  $3 + 1$  recoloring operations. To complete the proof of Theorem 17, it now suffices to construct a pair of 3-colorings with different A-parities for each relevant combination of  $h$  and  $w$ . We give the details in the full version.

## 8 Simple corollaries

► **Lemma 20.** *Assume that we are given a graph  $G$  and input and target colorings with  $k \geq 3$  colors. Assume that in  $O(f(n))$  rounds we can find an independent set  $I$  of  $G$  such that  $V \setminus I$  induces a forest of trees of depth at most  $O(d(n))$ . Then in  $O(f(n) + d(n))$  rounds we can solve  $k + 1$  recoloring, with a schedule of length  $O(d(n))$ .*

**Proof.** Each node in  $I$  switches to color  $k + 1$ . We then use the algorithm described in the proof of Lemma 10 to find a recoloring with schedule of length  $O(d(n))$  for each connected component after the removal of  $I$ . After that, each node of  $I$  can switch to its final color. ◀

► **Lemma 21.** *In cycles and paths,  $3 + 1$  recoloring is possible in  $O(1)$  rounds, with a schedule of length  $O(1)$ .*

**Proof.** Use the input coloring to find a maximal independent set  $I$ . Nodes of  $V \setminus I$  induce paths of length  $O(1)$ , apply Lemma 20. ◀

► **Lemma 22.** *In subcubic graphs,  $4 + 1$  recoloring is possible in  $O(1)$  rounds, with a schedule of length  $O(1)$ .*

**Proof.** Use the input coloring to find a maximal independent set  $I$  in constant time. Nodes of  $I$  switch to color 5. Delete  $I$ ; we are left with a graph  $G'$  that consists of paths and isolated nodes. Apply Lemmas 3 and 21 to solve  $4 + 0$  recoloring in each connected component of  $G'$ . Finally nodes of  $I$  can switch to their target colors. ◀

## 12:14 Distributed Recoloring

■ **Table 1** Results: distributed recoloring in cycles (C) and paths (P).

graph family	input colors	extra colors	schedule length	communication rounds	reference
C/P	2	0	$\infty$		see full version
C/P	2	1	$O(1)$	0	Lemma 2
C	3	0	$\infty$		see full version
P	3	0	$\Theta(n)$	$\Theta(n)$	see full version
C/P	3	1	$O(1)$	$O(1)$	Lemma 21
C/P	3	2	$O(1)$	0	Lemma 2
C/P	4	0	$O(1)$	$O(1)$	Lemmas 21 and 3
C/P	4	3	$O(1)$	0	Lemma 2

■ **Table 2** Results: distributed recoloring in 3-regular trees.

input colors	extra colors	schedule length	communication rounds	reference
2	0	$\infty$		see full version
2	1	$O(1)$	0	Lemma 2
3	0	$\Theta(n)$	$\Theta(n)$	see full version
3	1	$O(1)$	$O(\log n)$	Theorem 4
3	2	$O(1)$	0	Lemma 2
4	0	$\Theta(\log n)$	$\Theta(\log n)$	Theorem 5 and see full version
4	1	$O(1)$	$O(1)$	Lemma 22
4	3	$O(1)$	0	Lemma 2
5	0	$O(1)$	$O(1)$	Lemmas 22 and 3

■ **Table 3** Results: distributed recoloring in trees.

input colors	extra colors	schedule length	communication rounds	reference
2	0	$\infty$		see full version
2	1	$O(1)$	0	Lemma 2
3	0	$\Theta(n)$	$\Theta(n)$	see full version
3	1	$O(1)$	$O(\log n)$	Theorem 4
3	2	$O(1)$	0	Lemma 2
4	0	$\Theta(\log n)$	$\Theta(\log n)$	Theorem 5 and see full version
4	1	$O(1)$	$O(\log n)$	Theorem 4
4	3	$O(1)$	0	Lemma 2

■ **Table 4** Results: distributed recoloring in toroidal grids. The distributed complexity of  $4 + 1$  recoloring is left as an open question. However, by prior work it is known that  $4 + 1$  recoloring is always possible: grids are 4-regular graphs, therefore they are 4-recolorable with Kempe operations, and hence also with 1 extra color.

input colors	extra colors	schedule length	communication rounds	reference
2	0	$\infty$		see full version
2	1	$O(1)$	0	Lemma 2
3	0	$\infty$		see full version
3	1	$\infty$		Theorem 17
3	2	$O(1)$	0	Lemma 2
4	0	$\infty$		see full version
4	1	?	?	
4	2	$O(1)$	$O(1)$	see full version
4	3	$O(1)$	0	Lemma 2
5	0	$\infty$		see full version
5	1	$O(1)$	$O(1)$	see full version
5	4	$O(1)$	0	Lemma 2
6	0	$O(1)$	$O(1)$	Lemma 3 and see full version

■ **Table 5** Results: distributed recoloring in subcubic graphs.

input colors	extra colors	schedule length	communication rounds	reference
2	0	$\infty$		see full version
2	1	$O(1)$	0	Lemma 2
3	0	$\infty$		see full version
3	1	$O(\log n)$	$O(\log^2 n)$	Theorem 13
3	2	$O(1)$	0	Lemma 2
4	0	$\infty$		see full version
4	1	$O(1)$	$O(1)$	Lemma 22
4	3	$O(1)$	0	Lemma 2
5	0	$O(1)$	$O(1)$	Lemma 3

## References

- 1 Pierre Aboulker, Marthe Bonamy, Nicolas Bousquet, and Louis Esperet. Distributed coloring in sparse graphs with fewer colours. *arXiv preprint arXiv:1802.05582*, 2018.
- 2 Leonid Barenboim. Deterministic  $(\Delta + 1)$ -coloring in sublinear (in  $\Delta$ ) time in static, dynamic and faulty networks. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 345–354, 2015.
- 3 Leonid Barenboim and Michael Elkin. Distributed graph coloring: Fundamentals and recent developments. *Synthesis Lectures on Distributed Computing Theory*, 4(1):1–171, 2013.
- 4 Marthe Bonamy and Nicolas Bousquet. Recoloring graphs via tree decompositions. *European Journal of Combinatorics*, 69:200–213, 2018.
- 5 Marthe Bonamy, Nicolas Bousquet, Carl Feghali, and Matthew Johnson. On a conjecture of Mohar concerning Kempe equivalence of regular graphs. *arXiv preprint arXiv:1510.06964*, 2015.
- 6 Paul Bonsma and Luis Cereceda. Finding paths between graph colourings: PSPACE-completeness and superpolynomial distances. *Theoretical Computer Science*, 410(50):5215–5226, 2009.
- 7 Paul Bonsma, Amer E Mouawad, Naomi Nishimura, and Venkatesh Raman. The complexity of bounded length graph recoloring and CSP reconfiguration. In *International Symposium on Parameterized and Exact Computation*, pages 110–121. Springer, 2014.
- 8 Nicolas Bousquet and Guillem Perarnau. Fast recoloring of sparse graphs. *European Journal of Combinatorics*, 52:1–11, 2016.
- 9 Sebastian Brandt, Juho Hirvonen, Janne H Korhonen, Tuomo Lempiäinen, Patric RJ Östergård, Christopher Purcell, Joel Rybicki, Jukka Suomela, and Przemysław Uznański. LCL problems on grids. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 101–110. ACM, 2017.
- 10 Luis Cereceda, Jan Van den Heuvel, and Matthew Johnson. Mixing 3-colourings in bipartite graphs. *European Journal of Combinatorics*, 30(7):1593–1606, 2009.
- 11 Luis Cereceda, Jan Van Den Heuvel, and Matthew Johnson. Finding paths between 3-colorings. *Journal of graph theory*, 67(1):69–82, 2011.
- 12 Yi-Jung Chang, Tsvi Kopelowitz, and Seth Pettie. An exponential separation between randomized and deterministic complexity in the LOCAL model. In *Foundations of Computer Science (FOCS)*, pages 615–624, 2016.
- 13 Yi-Jung Chang, Wenzheng Li, and Seth Pettie. An optimal distributed  $(\Delta + 1)$ -coloring algorithm? In *Proceedings of the 50th ACM Symposium on Theory of Computing (STOC)*, 2018.
- 14 Carl Feghali, Matthew Johnson, and Daniël Paulusma. A reconfigurations analogue of Brooks’ theorem and its consequences. *Journal of Graph Theory*, 83(4):340–358, 2016.
- 15 Carl Feghali, Matthew Johnson, and Daniël Paulusma. Kempe equivalence of colourings of cubic graphs. *European Journal of Combinatorics*, 59:1–10, 2017.
- 16 Mark Jerrum. A very simple algorithm for estimating the number of  $k$ -colorings of a low-degree graph. *Random Structures & Algorithms*, 7(2):157–165, 1995.
- 17 Alfred B Kempe. On the geographical problem of the four colours. *American Journal of Mathematics*, 2(3):193–200, 1879.
- 18 Michel Las Vergnas and Henri Meyniel. Kempe classes and the Hadwiger conjecture. *Journal of Combinatorial Theory, Series B*, 31(1):95–104, 1981.
- 19 Daniel C McDonald. Connectedness and Hamiltonicity of graphs on vertex colorings. *arXiv preprint arXiv:1507.05344*, 2015.
- 20 Gary L. Miller and John H. Reif. Parallel tree contraction part 1: Fundamentals. *Advances in Computing Research*, 5:47–72, 1989.

- 21 Moni Naor and Larry Stockmeyer. What can be computed locally? *SIAM Journal on Computing*, 24(6):1259–1277, 1995.
- 22 Alessandro Panconesi and Aravind Srinivasan. Improved distributed algorithms for coloring and network decomposition problems. In *Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, pages 581–592. ACM, 1992.
- 23 Alessandro Panconesi and Aravind Srinivasan. The local nature of  $\Delta$ -coloring and its algorithmic applications. *Combinatorica*, 15(2):255–280, 1995.
- 24 Jan van den Heuvel. The complexity of change. *Surveys in Combinatorics*, 409(2013):127–160, 2013.





# A Tight Lower Bound for Semi-Synchronous Collaborative Grid Exploration

**Sebastian Brandt**

ETH Zürich, Switzerland

brants@ethz.ch

**Jara Uitto**<sup>1</sup>

ETH Zürich, Switzerland

juitto@ethz.ch

**Roger Wattenhofer**

ETH Zürich, Switzerland

wattenhofer@ethz.ch

---

## Abstract

Recently, there has been a growing interest in grid exploration by agents with limited capabilities. We show that the grid cannot be explored by three semi-synchronous finite automata, answering an open question by Emek et al. [TCS'15] in the negative.

In the setting we consider, time is divided into discrete steps, where in each step, an adversarially selected subset of the agents executes one look-compute-move cycle. The agents operate according to a shared finite automaton, where every agent is allowed to have a distinct initial state. The only means of communication is to sense the states of the agents sharing the same grid cell. The agents are equipped with a global compass and whenever an agent moves, the destination cell of the movement is chosen by the agent's automaton from the set of neighboring grid cells. In contrast to the four agent protocol by Emek et al., we show that three agents do not suffice for grid exploration.

**2012 ACM Subject Classification** Computing methodologies → Mobile agents

**Keywords and phrases** Finite automata, Graph exploration, Mobile robots

**Digital Object Identifier** 10.4230/LIPIcs.DISC.2018.13

**Related Version** A full version of the paper is available at [8], <https://arxiv.org/abs/1705.03834>.

## 1 Introduction

Consider the problem of exploring an infinite grid with a set of mobile robots, ants, or *agents*. In practical applications, it is often desirable to make use of inexpensive and simple devices and therefore, a finite automaton is an attractive choice for modeling these agents. Furthermore, neither reliable communication nor synchronous time is always available and thus, distributed and non-synchronous solutions are needed. Also exploration models inspired by biology require these features; for example models for ant foraging assume limited capabilities and distributed searching. In both settings mentioned above, it is often reasonable to assume simple means of communication of nearby agents.

---

<sup>1</sup> Partially supported by ERC Grant No. 336495 (ACDC).



### Semi-Synchrony

Recently, there has been a growing interest in studying constant memory agents performing exploration on an infinite grid. An infinite grid is a natural discrete version of a plane which disallows the bounded memory agents to make any use of the boundaries of the grid. Emek et al. [17] introduced a model where the agents are able to communicate by sensing each other's states and showed a tight upper bound for the time needed for  $k$  agents to find a treasure<sup>2</sup> at distance  $D$ . As the first step into the model, let us introduce the way that the semi-synchrony is defined. The time is divided into discrete time steps, and in each time step, an adversarially chosen subset of the agents performs a look-compute-move cycle in parallel. In each cycle, the chosen agents first sense the states of all the other agents in the same cell and then, determined by their transition function, either stay still or move to an adjacent grid cell. We point out that in every step, every agent performs the "look" action before any agent executes their "compute" step, i.e., agents sharing a cell and activated in the same time step see each other's states before any of them executes a state transition. This definition allows an arbitrary discrepancy in the number of steps the agents perform but ensures that, whenever two agents meet, at least one of them will be able to sense the presence of the other agent.

All input parameters, such as  $D$  and  $k$  are unknown to the agents and they are all initially located in the origin of the grid. Motivated by the fact that ants are able to perform very precise path integration, it is assumed that the agents are endowed with a global compass.

### Previous Results

Following up on the above model, Emek et al. [16] studied the minimum number of agents needed to explore the infinite grid, where exploring refers to reaching any fixed cell within (expected) finite time. They showed that three randomized and four deterministic semi-synchronous agents are enough for the exploration task. We want to point out that the *asynchronous* environment in their paper is referred to as semi-synchronous in older literature [25, 26]. The paper left two open questions:

*Can two agents controlled by a randomized FA solve the synchronous or asynchronous version of the ANTS<sup>3</sup> problem?*

*Is there an effective FA-protocol for async-ANTS for three agents when no random bits are available?*

Very recently, Cohen et al. solved the first question by showing that two randomized agents do not suffice [11]. The main result of this paper is a negative answer to the second question:

► **Theorem 1.** *Three semi-synchronous agents controlled by a finite automaton are not sufficient to explore the infinite grid.*

Our result is obtained by solving two technical challenges. First, we carefully design an adversarial schedule for the agents that, under the assumption that the agents actually explore the entire grid, forces them to obey a movement pattern with the following property: There is a fixed width  $w$  and fixed slope  $s$  such that at any point in time, all agents are

---

<sup>2</sup> In the deterministic case, exploring the grid and finding a treasure are equivalent. In the randomized case, considering a treasure is more convenient as the exploration is equivalent to hitting every cell in expected finite time.

<sup>3</sup> The ANTS problem in their context is the same as our grid exploration problem.

contained in a band of width  $w$  and slope  $s$ . Second, we formally show that the agents cannot encode a super-constant amount of information in their relative positions. In other words, while the relative distance can be unbounded and represent an unbounded amount of information, we can bound the amount of information the agents can infer from their relative positions. Due to space constraints, most of our proofs are deferred to the full version of the paper [8].

## 2 Related Work

Graph exploration is a widely studied problem in the computer science literature. In the typical setting one or more agents are placed on some node of a graph and the goal is to visit every node and/or edge of the graph by moving along the edges. There is a wide selection of variants of graph exploration and one of the standard ways to classify these variants is to divide them into *directed* and *undirected* variants [12, 1]. In the directed model, the edges of the graph only allow traversing into one direction, whereas in the undirected model, traversing both ways is allowed. Our work assumes the undirected graph exploration model.

Other typical parameters of the problem are the conditions of a successful exploration and symmetry breaking mechanisms. Some related works demand that the agents are required to halt after a successful exploration [13] or that the agents must return to their starting point after the exploration [3]. From the perspective of symmetry breaking, one characterization is to break the problem into the case of equipping nodes with unique identifiers [23, 15] and into the case where nodes are anonymous [9, 24, 5]. Since the memory of our agents is restricted to a constant amount of bits with respect to the size of the graph, the unique identifiers are not helpful.

The agents typically operate in *look-compute-move* cycles, where they first gather the local information, then perform local computations, and finally, decide to which node they move. This execution model can be divided into *synchronous* [26], *semi-synchronous* [25, 26] and *asynchronous* variants [27, 19], referred to as  $\mathcal{FSYNC}$ ,  $\mathcal{SSYNC}$ , and  $\mathcal{ASYNC}$ . In the  $\mathcal{FSYNC}$  model, all agents execute their cycles simultaneously in discrete rounds. In the  $\mathcal{SSYNC}$  model only a subset (not necessarily proper) of the agents is *activated* in every round and in the  $\mathcal{ASYNC}$  model, the cycles are not assumed to be atomic. To avoid confusion, we refer to the non-synchronous rounds as time steps. In this paper, we consider the semi-synchronous model. Note that since the  $\mathcal{ASYNC}$  model is weaker than the  $\mathcal{SSYNC}$  model, we directly obtain our lower bound result for the  $\mathcal{ASYNC}$  model as well.

The standard efficiency measure of a graph exploration algorithm executed in the  $\mathcal{FSYNC}$  model is the number of synchronous rounds it takes until the graph is explored [23]. In the non-synchronous models, this measure is typically generalized to the maximum delay between activation times of any agent [10]. A widely-studied classic is the *cow-path* problem, where the goal of the cow is to find food or a treasure on a line as fast as possible. There is an algorithm with a constant competitive ratio for the case of a line and in the case of a grid, a simple spiral search is optimal and the problem has been generalized to the case of many cows [4, 22]. Some more recent work studied the time complexity of  $n$  distributed agents searching for a treasure in distance  $D$  on a grid and a  $\Theta(D/n^2 + D)$  bound was shown in the case of Turing machines without communication and in the case of communicating finite automata [18, 17].

Our work does not focus on the time complexity of the problem, but rather on the computability, i.e., what is the minimum number of agents that are required to find the treasure. The canonical algorithm in the case of little memory is the random walk, where the

classic result states that a random walk explores an  $n$ -node graph in polynomial time [2]. In the case of infinite grids, it was shown in a recent paper that, even with a globally consistent orientation, two randomized agents cannot locate the treasure in finite expected time [11]. By combining this result with previous work [6, 16], it follows that this lower bound is tight. In the deterministic case, our lower bound of three deterministic semi-synchronous agents closes the remaining gap in the results of [16].

Another typical measure for efficiency is the number of bits of memory needed per agent [20, 13]. For example, it was shown by Fraigniaud et al., that  $\Theta(D \log \Delta)$  bits are needed for a single agent to locate the treasure, where  $D$  and  $\Delta$  denote the diameter and the maximum degree of the graph, respectively. The memory of our agents is bounded by a universal constant, independent of any graph parameters.

Work that falls close to our work is the study of graph exploration in *labyrinths*, i.e., graphs that can be seen as 2-dimensional grids, where some subset of the nodes cannot be entered by the agents. The classic results state that all co-finite (finite amount of cells not blocked) labyrinths can be explored by two finite automata and an automaton with two pebbles [7], and that finite labyrinths (finite amount of cells are blocked) can be explored using one agent with four pebbles [6], where a pebble is a movable marker. Furthermore, it is known since long that there are finite and co-finite labyrinths where one pebble is not enough [21] and that no finite set of finite automata can explore all planar graphs [24]. More recently, it was shown that  $\Theta(\log \log n)$  pebbles for an agent with  $\Theta(\log \log n)$  memory is the right answer for general graphs [14]. Notice that since we do not assume synchronous communication between agents and a pebble can always be simulated by a finite automaton, our result also yields the same bound for the pebble model.

### 3 Preliminaries

#### 3.1 The Model

The model we use is the same as in [16]. We consider a group of  $n$  agents whose task is to explore every cell of the infinite 2-dimensional grid where a cell is considered as explored when it has been visited by at least one of the agents. We identify each cell of the grid with a pair of integers, i.e., the grid can be considered as  $\mathbb{Z}^2$ , with two cells being *neighbors* if and only if they differ in one coordinate by exactly 0 and in the other coordinate by exactly 1.

In the beginning, all agents are placed in the same cell, called the *origin*. W.l.o.g., we will assume that the origin has the coordinates  $(0, 0)$ . For the agents, all cells, including the origin, are indistinguishable; in particular, they do not have access to the coordinates of the cells.

Each agent is endowed with a *compass*, i.e., each agent is able to distinguish between the four (globally consistent) cardinal directions in any cell and all agents have the same notion of those directions. The behavior of each agent is governed by a deterministic finite automaton. While we allow the agents to use different finite automata, we will assume that the agents use the same finite automaton but have different initial states. Since in all cases we consider,  $n$  is a constant, the two formulations are equivalent.

The only way in which communication takes place is the following: Each agent senses for any state  $q$  of the finite automaton whether there is at least one other agent in the same cell in state  $q$ . In each step of the execution, an agent moves to an adjacent cell or stays in the current cell, solely based on its current state in the finite automaton and the subset of states  $q$  for which another agent in state  $q$  is present in the current cell.

Given the above, we are set to describe our finite automaton more formally. Let  $Q$  denote the set of states, with each agent having its own initial state in  $Q$ . The set of input symbols is  $2^Q$ , the set of all subsets of  $Q$ , reflecting the fact that for each state from  $Q$  an agent in this state might be present or not in the considered cell. The transition function  $\delta : Q \times 2^Q \rightarrow Q \times \{0, 1, 2, 3, 4\}$  provides an agent in state  $q \in Q$  (sensing a subset  $Q' \subseteq Q$  of states present in the same cell) with a new state  $q' \in Q$  and a movement, where 1, 2, 3, 4 stand for the four cardinal directions while 0 indicates that the agent stays in the current cell.

The *SSYNC* [25, 26] environment in which the agents perform their exploration is *semi-synchronous*. More specifically, we assume that the order of the steps of the agents is determined by an *adversarial scheduler* that knows the finite automaton governing the agents' behavior. Each step of an agent is a complete *look-compute-move* cycle, where first an agent senses for which states agents are present in the current cell, then it applies the transition function with the sensed states and its own current state as input, and finally it moves as indicated by the result. Cycles of different agents may occur at the same time, in which case each of the agents completes the sensing before any of the agents starts to move. Cycles that do not occur at the same time have no overlap, i.e., the movement performed in an earlier cycle is completed before the sensing in a later cycle starts. Hence, we may consider the order of the individual components of the execution as given by a mapping of the agents' steps to points in time.

We call such a mapping a *schedule*. Since the look-compute-move cycles of the agents are atomic in nature, we can assume w.l.o.g. that the static configurations of the agents on the grid (including the information about the states they are currently in) occur at integer points in time  $t = 0, 1, \dots$ , and that the steps of the agents determining the transition from one configuration to a new one take place between these points in time. If an agent's action is scheduled between time  $t$  and  $t + 1$ , we say, for the sake of simplicity, that the action takes place at time  $t$ . In order to prevent the adversary from delaying a single agent indefinitely, we adopt the common requirement that each agent is scheduled infinitely often. For our lower bound we will only use adversarial schedules where no two agents are scheduled at the same time.

### 3.2 Definitions and Notation

For the notion of distance between two cells we will use the Manhattan distance. Let  $c = (x, y)$ ,  $c' = (x', y')$  be two cells of the infinite grid. Then, the *distance* between  $c$  and  $c'$  is defined as  $\text{Dist}(c, c') = |x - x'| + |y - y'|$ . Moreover, we call the first coordinate of a cell the *x-coordinate* and the second coordinate the *y-coordinate*. We denote the cell an agent  $a$  occupies at time  $t$  by  $c_t(a) = (x_t(a), y_t(a))$ . Similarly, we denote the state of the finite automaton in which agent  $a$  is at time  $t$  by  $q_t(a)$ . If  $a = a_i$  for some  $1 \leq i \leq 3$ , then we also write  $c_t^i, x_t^i, y_t^i, q_t^i$  instead of  $c_t(a_i), x_t(a_i), y_t(a_i), q_t(a_i)$ , respectively. Moreover, we denote the number of states of the finite automaton governing the behavior of the three agents by  $N$ .

In our lower bound proof, we show for each finite automaton that three agents governed by this automaton are not sufficient to explore the grid (or, more precisely, that there is an adversarial schedule for this automaton under which the agents do not explore every cell of the grid). In this context, we consider the number  $N$  as a constant, which also implies that the result of applying any fixed polynomial function to  $N$  is a constant as well. For the proof of our lower bound we require another intuitive definition. Let  $\ell$  be an infinite line in the Euclidean plane and  $d$  some positive real number. Let  $B$  be the set of all points in the plane with integer coordinates and Euclidean distance at most  $d$  to  $\ell$ . Let  $B'$  be the set of all grid cells that have the same coordinates as some point in  $B$ . Then we call  $B'$  a *band*.

### 3.3 A Single Agent

Consider a single agent  $a$  moving on the grid. Since the number of states of its finite automaton is finite,  $a$  must repeat a state at some point, i.e., there must be points in time  $t, t'$  such that  $q_t(a) = q_{t'}(a)$  and  $q_{t''}(a) \neq q_t(a)$  for all  $t < t'' < t'$ . As shown in [16], agent  $a$  will then, starting at time  $t'$ , repeat the exact behavior it showed starting at time  $t$  regarding both movement on the grid and updating of its state. We call the 2-dimensional vector  $c_{t'}(a) - c_t(a) = (x_{t'}(a) - x_t(a), y_{t'}(a) - y_t(a))$  the *travel vector* of agent  $a$  (from time  $t$  to time  $t'$ ). Moreover, we call the time difference  $t' - t$  the *travel period*.

Note that travel vector and travel period do not depend on the choice of  $t$  and  $t'$  (provided  $t$  and  $t'$  satisfy the properties mentioned above). In the case of multiple agents, we use the same definitions for any time segment where only a single agent is scheduled and does not encounter another agent. In particular, we can only speak of a travel vector and a travel period when there are two points in time (in the considered time segment) where the scheduled agent repeats a state and at both times as well as in the time between, the agent is alone in its cell.

## 4 Techniques

In order to show our main result, we use a (large) proof by contradiction. In the following we give a (very informal and possibly slightly inaccurate) high-level overview of how it proceeds. Our assumption, that holds throughout the remainder of the paper, is that three agents actually suffice to explore the grid. From this assumption, we derive a contradiction as follows:

First, we fix an adversarial schedule for the three agents that has certain advantageous properties. (We will show that it is already possible to derive a contradiction for this specific schedule.) Then, using the finiteness of the number of configurations of agents in any bounded area, we show that for each distance  $D$  there is a point in time such that from this time onwards, there are always at least two agents that have distance at least  $D$ . However, since we can prove that any two agents must meet infinitely often, there must be infinitely many travels between the two far-away agents (which are not always the same agents). We show that the vector along which such a travel takes place must have a fixed slope that is the same for all such travel vectors (from a sufficiently large point in time on). Otherwise, there would exist two subsequent travels forth and back of different slope, which would imply that the traveling agent on its way back would miss the agent it is supposed to meet (which is the agent from whose position the first of the two travels started, roughly speaking). This also holds if the traveling agent explores some area to the left and right of its travel direction (during its travel), since the distance  $D$  between the two endpoints can be made arbitrarily large.

The crucial part of the proof is to show that the state of the traveling agent at the end of its travel does not depend on the exact vector between the start and the endpoint of its travel, but only on this vector “modulo” some other vector  $v$  that is obtained by combining all of the finitely many possible traveling vectors of the aforementioned fixed slope. Proving this statement enables us to show that, at the start of a travel, the information 1) about the states and relative locations “modulo  $v$ ” of the agents, and 2) about which agent is scheduled next and which is the traveling agent, are sufficient to determine the same information at the start of the next travel. Since there are only finitely many of these information tuples (exactly because they contain only the modulo version of the relative locations), at some point a tuple has to occur again. Hence, in a sense, the whole configuration consisting of

the three agents repeats its previous movement from this point on, at least if one ignores any movement in the direction of the fixed slope. Thus, in each repetition between two occurrences of the information tuple, the whole configuration moves by some fixed (and always the same) vector, which implies that the agents explore “at most half” of the grid.

## 5 The Schedule

From this section on, we assume that three semi-synchronous agents whose behavior is governed by a finite automaton suffice to explore the grid. Let  $a_1$ ,  $a_2$  and  $a_3$  be these agents. We start our proof by contradiction by specifying a schedule that we assume to be the adversarial schedule for the remainder of this paper:

We first schedule agent  $a_1$  for some number of time steps, then agent  $a_2$ , then  $a_3$ , and then we iterate, again starting with  $a_1$ . The number of steps an agent is scheduled can vary. In other words, we can describe our schedule as a sequence

$$\mathcal{S} = (\mathcal{S}_1^1, \mathcal{S}_1^2, \mathcal{S}_1^3, \mathcal{S}_2^1, \mathcal{S}_2^2, \mathcal{S}_2^3, \mathcal{S}_3^1, \dots)$$

of subschedules where in each subschedule  $\mathcal{S}_j^i$  only agent  $a_i$  is scheduled. The number of time steps in a subschedule  $\mathcal{S}_j^i$  is determined as follows:

1. If there is a (finite) number  $u > 0$  of time steps after which agent  $a_i$  is in a cell occupied by another agent, then the subschedule  $\mathcal{S}_j^i$  ends after  $u_{\min}$  time steps where  $u_{\min}$  denotes the smallest such  $u$ .
2. If Case 1 does not apply, but there is a (finite) number  $u > 0$  of time steps after which  $a_i$  is in the same state in the same cell as it was at some earlier point in time during  $\mathcal{S}_j^i$ , then do the following:

Fix a total order on the state space of  $a_i$ 's finite automaton. (This total order can be chosen arbitrarily, but in each application of Case 2 for agent  $a_i$  the same order has to be used.) Let  $q$  be the smallest state according to this order which  $a_i$  assumes at least twice in the same cell (if we scheduled  $a_i$  indefinitely). Then  $\mathcal{S}_j^i$  ends after the smallest positive number of steps after which  $a_i$  is in state  $q$  and in a cell where  $a_i$  would assume  $q$  at least twice. Note that the property that  $a_i$  would assume  $q$  twice implies that it would repeat the exact behavior between the first and the second assumption of  $q$  infinitely often afterwards, thus iterating through the exact same movement on and on.

3. If none of the two above cases occurs, i.e.,  $a_i$  would move on indefinitely without meeting any other agent or being in the same state in the same cell as before, then we schedule as follows: Let  $(x, y)$  be the travel vector of  $a_i$ 's movement, and  $k$  the travel period. Then the subschedule  $\mathcal{S}_j^i$  ends at the first time  $t$  (strictly after the start of  $\mathcal{S}_j^i$ ) for which the following property is satisfied:

For each cell  $(x_t^r, y_t^r)$  occupied by an agent  $a_r$ ,  $r \neq i$ , we have that 1)  $x_t^i - x_t^r > k$  if  $x > 0$ , and  $x_t^i - x_t^r < -k$  if  $x < 0$ , and 2)  $y_t^i - y_t^r > k$  if  $y > 0$ , and  $y_t^i - y_t^r < -k$  if  $y < 0$ . The definition of the travel vector ensures that there is such a (finite) point in time  $t$ . Note that Case 3 can only occur if  $x \neq 0$  or  $y \neq 0$ <sup>4</sup>. Moreover, if this case actually occurs, then the complete subsequent schedule is adapted according to the following special rule (overriding all of the above): After time  $t$ , the two agents  $a_r$ ,  $r \neq i$ , are scheduled for one time step each (in arbitrary order), then agent  $a_i$  is scheduled for  $k$  time steps, i.e., exactly one travel period, and then we iterate this new scheduling.

<sup>4</sup> If  $x = y = 0$ , agent  $a$  stays within a constant distance from the cell where the subschedule started. Hence, if Case 1 does not occur, every state/cell combination possible within this constant distance is assumed implying that Case 2 must occur.





■ **Figure 1** In Figure 1a, Case 2 of our schedule is shown. Note that the agent already stops when it visits the cell on the right (in state  $q$ ) for the first time (unless this happens after 0 time steps). In Figure 1b, we see Case 3 of our schedule. One agent would move arbitrarily far away if scheduled sufficiently long. By letting this agent move away far enough and then scheduling it sufficiently often for a long enough period of time, we make sure that it will not interact anymore with any of the other two agents.

Observe that according to this schedule, the number of time steps a scheduled agent can stay put in a cell during one of its subschedules is upper bounded by  $N$ . Also note that in each of the three cases, the number of steps in the subschedule is positive (and finite). For an illustration of Cases 2 and 3, see Figure 1. We now collect a few lemmas that highlight certain properties of the three cases.

► **Lemma 2.** *Case 3 cannot occur.*

**Proof.** Recall that we assume (globally) that the three agents explore the entire infinite grid. Assume that Case 3 occurs and let  $a_i$  denote the agent that would move on indefinitely without meeting another agent. Then, at the beginning of the first iteration according to the special rule, the distance of agent  $a_i$  to any of the other agents is more than  $k$  in at least one (of  $x$ - and  $y$ -) direction and  $a_i$  moves away from the agents according to the travel vector. After each of the other agents makes a step, this distance is still at least  $k$ . Hence, agent  $a_i$  cannot encounter one of the other agents during its next  $k$  steps, since in total it moves away from the other agents, according to the specification of Case 3.

The direction of the travel vector also ensures that the distance to the other agents is again increased to more than  $k$  (in at least one direction). Thus, the same arguments hold for the next iteration, and we obtain by induction that agent  $a_i$  will never encounter another agent after the occurrence of Case 3. It follows that, if three agents suffice to explore the grid, then also a team of two agents and a separate single agent can explore the grid without any communication between the team and the single agent. From [16], we know that this is not possible since a team of two agents (hence, also a single agent) can only explore a band of constant width. ◀

Following Lemma 2, we will assume in the following that Case 3 does not occur, i.e., each agent's subschedule ends because it encounters another agent or because it repeats a pair state/cell. This allows us to group the possible subschedules of an agent into two categories: We say that a subschedule  $\mathcal{S}_j^i$  is of *type 1* if  $\mathcal{S}_j^i$  ends because of the condition given in Case 1, and of *type 2* if  $\mathcal{S}_j^i$  ends because of the condition given in Case 2.

► **Lemma 3.** *Any subschedule of type 2 consists of at most  $N$  time steps.*

**Proof.** Assume for a contradiction that there is a subschedule  $\mathcal{S}_j^i$  of type 2 that consists of at least  $N + 1$  time steps and starts at some time  $t$ . Then, by the pigeonhole principle, there must be two points in time  $t < t' < t'' \leq t + N + 1$  such that  $q_{t'}^i = q_{t''}^i$ . Moreover, it must also hold that  $c_{t'}^i = c_{t''}^i$ , since otherwise  $a_i$  would move according to some non-zero travel vector (from time  $t'$  onwards) which would imply that  $\mathcal{S}_j^i$  is not of type 2.



This implies that if  $a_i$ 's subschedule would also continue at and after time  $t + N + 1$  on an empty grid, then  $a_i$  would cycle through the same movement on and on, starting from time  $t'$ . Hence, if there is a cell  $c$  that is visited by  $a_i$  in some state  $q$  in the (continued) movement after time  $t''$ , then there must also be a point in time before  $t''$  (during  $\mathcal{S}_j^i$ ) at which  $a_i$  visits  $c$  in state  $q$ . It follows from the definition of our schedule that  $\mathcal{S}_j^i$  ends before time  $t''$ , yielding a contradiction to our assumption. ◀

► **Lemma 4.** *Any subschedule  $\mathcal{S}_j^i$  of type 1, where agent  $a_i$  ends in the same cell from which it started, consists of at most  $N(2N + 1)$  time steps. More generally, any subschedule  $\mathcal{S}_j^i$  of type 1, where  $a_i$  ends in a cell of distance at most  $D$  from the cell from which it started, consists of at most  $N(2N + 1 + D)$  time steps.*

**Proof.** We start by proving the special case where  $a_i$  ends in the same cell from which it started. Suppose for a contradiction that there is a subschedule  $\mathcal{S}_j^i$  as described in the lemma that consists of more than  $N(2N + 1)$  time steps. Let  $t$  and  $u$  denote the points in time when  $\mathcal{S}_j^i$  starts and ends, respectively. Since  $a_i$  does not encounter any other agent between time  $t$  and time  $u$ , it behaves like a single agent on an empty grid between  $t$  and  $u$ . In particular, there is a travel vector  $(x, y)$  of agent  $a_i$  from time  $t + 1$  to time  $u - 1$  since  $N(2N + 1) - 1 > N$ .

For reasons of symmetry, we can assume w.l.o.g. that  $x > 0$  and  $y \geq 0$ . Note that  $x = 0 = y$  is not possible since in that case  $a_i$  would cycle through the same (cyclic) movement over and over without meeting any other agent, which would imply that  $\mathcal{S}_j^i$  is not of type 1. Let  $p$  be the travel period which, according to its definition, is at most  $N$ . Let  $q$  be the state whose second occurrence during  $\mathcal{S}_j^i$  (excluding the occurrence of the state at the beginning of  $\mathcal{S}_j^i$ ) comes earliest. Let  $t'$  be the time when  $q$  occurs for the first time. Since  $t' \leq t + N$ , we know that  $x_{t'}^i \geq x_t^i - N$ .

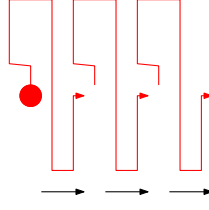
Now, as in each travel period  $a_i$  increases the  $x$ -coordinate of the cell it occupies by at least 1, it follows that at time  $t' + 2N \cdot p$  the  $x$ -coordinate of the cell  $a_i$  occupies is at least  $x_{t'}^i + N$ . Furthermore, since in each further travel period agent  $a_i$  would advance by at least one cell in (positive)  $x$ -direction in total and  $p \leq N$ , after time  $t' + 2N \cdot p$  agent  $a_i$  will never have an  $x$ -coordinate of less than  $x_t^i + 1$ , i.e., it will never reach  $c_t^i$  then. But  $a_i$  also cannot have visited  $c_t^i (= c_u^i)$  between time  $t + 1$  and  $t' + 2N \cdot p$  since  $t' + 2N \cdot p \leq t + N(2N + 1)$  and we assumed that  $\mathcal{S}_j^i$  consists of more than  $N(2N + 1)$  time steps. Thus, we obtain a contradiction, which proves the first lemma statement.

For the more general second statement, by an analogous proof we obtain that after time  $t' + 2N \cdot p + D \cdot p$  agent  $a_i$  will never have an  $x$ -coordinate of less than  $x_{t'}^i + 1 + D$ , i.e., it will never reach  $c_u^i$  then. But, since  $t' + 2N \cdot p + D \cdot p \leq t + N(2N + 1 + D)$ ,  $a_i$  also cannot have visited  $c_u^i$  between time  $t + 1$  and  $t' + 2N \cdot p + D \cdot p$ , under the assumption that  $\mathcal{S}_j^i$  consists of more than  $N(2N + 1 + D)$  time steps. Hence, this assumption must be false, and the lemma statement follows. ◀

## 6 Traveling and Meeting

Having defined and studied the schedule, we now proceed with our lower bound proof as described in Section 4. The next lemma shows that for each distance there is a point in time after which the farthest two agents are never closer than this distance.

► **Lemma 5.** *For each distance  $D$  there is a time  $T$  such that at any time  $t \geq T$  the largest pairwise distance of the three agents is at least  $D$ .*



■ **Figure 2** An example showing a possible movement (red) of an agent whose travel vector is given by the black arrows. The agent performs the total movement given by the travel vector in at most  $N$  time steps, or more precisely, during one travel period.

**Proof.** Suppose that the lemma statement is not true. Then there is an infinite sequence  $\mathcal{T}$  of points in time such that at each of these points in time the largest pairwise distance of the three agents is less than  $D$ . Since the distances of the agents are less than  $D$  at all points in time from  $\mathcal{T}$  and the number of states the three agents can be in is finite, it follows that there must be points in time  $t, t' \in \mathcal{T}$  such that 1) each agent is in the same state at  $t$  and  $t'$ , 2)  $x_t^i - x_{t'}^i = x_t^j - x_{t'}^j$  and  $y_t^i - y_{t'}^i = y_t^j - y_{t'}^j$  for all  $i, j \in \{1, 2, 3\}, i \neq j$ , and 3) the same agent is scheduled to move next. Since the agents are oblivious of the absolute coordinates of the grid, this implies that from time  $t'$  on, the agents will repeat the exact behavior they showed starting at time  $t$ . (Note that we use here that the schedule following a configuration is uniquely determined by the above information.) Hence, at time  $t' + (t' - t)$  the agents will again be in the exact same configuration and so on.

Define  $(x, y) = (x_{t'}^i - x_t^i, y_{t'}^i - y_t^i)$ , where  $i = 1$  (which implies that this equation also holds for  $i = 2, 3$ ). Vector  $(x, y)$  describes the total movement of each of the agents during each of the (repeating) time periods of length  $t' - t$ . It follows that each cell that has not been explored by time  $t$  must be at distance at most  $t' - t$  from some cell that is obtained by adding a multiple of the vector  $(x, y)$  to one cell from  $\{c_t^1, c_t^2, c_t^3\}$ ; otherwise it will never be explored. Since each such cell at distance at most  $t' - t$  (which is constant) must lie in a band of constant width and “direction”  $(x, y)$  that contains  $c_t^1, c_t^2$  or  $c_t^3$ , there are infinitely many cells that must have been explored before time  $t$ . This yields a contradiction. ◀

For any distance  $D$ , we denote by  $T_D$  the smallest time  $T$  for which it holds that at any time  $t \geq T$  the largest pairwise distance of the three agents is at least  $D$ . In the following we collect a number of useful definitions regarding the meetings of different agents. In particular, we distinguish between three different types of agents at times when one agent is traveling from another agent to the far-away agent whose existence is certified by Lemma 5. For an illustration of how a large distance between agents influences choices of travel vectors, see Figure 2.

► **Definition 6.** For any  $t \geq 0$ , we define the *meeting set*  $M_t$  as the set of agents that are not alone in the cell they occupy, at time  $t$ . We call the infinite sequence  $(M_0, M_1, \dots)$  the *meeting sequence*. If for a subsequence  $(M_t, M_{t+1}, \dots, M_{t+i})$  of the meeting sequence it holds that  $i > 0$ ,  $M_t \neq \emptyset \neq M_{t+i}$  and  $M_{t+j} = \emptyset$  for all  $0 < j < i$ , then we call the pair  $(t, t+i)$  a *meeting pair*. Now, let  $(t, u)$  be a meeting pair such that  $|M_t| = 2 = |M_u|$  and  $M_t \neq M_u$ . Then we call  $(t, u)$  a *travel meeting pair*. Moreover, we call the (uniquely defined) agent  $a$  contained in  $M_t \cap M_u$  a *traveling agent (for  $(t, u)$ )*, the agent contained in  $M_t \setminus \{a\}$  a *source agent* and the agent contained in  $M_u \setminus \{a\}$  a *destination agent*.

In order to continue according to our high-level proof idea from Section 4, we need a few helping lemmas that highlight properties of the previous definitions. We start with a lemma that shows an important property of the meeting sequence:

► **Lemma 7.** *Each of the three agents is contained in infinitely many of the  $M_t$  from the meeting sequence.*

**Proof.** Suppose that there is an agent  $a_i$  that is not contained in infinitely many of the  $M_t$ , i.e., there is a point in time  $u$  such that  $a_i \notin M_t$  for all  $t \geq u$ . Then, starting from time  $u$ , the exploration by the two agents  $a_r, r \neq i$  is entirely independent of the exploration by agent  $a_i$  since they never meet again. Thus, we get a contradiction analogously to the argumentation in the proof of Lemma 2. ◀

Next, we study travel meeting pairs more closely. In Lemma 8, we present bounds on the number of subschedules of the different types of agents in the time frame given by a travel meeting pair, and examine the types of the subschedules. Afterwards, in Lemma 9, we bound the number of time steps between two subsequent travel meeting pairs from above. In both cases, the results only hold from a large enough point in time onwards, but this is sufficient for our purposes since before that point in time only a constant number of cells were explored. Note that, in general, we do not attempt to minimize the dependence on  $N$  in our bounds as showing the finiteness of certain parameters is, again, sufficient for our purposes. Instead we prefer to choose the simplest arguments that lead to the desired finiteness results, even if they augment the actual bound by a few factors of  $N$ .

► **Lemma 8.** *There is a point in time  $T$  such that, for each travel meeting pair  $(t, u)$  with  $t \geq T$ , the following properties hold:*

1. *The traveling agent for  $(t, u)$  is scheduled exactly once (for a number of time steps) between time  $t$  and time  $u$ .*
2. *The subschedule of the traveling agent is of type 1 and ends exactly at time  $u$ .*
3. *The source and the destination agent for  $(t, u)$  are scheduled at most once (for a number of time steps).*
4. *If the source or the destination agent is scheduled, then its subschedule is of type 2.*

**Proof.** Recall the definition of  $T_D$  for any distance  $D$ . Let  $T \geq T_{2N+1}$ , and consider an arbitrary travel meeting pair  $(t, u)$  with  $t \geq T$  and traveling agent  $a_i$ . Observe that if the source agent is scheduled between time  $t$  and time  $u$ , then its subschedules must be of type 2, because the source agent is not contained in the meeting set  $M_u$ . Hence, if  $a_i$  is not scheduled at all between time  $t$  and time  $u$ , then the source agent must be scheduled at most once (because of the specification of our schedule) which implies that its distance from  $c_t^i$  at time  $u$  is at most  $N$ , by Lemma 3. But since in this case  $a_i$  and the destination agent meet at  $c_t^i$  at time  $u$ , we obtain a contradiction to the fact that  $T \geq T_{2N+1}$ . Thus, we know that  $a_i$  is scheduled at least once between time  $t$  and time  $u$ .

Now, assume for a contradiction that the first subschedule of  $a_i$  between time  $t$  and time  $u$  is of type 2. This implies that if one would schedule  $a_i$  on and on, it would repeat a state in the same (empty) cell after at most  $N + 1$  time steps and then cycle through (a part of) the same movement it performed before. Hence, even if there are more subschedules for  $a_i$  than one (between time  $t$  and time  $u$ ), it will never reach a cell that has a distance of more than  $N$  from  $c_t^i$ . Since analogous statements hold for the source agent, we know that at time  $u$  the distance between the source agent and the cell where  $a_i$  and the destination agent meet is at most  $2N$  which again contradicts our specification of  $T$ . Thus, we know that the first subschedule of  $a_i$  is of type 1.

It follows that  $a_i$ 's subschedule ends exactly at time  $u$  since the subschedule must end with  $a_i$  meeting the destination agent, which also implies that  $a_i$  is scheduled exactly once between time  $t$  and time  $u$ . Moreover, the subschedules of the source and the destination

agent (if they are scheduled at all between time  $t$  and time  $u$ ) must be of type 2 since  $(t, u)$  is a (travel) meeting pair. Furthermore, by the nature of our schedule, the source and the destination agent must be scheduled at most once between time  $t$  and time  $u$ . ◀

► **Lemma 9.** (Proof deferred to the full version) *There is a point in time  $T$  such that the following holds: If  $(t, u)$  and  $(t', u')$  are travel meeting pairs such that  $T \leq t < t'$  and there exists no travel meeting pair  $(t'', u'')$  with  $t < t'' < t'$ , then  $t' - u \leq 8(N + 1)^5$ .*

Using Lemma 9, we show in the following that for any travel meeting pair  $(t, u)$ , the information about the states of the agents, which two agents are in the same cell, and who is scheduled next, all at time  $u$ , already uniquely determines a lot of information about the agents at the starting time of the next travel meeting pair. Again, this result only holds from a sufficiently large point in time onwards. This concludes our collection of helping lemmas.

► **Lemma 10.** (Proof deferred to the full version) *There is a point in time  $T$  such that the following holds: For any two subsequent travel meeting pairs  $(t, u)$ ,  $(t', u')$  with  $T \leq t < t'$ , the tuple  $(q_u^1, q_u^2, q_u^3, a_u^{next}, M_u)$  uniquely determines the tuple  $(q_{t'}^1, q_{t'}^2, q_{t'}^3, c_{t'}^1 - c_u^1, c_{t'}^2 - c_u^2, c_{t'}^3 - c_u^3, a_{t'}^{next}, M_{t'})$ , where  $a_u^{next}$ , resp.  $a_{t'}^{next}$ , denotes the agent scheduled at time  $u$ , resp.  $t'$ .*

## 7 The Travel Vector and a Modulo Operation

After collecting the above helping lemmas, we are now all set to formally prove the (remaining) statements from our proof sketch. Before going through the statements one by one, let us for convenience define the notion of a travel: Let  $(t, u)$  be a travel meeting pair. By Lemma 8, we know that the traveling agent for  $(t, u)$  is scheduled exactly once between  $t$  and  $u$ . We call the corresponding subschedule (or the movement during that subschedule) a *travel*. Recall the definition of travel vector and travel period. Note that a travel only has a travel vector (and period) if the traveling agent repeats a state (in empty cells) during the travel. Furthermore, observe that if a travel has a travel vector, then at least one entry of the travel vector is non-zero, due to the choice of our schedule. We now prove the first of the remaining statements, namely, that after a certain point in time, any travel vector has the same slope.

► **Lemma 11.** *There is a point in time  $T$  and a (possibly negative) ratio  $r$  such that each travel starting at time  $T$  or later has travel vector  $(x, y)$  with  $y/x = r$ . For the sake of simplicity, assume that  $r$  is set to  $\infty$  if  $x = 0$ .*

**Proof.** Let  $T$  be sufficiently large so that  $T \geq T_{N+2}$  holds and Lemma 8 and Lemma 9 apply. Then we know that any travel starting at time  $T$  or later actually has a travel vector (and period). Now, consider two travel meeting pairs  $(t, u)$  and  $(t', u')$  with  $T \leq t < t'$  such that there is no travel meeting pair  $(t'', u'')$  with  $t < t'' < t'$ . Let  $(x, y), (x', y')$  be the travel vectors for the travels corresponding to  $(t, u)$  and  $(t', u')$ , respectively. Assume that  $y'/x' \neq y/x$ , where, again, we set the ratio to  $\infty$  if the denominator is 0. Note that not both of  $x$  and  $y$  (or  $x'$  and  $y'$ ) can be 0. Let  $c_0$  and  $c_1$  be the cells at which the travel with travel vector  $(x, y)$  starts and ends, respectively, and  $c'_0$  and  $c'_1$  analogously for the travel with travel vector  $(x', y')$ .

By the characterization of the travel of a single agent and the fact that the travel period is always at most  $N$ , we know that there are positive integers  $b$  and  $b'$  such that  $\text{Dist}(c_1, c_0 + b \cdot (x, y)) \leq N$  and  $\text{Dist}(c'_1, c'_0 + b' \cdot (x', y')) \leq N$ . Moreover, by Lemma 3 and Lemma 8, the source agent for  $(t, u)$  travels at most a distance of  $N$  between time  $t$  and  $u$  since its subschedule is of type 2 if the agent is scheduled at all. The same holds for the destination agent for  $(t', u')$  between time  $t'$  and  $u'$ . By Lemma 9, it follows that

$\text{Dist}(c_0, c'_1) \leq 8(N+1)^5 + 2N$  (since the source agent for the first of the two travels is the destination agent for the second) and  $\text{Dist}(c_1, c'_0) \leq 8(N+1)^5$ . Combining our above distance observations, we also obtain  $\text{Dist}(c'_1, c_0 + b \cdot (x, y) + b' \cdot (x', y')) \leq N + 8(N+1)^5 + N$ , which together with  $\text{Dist}(c_0, c'_1) \leq 8(N+1)^5 + 2N$  implies  $\text{Dist}(c_0, c_0 + b \cdot (x, y) + b' \cdot (x', y')) \leq 16(N+1)^5 + 4N$ .

Let  $D \geq N$  be some positive integer. We now require, additionally to the above requirements regarding  $T$ , that  $T \geq T_D$ . Also fix some arbitrary  $x, y, x', y'$  such that  $(x, y)$  and  $(x', y')$  are possible travel vectors of a single agent. For a contradiction, assume that  $x, y, x', y'$  have the properties specified at the beginning of the proof (which implies that also all of the above conclusions hold).

At the time when the first of the two considered travels starts there are two agents at  $c_0$  and  $c_1$  while the last agent is in distance at most  $N$  from  $c_0$ . Hence, the distance between  $c_0$  and  $c_1$  is at least  $D - N$ . This implies that  $b \cdot (|x| + |y|) \geq \text{Dist}(c_1, c_0) - N \geq D - 2N$ . Analogously, we obtain  $b' \cdot (|x'| + |y'|) \geq D - 2N$ . Since  $x, y, x', y'$  are fixed, we can therefore make  $b$  and  $b'$  arbitrarily large by increasing  $D$ . By increasing  $b$  and  $b'$ , we can in turn make  $\text{Dist}(c_0, c_0 + b \cdot (x, y) + b' \cdot (x', y'))$  arbitrarily large, since  $y'/x' \neq y/x$  (which implies that there is an angle between the two vectors  $(x, y)$  and  $(x', y')$  that is not  $0^\circ$  or  $180^\circ$ ). Hence, if  $D$  is sufficiently large, then the above inequality  $\text{Dist}(c_0, c_0 + b \cdot (x, y) + b' \cdot (x', y')) \leq 16(N+1)^5 + 4N$  is not satisfied anymore, which shows that  $y'/x' = y/x$ .

Note that the magnitude  $D$  has to reach for this (in our proof by contradiction) depends on  $x, y, x', y'$ . However, since the number of possible travel vectors of a single agent is bounded by the number of states in its finite automaton, we can simply derive a sufficiently large  $D$  for each of the finitely many possible combinations for  $x, y, x', y'$  and then choose a  $T$  that is larger than all of the  $T_D$ . ◀

Note that the exact value of  $r$  depends only on the finite automaton governing the behavior of the three agents. From now on, we denote the ratio whose existence is certified by Lemma 11 by  $r$ . W.l.o.g., we can (and will) assume that  $r \geq 0$  (and that  $r \neq \infty$ ), for reasons of symmetry. Recall that any travel vector has at least one non-zero entry. The next step on our agenda is essentially to show that the state of an agent at the end of a travel does not depend on (the full information about) the vector between start and endpoint of that travel (and other parameters), but only on a reduced amount of information regarding this vector (and the other parameters). More specifically, the required information about this vector is the result of applying a certain modulo operation to the vector.

We then proceed by showing that the information about 1) the states of the agents, 2) their relative locations after applying the modulo operation, 3) which agents shared a cell most recently, and 4) which agent is scheduled next, at the start of a travel, is enough to determine the exact same information at the end of the travel. Now, we benefit from the previous reduction of information due to our modulo operation in the sense that we can show that there are only constantly many combinations of relative locations of the three agents (that can actually occur) after applying the modulo operation. This, in turn, implies that there are only constantly many possibilities for the whole aforementioned information tuple at the start and end of a travel, which will enable us to prove our main theorem. We start by defining our modulo operation in Definition 12. Then we show a technical helping lemma, Lemma 13, which finally enables us to prove the aforementioned relation between the information tuple at the start and end of a travel in Lemma 14. Note that for technical reasons, Lemma 14 gives a slightly different statement than indicated above, dealing with travel meeting pairs instead of travels.

► **Definition 12.** Let  $\{(x_1, y_1), (x_2, y_2), \dots, (x_k, y_k)\}$  be the set of travel vectors that the agents can have if you let one of them explore the grid starting in an arbitrary state (which clearly is a superset of the actually occurring travel vectors in our multi-agent case). Let  $R$  be the subset of the above set that contains exactly the vectors  $(x_j, y_j)$  that satisfy  $y_j/x_j = r$ . From now on, denote by  $x$  the least common multiple of the  $|x_j|$  from the vectors in  $R$  and set  $y := rx$ . It follows that  $(x, y)$  is a (possibly negative) integer multiple of any of the vectors from  $R$ . Note that  $R$  cannot be empty since otherwise it is not possible that the agents explore the entire grid, due to Lemma 7 and Lemma 11.

Now, let  $w, z$  be integers and let  $b$  be the smallest integer such that  $w + bx \geq 0$ . (This is well-defined since  $x > 0$ , due to  $r \neq \infty$ .) We define  $(w, z) \pmod{(x, y)} := (w + bx, z + by)$ . For two cells  $(w', z'), (w'', z'')$ , we define  $(w'', z'') \ominus (w', z') := (w'' - w', z'' - z') \pmod{(x, y)}$ .

Note that Definition 12 ensures that for any  $(w, z), (w', z')$  where  $(w' - w, z' - z)$  is a multiple of  $(x, y)$ , we have that  $(w, z) \pmod{(x, y)} = (w', z') \pmod{(x, y)}$ .

► **Lemma 13.** *Let  $a$  be an agent,  $q$  a state from  $a$ 's finite automaton and  $c, c', c''$  cells of the grid such that the following properties are satisfied:*

1.  $\text{Dist}(c, c') \geq N$  and  $\text{Dist}(c'', c') \geq N$
2. *There is an integer  $b$  such that  $c'' - c = b \cdot (w, z)$ , where  $(w, z)$  is agent  $a$ 's travel vector if it starts in state  $q$ .*
3. *If agent  $a$  starts in cell  $c$  in state  $q$  on an otherwise empty grid, then it arrives at  $c'$  after finite time.*
4. *If agent  $a$  starts in cell  $c''$  in state  $q$  on an otherwise empty grid, then it arrives at  $c'$  after finite time.*

*Let  $q'$  denote the state in which  $a$  arrives at  $c'$  (for the first time) when starting from  $c$  (in state  $q$ ), and  $q''$  the state in which  $a$  arrives at  $c'$  (for the first time) when starting from  $c''$  (in state  $q$ ). Then it holds that  $q' = q''$ .*

**Proof.** If  $c = c''$ , then the lemma holds trivially, thus assume that  $c \neq c''$ . W.l.o.g., we can assume that  $b > 0$ , which implies that, if agent  $a$  starts in cell  $c$  in state  $q$  (say, at time  $t$ ), then  $a$  arrives at some point in time  $u > t$  in cell  $c''$  in state  $q$  (possibly  $a$  visited  $c''$  before in some other state). Hence, if  $a$  does not visit cell  $c'$  between time  $t$  and time  $u$ , then the lemma also holds since after arriving at  $c''$  in state  $q$ ,  $a$  will perform the exact same movement as if it started in  $c''$  in state  $q$ .

Thus, consider the last remaining case, i.e., assume that  $a$  visits  $c'$  for the first time at some time  $t < t' < u$ . W.l.o.g., we can assume that  $w$  and  $z$  are non-negative and  $w \geq z$ . (Also recall that at least one of  $w$  and  $z$  is non-zero.) Let  $c_0, c_1, \dots$  be the cells that  $a$  visits in state  $q$  at and after time  $t$ , where  $c_0$  and  $c_k$ , for some  $k > 0$ , are the cells that  $a$  visits at time  $t$  and  $u$ , respectively, i.e.,  $c_0 = c$  and  $c_k = c''$ . Observe that  $c_{j+1} = c_j + (w, z)$  holds for each  $j$ . Denote the  $x$ -coordinates of  $c'$  and  $c_k = c''$  by  $x'$  and  $x''$ , respectively. Since  $w \geq z$ , it follows that  $\text{Dist}(c_j, c') \geq \text{Dist}(c'', c') \geq N$  for all  $j \geq k$  if  $x' \leq x''$ , and  $\text{Dist}(c_j, c') \geq \text{Dist}(c'', c') \geq N$  for all  $0 \leq j \leq k$  if  $x' \geq x''$ . Let  $h$  be the largest index such that  $a$  visits  $c_h$  in state  $q$  at or before time  $t'$ . Then  $h < k$ , and  $\text{Dist}(c_h, c') \leq N - 1$  since traveling from  $c_h$  (in state  $q$ ) to  $c_{h+1}$  (in state  $q$ ) takes  $a$  at most one travel period, so at most  $N$  time steps. If  $x' \geq x''$ , then we obtain a contradiction to our above observation, thus it follows that  $x' < x''$ . But this implies  $\text{Dist}(c_j, c') \geq N$  for all  $j \geq k$  which in turn implies for all  $j \geq k$  that  $c'$  cannot be visited by  $a$  between visiting  $c_j$  (in state  $q$ ) and  $c_{j+1}$  (in state  $q$ ). Hence,  $a$  does not visit  $c'$  at or after time  $u$ . Since  $a$  performs the exact same movement



from time  $u$  onwards as if it would have initially started in  $c''$  in state  $q$ , it follows that agent  $a$  starting in  $c''$  in state  $q$  never visits  $c'$ , which is a contradiction to our assumptions. Thus, this last remaining case cannot occur, which completes the proof.  $\blacktriangleleft$

► **Lemma 14.** (Proof deferred to the full version) *Let  $(t, u)$  be a travel meeting pair. Consider the tuple  $Q_t := (q_t^1, q_t^2, q_t^3, c_t^1 \ominus c_t^2, c_t^1 \ominus c_t^3, c_t^2 \ominus c_t^3, a_t^{next}, M_t)$ , where  $a_t^{next}$  again denotes the agent that is scheduled at time  $t$ . There is a point in time  $T$  such that the following holds: If  $t \geq T$ , then  $Q_t$  uniquely determines the tuple  $Q_u = (q_u^1, q_u^2, q_u^3, c_u^1 \ominus c_u^2, c_u^1 \ominus c_u^3, c_u^2 \ominus c_u^3, a_u^{next}, M_u)$ .*

## 8 Three Semi-Synchronous Agents Do Not Suffice

We now conclude our lower bound proof with Theorem 1. Roughly speaking, Lemma 14 certifies that the behavior of the agents between any two subsequent occurrences of the same fixed information tuple  $Q_t$  is reasonably similar. Since there are only finitely many different  $Q_t$  that actually occur, it follows that the behavior of the agents loops, in a very informal sense. From this, we can derive a contradiction to the assumption that all cells are explored.

► **Theorem 1.** *Three semi-synchronous agents controlled by a finite automaton are not sufficient to explore the infinite grid.*

**Proof.** Suppose for a contradiction that three agents suffice to explore the grid. From the definition of a travel meeting pair and Lemma 7, it follows that there are points in time  $t_1 < u_1 \leq t_2 < u_2 \leq t_3 < \dots$  such that  $(t_j, u_j)$  is a travel meeting pair for any  $j \geq 1$  and for every travel meeting pair  $(t', u')$  there is a  $j \geq 1$  with  $t' = t_j$  and  $u' = u_j$ .

Recall the definition of  $Q_t$  in Lemma 14. Let  $T$  be sufficiently large so that  $T \geq T_1$  holds (where  $T_1$  is just  $T_D$  for  $D = 1$ ) and Lemmas 8, 9, 10, 11 and 14 apply, and let  $k$  be an index such that  $t_k \geq T$  and there is a  $h > k$  with  $h - k$  even and  $Q_{t_k} = Q_{t_h}$ . Such a  $k$  must exist since there is only a finite number of tuples of the general form  $Q_t$  (after time  $T$ ) and the number of travel meeting pairs is infinite, by Lemma 7. Note that the finiteness of the number of tuples, in particular the finiteness of the (combinations of the) relative locations of the agents modulo  $(x, y)$ , relies on the fact that the possible travel vectors after time  $T$  are restricted by Lemma 11, together with the fact that in the time span given by a travel meeting pair source and destination agent are scheduled for at most  $N$  steps, by Lemma 3 and Lemma 8.

Consider the sequence  $((t_k, u_k), (t_{k+1}, u_{k+1}), \dots, (t_h, u_h))$  of travel meeting pairs, where  $h$  is the smallest index such that  $h > k$  holds,  $h - k$  is even, and  $Q_{t_k} = Q_{t_h}$ . We examine the cells that are explored by the source agent for  $(t_k, u_k)$  between time  $t_k$  and  $t_{k+1}$  and by the destination agent for  $(t_{k+1}, u_{k+1})$  (which is the same as the aforementioned source agent) between time  $t_{k+1}$  and  $t_{k+2}$ . Then we iterate this examination, in each iteration increasing the indices by 2, and stop at time  $t_h$ . We say that the cells explored in the described way are explored during *even* explorations.

In the first iteration, we obtain the following picture, where we denote the source agent for  $(t_k, u_k)$  (i.e., the destination agent for  $(t_{k+1}, u_{k+1})$ ) by  $a$ : The exact vector by which  $a$  moves between time  $t_k$  and  $u_k$  is uniquely determined by  $Q_{t_k}$ , as observed in the proof of Lemma 14. The exact vector by which  $a$  moves between time  $u_k$  and  $t_{k+1}$  is uniquely determined by  $Q_{u_k}$ , by Lemma 10. Similarly, the exact vectors by which  $a$  moves between time  $t_{k+1}$  and  $u_{k+1}$  and between time  $u_{k+1}$  and  $t_{k+2}$  are uniquely determined by  $Q_{t_{k+1}}$  and  $Q_{u_{k+1}}$ , respectively.

Moreover, by combining Lemma 10 and Lemma 14, we see that  $Q_{u_k}$ ,  $Q_{t_{k+1}}$ ,  $Q_{u_{k+1}}$ , and  $Q_{t_{k+2}}$  are all uniquely determined by  $Q_{t_k}$ . Thus, the exact vector by which  $a$  moves between time  $t_k$  and time  $t_{k+2}$  is uniquely determined by  $Q_{t_k}$ . Furthermore, by Lemma 3, Lemma 8,

and Lemma 9, the number of cells  $a$  visits between time  $t_k$  and time  $t_{k+2}$  is bounded by a constant. Note that each  $Q_{t_j}$  also uniquely determines which agent is the traveling agent (and hence which agent is the source/destination agent) for  $(t_j, u_j)$ , as observed in the proof of Lemma 14.

For the second, third,  $\dots$ , iteration we obtain an analogous picture. Hence, the tuples  $Q_{t_{k+2}}, Q_{t_{k+4}}, \dots$  are all uniquely determined by  $Q_{t_k}$ , and the locations of the respective source agents at times  $t_{k+2}, t_{k+4}, \dots$  are all uniquely determined by  $Q_{t_k}$  and the location of the source agent for  $(t_k, u_k)$  at time  $t_k$ .

We obtain the following bigger picture: The location of the source agent for  $(t_k, u_k)$  at time  $t_k$  together with  $Q_{t_k}$  uniquely determines both  $Q_{t_h}$  and the location of the source agent for  $(t_h, u_h)$  at time  $t_h$ , which, in turn, uniquely determine  $Q_{t_{h+(h-k)}}$  and the location of the source agent for  $(t_{h+(h-k)}, u_{h+(h-k)})$  at time  $t_{h+(h-k)}$ , and so on. Hence, there is a vector  $(w, z)$  such that the locations of the respective source agents at times  $t_k, t_h, t_{h+(h-k)}, t_{h+2(h-k)}, \dots$  are  $c, c + (w, z), c + 2(w, z), \dots$ , where  $c$  denotes the cell occupied by the respective source agent at time  $t_k$ . Moreover, since the number of cells explored during an even exploration between time  $t_k$  and  $t_h$  (and similarly between time  $t_{h+j(h-k)}$  and  $t_{h+(j+1)(h-k)}$  for each  $j \geq 0$ ) is bounded by a constant (which follows from a similar observation above), we get that there is a constant  $L$  such that each cell explored during an even exploration has a distance of at most  $L$  to some cell of the form  $c + j' \cdot (w, z)$ , where  $j'$  is some non-negative integer.

Moreover, by Lemmas 3, 8, 9, 11, and the definition of even explorations, we know that each explored cell is close to the travel of a traveling agent, i.e., there is a constant  $L'$  such that each cell explored at or after time  $t_k$  has a distance of at most  $L'$  to some cell of the form  $c' + j'' \cdot (x, y)$ , where  $j''$  is some integer and  $c'$  a cell explored during an even exploration. Combining our observations and adding the fact that only a constant number of cells are explored up to time  $t_k$ , it follows that there is a constant  $L''$  such that each cell explored by the agents has a distance of at most  $L''$  to some cell of the form  $c + j' \cdot (w, z) + j'' \cdot (x, y)$ , where  $j', j''$  are integers and  $j'$  is non-negative. Hence, we can draw a line in the grid such that all explored cells are to one side of the line, yielding a contradiction to the assumption that three agents suffice to explore the grid.  $\blacktriangleleft$

---

## 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 *FOCS*, pages 218–223, 1979.
- 3 Igor Averbakh and Oded Berman. A Heuristic with Worst-case Analysis for Minimax Routing of Two Travelling Salesmen on a Tree. *Discrete Appl. Math.*, 68(1-2):17–32, 1996.
- 4 Ricardo A. Baeza-Yates, Joseph C. Culberson, and Gregory J. E. Rawlins. Searching in the Plane. *Information and Computation*, 106:234–252, 1993.
- 5 M. A. Bender and D. K. Slonim. The Power of Team Exploration: Two Robots can Learn Unlabeled Directed Graphs. In *FOCS*, pages 75–85, 1994.
- 6 M. Blum and W. J. Sakoda. On the capability of finite automata in 2 and 3 dimensional space. In *FOCS*, pages 147–161, 1977.
- 7 Manuel Blum and Dexter Kozen. On the Power of the Compass (or, Why Mazes Are Easier to Search Than Graphs). In *FOCS*, pages 132–142, 1978.
- 8 Sebastian Brandt, Jara Uitto, and Roger Wattenhofer. Tight Bounds for Asynchronous Collaborative Grid Exploration. *CoRR*, abs/1705.03834, 2017. URL: <http://arxiv.org/abs/1705.03834>.



- 9 Lothar Budach. Automata and Labyrinths. *Mathematische Nachrichten*, 86(1):195–282, 1978.
- 10 Marek Chrobak, Leszek Gasiñec, Thomas Gorry, and Russell Martin. *Group Search on the Line*, pages 164–176. Springer Berlin Heidelberg, 2015. doi:10.1007/978-3-662-46078-8\_14.
- 11 Lihi Cohen, Yuval Emek, Oren Louidor, and Jara Uitto. Exploring an Infinite Space with Finite Memory Scouts. In *SODA*, pages 207–224, 2017.
- 12 Xiaotie Deng and Christos Papadimitriou. Exploring an Unknown Graph. *Journal of Graph Theory*, 32:265–297, 1999.
- 13 Krzysztof Diks, Pierre Fraigniaud, Evangelos Kranakis, and Andrzej Pelc. Tree Exploration with Little Memory. *Journal of Algorithms*, 51:38–63, 2004.
- 14 Yann Disser, Jan Hackfeld, and Max Klimm. Undirected Graph Exploration with  $\Theta(\log \log n)$  Pebbles. In *SODA*, pages 25–39, 2016.
- 15 Christian A. Duncan, Stephen G. Kobourov, and V. S. Anil Kumar. Optimal Constrained Graph Exploration. *ACM Trans. Algorithms*, 2(3):380–402, 2006.
- 16 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. doi:10.1016/j.tcs.2015.05.054.
- 17 Yuval Emek, Tobias Langner, Jara Uitto, and Roger Wattenhofer. Solving the ANTS Problem with Asynchronous Finite State Machines. In *ICALP*, pages 471–482, 2014.
- 18 Ofer Feinerman, Amos Korman, Zvi Lotker, and Jean-Sebastien Sereni. Collaborative Search on the Plane Without Communication. In *PODC*, pages 77–86, 2012.
- 19 P. Flocchini, G. Prencipe, N. Santoro, and P. Widmayer. Distributed Coordination of a Set of Autonomous Mobile Robots. In *Intelligent Vehicles Symposium*, pages 480–485, 2000.
- 20 Pierre Fraigniaud and David Ilcinkas. *Digraphs Exploration with Little Memory*, pages 246–257. Springer Berlin Heidelberg, 2004. doi:10.1007/978-3-540-24749-4\_22.
- 21 Frank Hoffmann. One Pebble Does Not Suffice to Search Plane Labyrinths. In *FCT*, pages 433–444, 1981.
- 22 Alejandro López-Ortiz and Graeme Sweet. Parallel Searching on a Lattice. In *CCCG*, pages 125–128, 2001.
- 23 Petrişor Panaite and Andrzej Pelc. Exploring Unknown Undirected Graphs. In *SODA*, pages 316–322, 1998.
- 24 H. A. Rollik. *Automaten in Planaren Graphen*, pages 266–275. Springer Berlin Heidelberg, Berlin, Heidelberg, 1979. doi:10.1007/3-540-09118-1\_28.
- 25 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.
- 26 Ichiro Suzuki and Masafumi Yamashita. Distributed Anonymous Mobile Robots: Formation of Geometric Patterns. *SIAM Journal on Computing*, 28(4):1347–1363, 1999.
- 27 Ichiro Suzuki and Masafumi Yamashita. Distributed Anonymous Mobile Robots - Formation and Agreement Problems. In *SIROCCO*, pages 1347–1363, 1996.



# Multi-Shot Distributed Transaction Commit

Gregory Chockler

Royal Holloway, University of London, UK

Alexey Gotsman<sup>1</sup>

IMDEA Software Institute, Madrid, Spain

---

## Abstract

---

Atomic Commit Problem (ACP) is a single-shot agreement problem similar to consensus, meant to model the properties of transaction commit protocols in fault-prone distributed systems. We argue that ACP is too restrictive to capture the complexities of modern transactional data stores, where commit protocols are integrated with concurrency control, and their executions for different transactions are interdependent. As an alternative, we introduce Transaction Certification Service (TCS), a new formal problem that captures safety guarantees of multi-shot transaction commit protocols with integrated concurrency control. TCS is parameterized by a certification function that can be instantiated to support common isolation levels, such as serializability and snapshot isolation. We then derive a provably correct crash-resilient protocol for implementing TCS through successive refinement. Our protocol achieves a better time complexity than mainstream approaches that layer two-phase commit on top of Paxos-style replication.

**2012 ACM Subject Classification** Theory of computation → Distributed computing models

**Keywords and phrases** Atomic commit problem, two-phase commit, Paxos

**Digital Object Identifier** 10.4230/LIPIcs.DISC.2018.14

## 1 Introduction

Modern data stores are often required to manage massive amounts of data while providing stringent transactional guarantees to their users. They achieve scalability by partitioning data into independently managed *shards* (aka *partitions*) and fault-tolerance by replicating each shard across a set of servers [9, 14, 42, 34]. Implementing such systems requires sophisticated protocols to ensure that distributed transactions satisfy a conjunction of desirable properties commonly known as ACID: Atomicity, Consistency, Isolation and Durability.

Traditionally, distributed computing literature abstracts ways of achieving these properties into separate problems: in particular, atomic commit problem (ACP) for Atomicity and concurrency control (CC) for Isolation. ACP is formalised as a *one-shot* agreement problem in which multiple shards involved in a transaction need to reach a decision on its final outcome: COMMIT if all shards voted to commit the transaction, and ABORT otherwise [13]. Concurrency control is responsible for determining whether a shard should vote to commit or abort a transaction based on the locally observed conflicts with other active transactions. Although both ACP and CC must be solved in any realistic transaction processing system, they are traditionally viewed as disjoint in the existing literature. In particular, solutions for ACP treat the votes as the inputs of the problem, and leave the interaction with CC, which is responsible for generating the votes, outside the problem scope [38, 2, 23, 16].

---

<sup>1</sup> Alexey Gotsman was supported by an ERC Starting Grant RACCOON.



© Gregory Chockler and Alexey Gotsman;

licensed under Creative Commons License CC-BY

32nd International Symposium on Distributed Computing (DISC 2018).

Editors: Ulrich Schmid and Josef Widder; Article No. 14; pp. 14:1–14:18

Leibniz International Proceedings in Informatics



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

This separation, however, is too simplistic to capture the complexities of many practical implementations in which commit protocols and concurrency control are tightly integrated, and as a result, may influence each other in subtle ways. For example, consider the classical *two-phase commit (2PC)* protocol [15] for solving ACP among reliable processes. A transaction processing system typically executes a 2PC instance for each transaction [32, 31, 34, 39]. When a process  $p_i$  managing a shard  $s$  receives a transaction  $t$ , it performs a local concurrency-control check and accordingly votes to commit or abort  $t$ . The votes on  $t$  by different processes are aggregated, and the final decision is then distributed to all processes. If  $p_i$  votes to commit  $t$ , as long as it does not know the final decision on  $t$ , it will have to conservatively presume  $t$  as committed. This may cause  $p_i$  to vote ABORT in another 2PC instance for a transaction  $t'$  conflicting with  $t$ , even if in the end  $t$  is aborted. In this case, the outcome of one 2PC instance (for  $t'$ ) depends on the internals of the execution of another instance (for  $t$ ) and the concurrency-control policy used.

At present, the lack of a formal framework capturing such intricate aspects of real implementations makes them difficult to understand and prove correct. In this paper, we take the first step towards bridging this gap. We introduce *Transaction Certification Service (TCS, §2)*, a new formal problem capturing the safety guarantees of a multi-shot transaction commit protocol with integrated concurrency control. The TCS exposes a simple interface allowing clients to submit transactions for *certification* via a `certify` request, which returns COMMIT or ABORT. A TCS is meant to be used in the context of transactional processing systems with optimistic concurrency control, where transactions are first executed optimistically, and the results (e.g., read and write sets) are submitted for certification to the TCS. In contrast to ACP, TCS does not impose any restrictions on the number of repeated `certify` invocations or their concurrency. It therefore lends itself naturally to formalising the interactions between transaction commit and concurrency control. To this end, TCS is parameterised by a *certification function*, which encapsulates the concurrency-control policy for the desired isolation level, such as serializability and snapshot isolation [1]. The correctness of TCS is then formulated by requiring that its certification decisions be consistent with the certification function.

We leverage TCS to develop a formal framework for constructing provably correct multi-shot transaction commit protocols with customisable isolation levels. The core ingredient of our framework is a new *multi-shot two-phase commit protocol* (§3). It formalises how the classical 2PC interacts with concurrency control in many practical transaction processing systems [32, 31, 34, 39] in a way that is parametric in the isolation level provided. The protocol also serves as a *template* for deriving more complex TCS implementations. We prove that the multi-shot 2PC protocol correctly implements a TCS with a given certification function, provided the concurrency-control policies used by each shard match this function.

We next propose a *crash fault-tolerant* TCS implementation and establish its correctness by proving that it simulates multi-shot 2PC (§4). A common approach to making 2PC fault-tolerant is to get every shard to simulate a reliable 2PC process using a replication protocol, such as Paxos [16, 18, 9, 14, 42]. Similarly to recent work [41, 26], our implementation optimises the time complexity of this scheme by weaving 2PC and Paxos together. In contrast to previous work, our protocol is both generic in the isolation level and rigorously proven correct. It can therefore serve as a reference solution for future distributed transaction commit implementations. Moreover, a variant of our protocol has a time complexity matching the lower bounds for consensus [24, 6] and non-blocking atomic commit [13].

The main idea for achieving such a low time complexity is to eliminate the Paxos consensus required in the vanilla fault-tolerant 2PC to persist the final decision on a transaction at a shard. Instead, the decision is propagated to the relevant shard replicas asynchronously.

This means that different shard replicas may receive the final decision on a transaction at different times, and thus their states may be inconsistent. To deal with this, in our protocol the votes are computed locally by a single shard *leader* based on the information available to it; other processes merely store the votes. Similarly to [29, 22], such a *passive replication* approach requires a careful design of recovery from leader failures. Another reduction in time complexity comes from the fact that our protocol avoids consistently replicating the 2PC coordinator: we allow any process to take over as a coordinator by accessing the current state of the computation at shards. The protocol ensures that all coordinators will reach the same decision on a transaction.

## 2 Transaction Certification Service

**Interface.** A *Transaction Certification Service (TCS)* accepts *transactions* from  $\mathcal{T}$  and produces *decisions* from  $\mathcal{D} = \{\text{ABORT}, \text{COMMIT}\}$ . Clients interact with the TCS using two types of *actions*: certification requests of the form `certify`( $t$ ), where  $t \in \mathcal{T}$ , and responses of the form `decide`( $t, d$ ), where  $d \in \mathcal{D}$ .

In this paper we focus on transactional processing systems using optimistic concurrency control. Hence, we assume that a transaction submitted to the TCS includes all the information produced by its optimistic execution. As an example, consider a transactional system managing *objects* in the set  $\text{Obj}$  with values in the set  $\text{Val}$ , where transactions can execute reads and writes on the objects. The objects are associated with a totally ordered set  $\text{Ver}$  of *versions* with a distinguished minimum version  $v_0$ . Then each transaction  $t$  submitted to the TCS may be associated with the following data:

- *Read set*  $R(t) \subseteq 2^{\text{Obj} \times \text{Ver}}$ : the set of objects with their versions that  $t$  read, which contains at most one version per object.
- *Write set* of  $W(t) \subseteq 2^{\text{Obj} \times \text{Val}}$ : the set of objects with their values that  $t$  wrote, which contains at most one value per object. We require that any object written has also been read:  $\forall (x, \_) \in W(t). (x, \_) \in R(t)$ .
- *Commit version*  $V_c(t) \in \text{Ver}$ : the version to be assigned to the writes of  $t$ . We require that this version be higher than any of the versions read:  $\forall (\_, v) \in R(t). V_c(t) > v$ .

**Certification functions.** A TCS is specified using a *certification function*  $f : 2^{\mathcal{T}} \times \mathcal{T} \rightarrow \mathcal{D}$ , which encapsulates the concurrency-control policy for the desired isolation level. The result  $f(T, t)$  is the decision for the transaction  $t$  given the set of the previously committed transactions  $T$ . We require  $f$  to be *distributive* in the following sense:

$$\forall T_1, T_2, t. f(T_1 \cup T_2, t) = f(T_1, t) \sqcap f(T_2, t), \quad (1)$$

where the  $\sqcap$  operator is defined as follows:  $\text{COMMIT} \sqcap \text{COMMIT} = \text{COMMIT}$  and  $d \sqcap \text{ABORT} = \text{ABORT}$  for any  $d$ . This requirement is justified by the fact that common definitions of  $f(T, t)$  check  $t$  for conflicts against each transaction in  $T$  separately.

For example, given the above domain of transactions, the following certification function encapsulates the classical concurrency-control policy for serializability [40]:  $f(T, t) = \text{COMMIT}$  iff none of the versions read by  $t$  have been overwritten by a transaction in  $T$ , i.e.,

$$\forall x, v. (x, v) \in R(t) \implies (\forall t' \in T. (x, \_) \in W(t') \implies V_c(t') \leq v). \quad (2)$$

A certification function for snapshot isolation (SI) [1] is similar, but restricts the certification check to the objects the transaction  $t$  writes:  $f(T, t) = \text{COMMIT}$  iff

$$\forall x, v. (x, v) \in R(t) \wedge (x, \_) \in W(t) \implies (\forall t' \in T. (x, \_) \in W(t') \implies V_c(t') \leq v). \quad (3)$$

It is easy to check that the certification functions (2) and (3) are distributive.

**Histories.** We represent TCS executions using *histories* – sequences of `certify` and `decide` actions such that every transaction appears at most once as a parameter to `certify`, and each `decide` action is a response to exactly one preceding `certify` action. For a history  $h$  we let  $\text{act}(h)$  be the set of actions in  $h$ . For actions  $a, a' \in \text{act}(h)$ , we write  $a \prec_h a'$  when  $a$  occurs before  $a'$  in  $h$ . A history  $h$  is *complete* if every `certify` action in it has a matching `decide` action. A complete history is *sequential* if it consists of pairs of `certify` and matching `decide` actions. A transaction  $t$  *commits* in a history  $h$  if  $h$  contains `decide`( $t$ , COMMIT). We denote by  $\text{committed}(h)$  the projection of  $h$  to actions corresponding to the transactions that are committed in  $h$ . For a complete history  $h$ , a *linearization*  $\ell$  of  $h$  [21] is a sequential history such that: (i)  $h$  and  $\ell$  contain the same actions; and (ii)

$$\forall t, t'. \text{decide}(t, \_) \prec_h \text{certify}(t') \implies \text{decide}(t, \_) \prec_\ell \text{certify}(t').$$

**TCS correctness.** A complete sequential history  $h$  is *legal* with respect to a certification function  $f$ , if its certification decisions are computed according to  $f$ :

$$\forall a = \text{decide}(t, d) \in \text{act}(h). d = f(\{t' \mid \text{decide}(t', \text{COMMIT}) \prec_h a\}, t).$$

A history  $h$  is *correct* with respect to  $f$  if  $h \mid \text{committed}(h)$  has a legal linearization. A TCS implementation is *correct* with respect to  $f$  if so are all its histories.

A correct TCS can be readily used in a transaction processing system. For example, consider the domain of transactions defined earlier. A typical system based on optimistic concurrency control will ensure that transactions submitted for certification read versions that already exist in the database. Formally, it will produce only histories  $h$  such that, for a transaction  $t$  submitted for certification in  $h$ , if  $(x, v) \in R(t)$ , then there exists a  $t'$  such that  $(x, v) \in W(t')$ , and  $h$  contains `decide`( $t'$ , COMMIT) before `certify`( $t$ ). It is easy to check that, if such a history  $h$  is correct with respect to the certification function (2), then it is also serializable. Hence, TCS correct with respect to certification function (2) can indeed be used to implement serializability.

### 3 Multi-Shot 2PC and Shard-Local Certification Functions

We now present a multi-shot version of the classical *two-phase commit* (2PC) protocol [15], parametric in the concurrency-control policy used by each shard. We then prove that the protocol implements a correct transaction certification service parameterised by a given certification function, provided per-shard concurrency control matches this function. Like 2PC, our protocol assumes reliable processes. In the next section, we establish the correctness of a protocol that allows crashes by proving that it simulates the behaviour of multi-shot 2PC.

**System model.** We consider an asynchronous message-passing system consisting of a set of processes  $\mathcal{P}$ . In this section we assume that processes are reliable and are connected by reliable FIFO channels. We assume a function  $\text{client} : \mathcal{T} \rightarrow \mathcal{P}$  determining the client process that issued a given transaction. The data managed by the system are partitioned into *shards* from a set  $\mathcal{S}$ . A function  $\text{shards} : \mathcal{T} \rightarrow 2^{\mathcal{S}}$  determines the shards that need to certify a given transaction, which are usually the shards storing the data the transaction accesses. Each shard  $s \in \mathcal{S}$  is managed by a process  $\text{proc}(s) \in \mathcal{P}$ . For simplicity, we assume that different processes manage different shards.

---

**Algorithm 1:** Multi-shot 2PC protocol at a process  $p_i$  managing a shard  $s_0$ .
 

---

```

1 next  $\leftarrow -1 \in \mathbb{Z}$ ;
2 txn[]  $\in \mathbb{N} \rightarrow \mathcal{T}$ ;
3 vote[]  $\in \mathbb{N} \rightarrow \{\text{COMMIT}, \text{ABORT}\}$ ;
4 dec[]  $\in \mathbb{N} \rightarrow \{\text{COMMIT}, \text{ABORT}\}$ ;
5 phase[]  $\leftarrow (\lambda k. \text{START}) \in \mathbb{N} \rightarrow \{\text{START}, \text{PREPARED}, \text{DECIDED}\}$ ;

6 function certify( $t$ )
7   send PREPARE( $t$ ) to proc(shards( $t$ ));

8 when received PREPARE( $t$ )
9   next  $\leftarrow$  next + 1;
10  txn[next]  $\leftarrow$   $t$ ;
11  vote[next]  $\leftarrow$   $f_{s_0}(\{\text{txn}[k] \mid k < \text{next} \wedge \text{phase}[k] = \text{DECIDED} \wedge \text{dec}[k] = \text{COMMIT}\}, t) \sqcap$ 
    $g_{s_0}(\{\text{txn}[k] \mid k < \text{next} \wedge \text{phase}[k] = \text{PREPARED} \wedge \text{vote}[k] = \text{COMMIT}\}, t)$ ;
12  phase[next]  $\leftarrow$  PREPARED;
13  send PREPARE_ACK( $s_0$ , next,  $t$ , vote[next]) to coord( $t$ );

14 when received PREPARE_ACK( $s$ ,  $pos_s$ ,  $t$ ,  $d_s$ ) for every  $s \in \text{shards}(t)$ 
15   send DECISION( $t$ ,  $\prod_{s \in \text{shards}(t)} d_s$ ) to client( $t$ );
16   forall  $s \in \text{shards}(t)$  do
17     send DECISION( $pos_s$ ,  $\prod_{s \in \text{shards}(t)} d_s$ ) to proc( $s$ )

18 when received DECISION( $k$ ,  $d$ )
19   dec[ $k$ ]  $\leftarrow$   $d$ ;
20   phase[ $k$ ]  $\leftarrow$  DECIDED;

21 non-deterministically for some  $k \in \mathbb{N}$ 
22   pre: phase[ $k$ ] = DECIDED;
23   phase[ $k$ ]  $\leftarrow$  PREPARED;

24 non-deterministically for some  $k \in \mathbb{N}$ 
25   pre: phase[ $k$ ]  $\neq$  START;
26   send PREPARE_ACK( $s_0$ ,  $k$ , txn[ $t$ ], vote[ $k$ ]) to coord( $t$ );

```

---

**Protocol: common case.** We give the pseudocode of the protocol in Algorithm 1 and illustrate its message flow in Figure 1a. Each handler in Algorithm 1 is executed atomically.

To certify a transaction  $t$ , a client sends it in a **PREPARE** message to the relevant shards (line 6)<sup>2</sup>. A process managing a shard arranges all transactions received into a total *certification order*, stored in an array `txn`; a `next` variable points to the last filled slot in the array. Upon receiving a transaction  $t$  (line 8), the process stores  $t$  in the next free slot of `txn`. The process also computes its *vote*, saying whether to **COMMIT** or **ABORT** the transaction, and stores it in an array `vote`. We explain the vote computation in the following; intuitively,

---

<sup>2</sup> In practice, the client only needs to send the data relevant to the corresponding shard. We omit this optimisation to simplify notation.

## 14:6 Multi-Shot Distributed Transaction Commit

the vote is determined by whether the transaction  $t$  conflicts with a previously received transaction. After the process managing a shard  $s$  receives  $t$ , we say that  $t$  is *prepared* at  $s$ . The process keeps track of transaction status in an array **phase**, whose entries initially store **START**, and are changed to **PREPARED** once the transaction is prepared. Having prepared the transaction  $t$ , the process sends a **PREPARE\_ACK** message with its position in the certification order and the vote to a *coordinator* of  $t$ . This is a process determined using a function  $\text{coord} : \mathcal{T} \rightarrow \mathcal{P}$  such that  $\forall t. \text{coord}(t) \in \text{proc}(\text{shards}(t))$ .

The coordinator of a transaction  $t$  acts once it receives a **PREPARE\_ACK** message for  $t$  from each of its shards  $s$ , which carries the vote  $d_s$  by  $s$  (line 14). The coordinator computes the final decision on  $t$  using the  $\sqcap$  operator (§2) and sends it in **DECISION** messages to the client and to all the relevant shards. When a process receives a decision for a transaction (line 18), it stores the decision in a **dec** array, and advances the transaction's phase to **DECIDED**.

**Vote computation.** A process managing a shard  $s$  computes votes as a conjunction of two *shard-local certification functions*  $f_s : 2^{\mathcal{T}} \times \mathcal{T} \rightarrow \mathcal{D}$  and  $g_s : 2^{\mathcal{T}} \times \mathcal{T} \rightarrow \mathcal{D}$ . Unlike the certification function of §2, the shard-local functions are meant to check for conflicts only on objects managed by  $s$ . They take as their first argument the sets of transactions already decided to commit at the shard, and respectively, those that are only prepared to commit (line 11). We require that the above functions be distributive, similarly to (1).

For example, consider the transaction model given in §2 and assume that the set of objects **Obj** is partitioned among shards:  $\text{Obj} = \bigsqcup_{s \in \mathcal{S}} \text{Obj}_s$ . Then the shard-local certification functions for serializability are defined as follows:  $f_s(T, t) = \text{COMMIT}$  iff

$$\forall x \in \text{Obj}_s. \forall v. (x, v) \in R(t) \implies (\forall t' \in T. (x, \_) \in W(t') \implies V_c(t') \leq v), \quad (4)$$

and  $g_s(T, t) = \text{COMMIT}$  iff

$$\begin{aligned} \forall x \in \text{Obj}_s. \forall v. ((x, \_) \in R(t) \implies (\forall t' \in T. (x, \_) \notin W(t'))) \wedge \\ ((x, \_) \in W(t) \implies (\forall t' \in T. (x, \_) \notin R(t'))) \end{aligned} \quad (5)$$

The function  $f_s$  certifies a transaction  $t$  against previously committed transactions  $T$  similarly to the certification function (2), but taking into account only the objects managed by the shard  $s$ . The function  $g_s$  certifies  $t$  against transactions  $T$  prepared to commit.

The first conjunct of (5) aborts a transaction  $t$  if it read an object written by a transaction  $t'$  prepared to commit. To motivate this condition, consider the following example. Assume that a shard managing an object  $x$  votes to commit a transaction  $t'$  that read a version  $v_1$  of  $x$  and wants to write a version  $v_2 > v_1$  of  $x$ . If the shard now receives another transaction  $t$  that read the version  $v_1$  of  $x$ , the shard has to abort  $t$ : if  $t'$  does commit in the end, allowing  $t$  to commit would violate serializability, since it would have read stale data. On the other hand, once the shard receives the abort decision on  $t'$ , it is free to commit  $t$ .

The second conjunct of (5) aborts a transaction  $t$  if it writes to an object read by a transaction  $t'$  prepared to commit. To motivate this, consider the following example, adapted from [37]. Assume transactions  $t_1$  and  $t_2$  both read a version  $v_1$  of  $x$  at shard  $s_1$  and a version  $v_2$  of  $y$  at shard  $s_2$ ;  $t_1$  wants to write a version  $v'_2 > v_2$  of  $y$ , and  $t_2$  wants to write a version  $v_2 > v_1$  of  $x$ . Assume further that  $s_1$  receives  $t_1$  first and votes to commit it, and  $s_2$  receives  $t_2$  first and votes to commit it as well. If  $s_1$  now receives  $t_2$  and  $s_2$  receives  $t_1$ , the second conjunct of (5) will force them to abort: if the shards let the transactions commit, the resulting execution would not be serializable, since one of the transactions must read the value written by the other.



A simple way of implementing (5) is, when preparing a transaction, to acquire read locks on its read set and write locks on its write set; the transaction is aborted if the locks cannot be acquired. The shard-local certification functions are a more abstract way of defining the behaviour of this and other implementations [32, 31, 34, 39, 37]. They can also be used to define weaker isolation levels than serializability. As an illustration, we can define shard-local certification functions for snapshot isolation as follows:  $f_s(T, t) = \text{COMMIT}$  iff

$$\forall x \in \text{Obj}_s. \forall v. (x, v) \in R(t) \wedge (x, \_) \in W(t) \implies (\forall t' \in T. (x, \_) \in W(t') \implies V_c(t') \leq v),$$

and  $g_s(T, t) = \text{COMMIT}$  iff

$$(x, \_) \in W(t) \implies (\forall t' \in T. (x, \_) \notin W(t')).$$

The function  $f_s$  restricts the global function (3) to the objects managed by the shard  $s$ . Since snapshot isolation allows reading stale data, the function  $g_s$  only checks for write conflicts.

For shard-local certification functions to correctly approximate a given global function  $f$ , we require the following relationships. For a set of transactions  $T \subseteq \mathcal{T}$ , we write  $T \mid s$  to denote the *projection* of  $T$  on shard  $s$ , i.e.,  $\{t \in T \mid s \in \text{shards}(t)\}$ . Then we require that

$$\forall t \in \mathcal{T}. \forall T \subseteq \mathcal{T}. f(T, t) = \text{COMMIT} \iff \forall s \in \text{shards}(t). f_s((T \mid s), t) = \text{COMMIT}. \quad (6)$$

In addition, for each shard  $s$ , the two functions  $f_s$  and  $g_s$  are required to be related to each other as follows:

$$\forall t. s \in \text{shards}(t) \implies (\forall T. g_s(T, t) = \text{COMMIT} \implies f_s(T, t) = \text{COMMIT}); \quad (7)$$

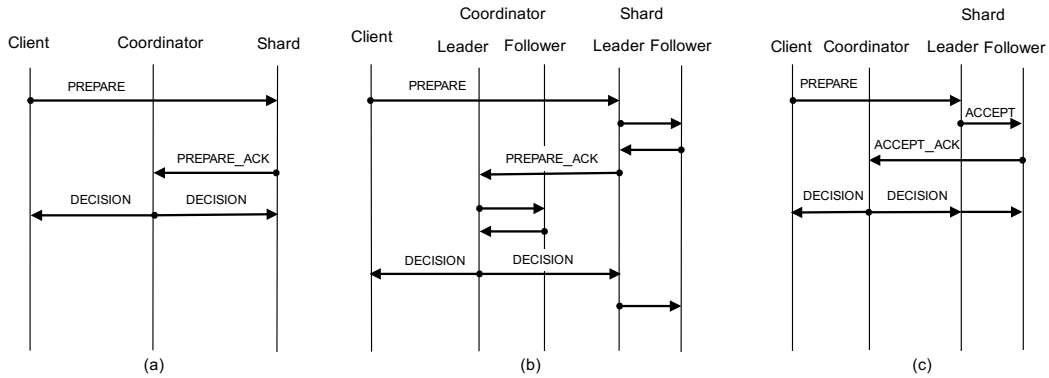
$$\forall t, t'. s \in \text{shards}(t) \cap \text{shards}(t') \implies (g_s(\{t\}, t') = \text{COMMIT} \implies f_s(\{t'\}, t) = \text{COMMIT}). \quad (8)$$

Property (7) requires the conflict check performed by  $g_s$  to be no weaker than the one performed by  $f_s$ . Property (8) requires a form of commutativity: if  $t'$  is allowed to commit after a still-pending transaction  $t$ , then  $t$  would be allowed to commit after  $t'$ . The above shard-local functions for serializability and snapshot isolation satisfy (6)-(8).

**Forgetting and recalling decisions.** The protocol in Algorithm 1 has two additional handlers at lines 21 and 24, executed non-deterministically. As we show in §4, these are required for the abstract protocol to capture the behaviour of optimised fault-tolerant TCS implementations. Because of process crashes, such implementations may temporarily lose the information about some final decisions, and later reconstruct it from the votes at the relevant shards. In the meantime, the absence of the decisions may affect some vote computations as we explained above. The handler at line 21 forgets the decision on a transaction (but not its vote). The handler at line 24 allows processes to resend the votes they know to the coordinator, which will then resend the final decisions (line 14). This allows a process that forgot a decision to reconstruct it from the votes stored at the relevant shards.

**Correctness.** The following theorem shows the correctness of multi-shot 2PC. In particular, it shows that the shard-local concurrency control given by  $f_s$  and  $g_s$  correctly implements the shard-agnostic concurrency control given by a global certification function  $f$ .

► **Theorem 1.** *A transaction certification service implemented using the multi-shot 2PC protocol in Algorithm 1 is correct with respect to a certification function  $f$ , provided shard-local certification functions  $f_s$  and  $g_s$  satisfy (6)-(8).*



■ **Figure 1** Message flow diagrams illustrating the behaviour of (a) multi-shot 2PC; (b) multi-shot 2PC with shards replicated using Paxos; (c) optimised protocol weaving together multi-shot 2PC and Paxos.

We give the proof in [7, §A]. Its main challenge is that, in multi-shot 2PC, certification orders at different shards may disagree on the order of concurrently certified transactions; however, a correct TCS has to certify transactions according to a single total order. We use the commutativity property (8) to show that per-shard certification orders arising in the protocol can be merged into the desired single total order.

#### 4 Fault-Tolerant Commit Protocol

**System model.** We now weaken the assumptions of the previous section by allowing processes to fail by crashing, i.e., permanently stopping execution. We still assume that processes are connected by reliable FIFO channels in the following sense: messages are delivered in the FIFO order, and messages between non-faulty processes are guaranteed to be eventually delivered. Each shard  $s$  is now managed by a group of  $2f + 1$  processes, out of which at most  $f$  can fail. We call a set of  $f + 1$  processes in this group a *quorum* for  $s$ . For a shard  $s$  we redefine  $\text{proc}(s)$  to be the set of processes managing this shard. For simplicity, we assume that the groups of processes managing different shards are disjoint.

**Vanilla protocol.** A straightforward way to implement a TCS in the above model is to use state-machine replication [36] to make a shard simulate a reliable process in multi-shot 2PC; this is usually based on a consensus protocol such as Paxos [27]. In this case, final decisions on transactions are never forgotten, and hence, the handlers at lines 21 and 24 are not simulated. Even though this approach is used by several systems [9, 14, 42], multiple researchers have observed that the resulting protocol requires an unnecessarily high number of message delays [41, 26, 28]. Namely, every action of multi-shot 2PC in Figure 1a requires an additional round trip to a quorum of processes in the same shard to persist its effect, resulting in the message-flow diagram in Figure 1b. Note that the coordinator actions have to be replicated as well, since multi-shot 2PC will block if the coordinator fails. The resulting protocol requires 7 message delays for a client to learn a decision on a transaction.

**Optimised protocol overview.** In Algorithms 2 and 3 we give a commit protocol that reduces the number of message delays by weaving together multi-shot 2PC across shards and a Paxos-like protocol within each shard. We omit details related to message retransmissions from the code. We illustrate the message flow of the protocol in Figure 1c and summarise the key invariants used in its proof of correctness in Figure 2.

A process maintains the same variables as in the multi-shot 2PC protocol (Algorithm 1) and a few additional ones. Every process in a shard is either the *leader* of the shard or a *follower*. If the leader fails, one of the followers takes over. A **status** variable records whether the process is a LEADER, a FOLLOWER or is in a special RECOVERING state used during leader changes. A period of time when a particular process acts as a leader is denoted using integer *ballots*. For a ballot  $b \geq 1$ , the process  $\text{leader}(b) = ((b - 1) \bmod (2f + 1))$  is the leader of the ballot. At any given time, a process participates in a single ballot, stored in a variable **ballot**. During leader changes we also use an additional ballot variable **cballot**.

Unlike the vanilla protocol illustrated in Figure 1b, our protocol does not perform consensus to persist the contents of a DECISION message in a shard. Instead, the final decision on a transaction is sent to the members of each relevant shard asynchronously. This means that different shard members may receive the decision on a transaction at different times. Since the final decision on a transaction affects vote computations on transactions following it in the certification order (§3), computing the vote on a later transaction at different shard members may yield different outcomes. To deal with this, in our protocol only the leader constructs the certification order and computes votes. Followers are passive: they merely copy the leader’s decisions. A final decision is taken into account in vote computations at a shard once it is received by the shard’s leader.

**Failure-free case.** To certify a transaction  $t$ , a client sends it in a PREPARE message to the relevant shards (line 10). A process  $p_i$  handles the message only when it is the leader of its shard  $s_0$  (line 12). We defer the description of the cases when another process  $p_j$  is resending the PREPARE message to  $p_i$  (line 13), and when  $p_i$  has already received  $t$  in the past (line 14).

Upon receiving PREPARE( $t$ ), the leader  $p_i$  first determines a process  $p$  that will serve as the coordinator of  $t$ . If the leader receives  $t$  for the first time (line 16), then, similarly to multi-shot 2PC, it appends  $t$  to the certification order and computes the vote based on the locally available information. The leader next performs an analogue of “phase 2” of Paxos, trying to convince its shard  $s_0$  to accept its proposal. To this end, it sends an ACCEPT message to  $s_0$  (including itself, for uniformity), which is analogous to the “2a” message of Paxos (line 21). The message carries the leader’s ballot, the transaction  $t$ , its position in the certification order, the vote and the identity of  $t$ ’s coordinator. The leader code ensures Invariant 1 in Figure 2: in a given ballot  $b$ , a unique transaction-vote pair can be assigned to a slot  $k$  in the certification order.

A process handles an ACCEPT message only if it participates in the corresponding ballot (line 23). If the process has not heard about  $t$  before, it stores the transaction and the vote and advances the transaction’s phase to PREPARED. It then sends an ACCEPT\_ACK message to the coordinator of  $t$ , analogous to the “2b” message of Paxos. This confirms that the process has accepted the transaction and the vote. The certification order at a follower is always a prefix of the certification order at the leader of the ballot the follower is in, as formalised by Invariant 2. This invariant is preserved when the follower receives ACCEPT messages due to the FIFO ordering of channels.

The coordinator of a transaction  $t$  acts once it receives a quorum of ACCEPT\_ACK messages for  $t$  from each of its shards  $s \in \text{shards}(t)$ , which carry the vote  $d_s$  by  $s$  (line 29). The coordinator computes the final decision on  $t$  and sends it in DECISION messages to the client and to each of the relevant shards. When a process receives a decision for a transaction (line 33), the process stores it and advances the transaction’s phase to DECIDED.

Once the final decision on a transaction is delivered to the leader of a shard, it is taken into account in future vote computations at this shard. Taking as an example the shard-local functions for serializability (4) and (5), if a transaction that wrote to an object  $x$  is finally

## 14:10 Multi-Shot Distributed Transaction Commit

1. If  $\text{ACCEPT}(b, k, t_1, d_1, \_)$  and  $\text{ACCEPT}(b, k, t_2, d_2, \_)$  messages are sent to the same shard, then  $t_1 = t_2$  and  $d_1 = d_2$ .
2. After a process receives and acknowledges  $\text{ACCEPT}(b, k, t, d, \_)$ , we have  $\text{txn} = \text{txn}|_k$  and  $\text{vote} = \text{vote}|_k$ , where  $\text{txn}$  and  $\text{vote}$  are the values of the arrays  $\text{txn}$  and  $\text{vote}$  at  $\text{leader}(b)$  when it sent the  $\text{ACCEPT}$  message.
3. Assume that a quorum of processes in  $s$  received  $\text{ACCEPT}(b, k, t, d, \_)$  and responded to it with  $\text{ACCEPT\_ACK}(s, b, k, t, d)$ , and at the time  $\text{leader}(b)$  sent  $\text{ACCEPT}(b, k, t, d, \_)$  it had  $\text{txn}|_k = \text{txn}$  and  $\text{vote}|_k = \text{vote}$ . Whenever at a process in  $s$  we have  $\text{status} \in \{\text{LEADER}, \text{FOLLOWER}\}$  and  $\text{ballot} = b' > b$ , we also have  $\text{txn}|_k = \text{txn}$  and  $\text{vote}|_k = \text{vote}$ .
4. If  $\text{ACCEPT}(b, k_1, t, \_, \_)$  and  $\text{ACCEPT}(b, k_2, t, \_, \_)$  messages are sent to the same shard, then  $k_1 = k_2$ .
5. At any process, all transactions in the  $\text{txn}$  array are distinct.
6. a. For any messages  $\text{DECISION}(\_, k, d_1)$  and  $\text{DECISION}(\_, k, d_2)$  sent to processes in the same shard, we have  $d_1 = d_2$ .  
b. For any messages  $\text{DECISION}(t, d_1)$  and  $\text{DECISION}(t, d_2)$  sent, we have  $d_1 = d_2$ .
7. a. Assume that a quorum of processes in  $s$  have sent  $\text{ACCEPT\_ACK}(s, b_1, k, t_1, d_1)$  and a quorum of processes in  $s$  have sent  $\text{ACCEPT\_ACK}(s, b_2, k, t_2, d_2)$ . Then  $t_1 = t_2$  and  $d_1 = d_2$ .  
b. Assume that a quorum of processes in  $s$  have sent  $\text{ACCEPT\_ACK}(s, b_1, k_1, t, d_1)$  and a quorum of processes in  $s$  have sent  $\text{ACCEPT\_ACK}(s, b_2, k_2, t, d_2)$ . Then  $k_1 = k_2$  and  $d_1 = d_2$ .

■ **Figure 2** Key invariants of the fault-tolerant protocol. We let  $\alpha|_k$  be the prefix of the sequence  $\alpha$  of length  $k$ .

decided to abort, then delivering this decision to the leader may allow another transaction writing to  $x$  to commit.

**Leader recovery.** We next explain how the protocol deals with failures, starting from a leader failure. The goal of the leader recovery procedure is to preserve Invariant 3: if in a ballot  $b$  a shard  $s$  accepted a vote  $d$  on a transaction  $t$  at the position  $k$  in the certification order, then this vote will persist in all future ballots; this is furthermore true for all votes the leader of ballot  $b$  took into account when computing  $d$ . The latter property is necessary for the shard to simulate the behaviour of a reliable process in multi-shot 2PC that maintains a unique certification order. To ensure this property, our recovery procedure includes an additional message from the new leader to the followers ensuring that, before a follower starts accepting proposals from the new leader, it has brought its state in sync with that of the leader (this is similar to [29, 22]). The ballot of the last leader a follower synchronised with in this way is recorded in  $\text{cballot}$ .

We now describe the recovery procedure in detail. When a process  $p_i$  suspects the leader of its shard of failure, it may try to become a new leader by executing the `recover` function (line 37). The process picks a ballot that it leads higher than  $\text{ballot}$  and sends it in a `NEW_LEADER` message to the shard members (including itself); this message is analogous to the “1a” message in Paxos. When a process receives a `NEW_LEADER(b)` message (line 39), it first checks that the proposed ballot  $b$  is higher than his. In this case, it sets its ballot to  $b$  and changes its status to `RECOVERING`, which causes it to stop processing `PREPARE`, `ACCEPT` and `DECISION` messages. It then replies to the new leader with a `NEW_LEADER_ACK` message containing all components of its state, analogous to the “1b” message of Paxos.

---

**Algorithm 2:** Fault-tolerant commit protocol at a process  $p_i$  in a shard  $s_0$ : failure-free case.

---

```

1 next  $\leftarrow -1 \in \mathbb{Z}$ ;
2 txn[]  $\in \mathbb{N} \rightarrow \mathcal{T}$ ;
3 vote[]  $\in \mathbb{N} \rightarrow \{\text{COMMIT}, \text{ABORT}\}$ ;
4 dec[]  $\in \mathbb{N} \rightarrow \{\text{COMMIT}, \text{ABORT}\}$ ;
5 phase[]  $\leftarrow (\lambda k. \text{START}) \in \mathbb{N} \rightarrow \{\text{START}, \text{PREPARED}, \text{DECIDED}\}$ ;
6 status  $\in \{\text{LEADER}, \text{FOLLOWER}, \text{RECOVERING}\}$ ;
7 ballot  $\leftarrow 0 \in \mathbb{N}$ ;
8 cballot  $\leftarrow 0 \in \mathbb{N}$ ;

9 function certify( $t$ )
10   send PREPARE( $t$ ) to proc(shards( $t$ ));

11 when received PREPARE( $t$ ) from  $p_j$  or a client
12   pre: status = LEADER;
13   if received from a process  $p_j$  then  $p \leftarrow p_j$  else  $p \leftarrow \text{coord}(t)$ ;
14   if  $\exists k. t = \text{txn}[k]$  then
15     send ACCEPT(ballot,  $k, t, \text{vote}[k], p$ ) to proc( $s_0$ )
16   else
17     next  $\leftarrow \text{next} + 1$ ;
18     txn[next]  $\leftarrow t$ ;
19     vote[next]  $\leftarrow f_{s_0}(\{\text{txn}[k] \mid k < \text{next} \wedge \text{phase}[k] = \text{DECIDED} \wedge \text{dec}[k] = \text{COMMIT}\}, t)$ 
20      $g_{s_0}(\{\text{txn}[k] \mid k < \text{next} \wedge \text{phase}[k] = \text{PREPARED} \wedge \text{vote}[k] = \text{COMMIT}\}, t)$ ;
21     phase[next]  $\leftarrow \text{PREPARED}$ ;
22     send ACCEPT(ballot, next,  $t, \text{vote}[\text{next}], p$ ) to  $s_0$ ;

23 when received ACCEPT( $b, k, t, d, p$ )
24   pre: status  $\in \{\text{LEADER}, \text{FOLLOWER}\} \wedge \text{ballot} = b$ ;
25   if phase[ $k$ ] = START then
26     txn[ $k$ ]  $\leftarrow t$ ;
27     vote[ $k$ ]  $\leftarrow d$ ;
28     phase[ $k$ ]  $\leftarrow \text{PREPARED}$ ;
29   send ACCEPT_ACK( $s_0, b, k, t, d$ ) to  $p$ ;

30 when for every  $s \in \text{shards}(t)$  received a quorum of ACCEPT_ACK( $s, b_g, pos_s, t, d_g$ )
31   send DECISION( $t, \prod_{s \in \text{shards}(t)} d_s$ ) to client( $t$ );
32   forall  $s \in \text{shards}(t)$  do
33     send DECISION( $b_s, pos_s, \prod_{s \in \text{shards}(t)} d_s$ ) to proc( $s$ )

34 when received DECISION( $b, k, d$ )
35   pre: status  $\in \{\text{LEADER}, \text{FOLLOWER}\} \wedge \text{ballot} \geq b \wedge \text{phase}[k] = \text{PREPARED}$ ;
36   dec[ $k$ ]  $\leftarrow d$ ;
37   phase[ $k$ ]  $\leftarrow \text{DECIDED}$ ;

```

---

---

**Algorithm 3:** Fault-tolerant commit protocol at a process  $p_i$  in a shard  $s_0$ : recovery.

---

```

37 function recover()
38   send NEW_LEADER(any ballot  $b$  such that  $b > \text{ballot} \wedge \text{leader}(\text{ballot}) = p_i$ ) to  $s_0$ ;

39 when received NEW_LEADER( $b$ ) from  $p_j$ 
40   pre:  $b > \text{ballot}$ ;
41   status  $\leftarrow$  RECOVERING;
42   ballot  $\leftarrow b$ ;
43   send NEW_LEADER_ACK(ballot, cballot, txn, vote, dec, phase) to  $p_j$ ;

44 when received {NEW_LEADER_ACK( $b$ , cballot $_j$ , txn $_j$ , vote $_j$ , dec $_j$ , phase $_j$ ) |  $p_j \in Q$ }
   from a quorum  $Q$  in  $s_0$ 
45   pre: status = RECOVERING  $\wedge$  ballot =  $b$ ;
46   var  $J \leftarrow$  the set of  $j$  with the maximal cballot $_j$ ;
47   forall  $k$  do
48     if  $\exists j \in J. \text{phase}_j[k] \geq \text{PREPARED}$  then
49       txn[ $k$ ]  $\leftarrow$  txn $_j[k]$ ;
50       vote[ $k$ ]  $\leftarrow$  vote $_j[k]$ ;
51       phase[ $k$ ]  $\leftarrow$  PREPARED;
52     if  $\exists j. \text{phase}_j[k] = \text{DECIDED}$  then
53       dec  $\leftarrow$  dec $_j[k]$ ;
54       phase[ $k$ ]  $\leftarrow$  DECIDED;
55   next  $\leftarrow$  min{ $k$  | phase[ $k$ ]  $\neq$  START};
56   cballot  $\leftarrow b$ ;
57   status  $\leftarrow$  LEADER;
58   send NEW_STATE( $b$ , txn, vote, dec, phase) to proc( $s_0$ )  $\setminus$  { $p_i$ };

59 when received NEW_STATE( $b$ , txn, vote, dec, phase) from  $p_j$ 
60   pre:  $b \geq \text{ballot}$ ;
61   status  $\leftarrow$  FOLLOWER;
62   cballot  $\leftarrow b$ ;
63   txn  $\leftarrow$  txn;
64   vote  $\leftarrow$  vote;
65   dec  $\leftarrow$  dec;
66   phase  $\leftarrow$  phase;

67 function retry( $k$ )
68   pre: phase[ $k$ ] = PREPARED;
69   send PREPARE(txn[ $k$ ]) to proc(shards(txn[ $k$ ]));

```

---

The new leader waits until it receives `NEW_LEADER_ACK` messages from a quorum of shard members (line 44). Based on the states reported by the processes, it computes a new state from which to start certifying transactions. Like in Paxos, the leader focusses on the states of processes that reported the maximal cballot (line 46): if the  $k$ -th transaction is `PREPARED` at such a process, then the leader marks it as accepted and copies the vote; furthermore, if the transaction is `DECIDED` at some process (with any ballot number), then the leader marks it as decided and copies the final decision. Given Invariant 2, we can show that the resulting

certification order does not have holes: if a transaction is PREPARED or DECIDED, then so are the previous transactions in the certification order.

The leader sets next to the length of the merged sequence of transactions, `cballot` to the new ballot and `status` to LEADER, which allows it to start processing new transactions (lines 55-57). It then sends a NEW\_STATE message to other shard members, containing the new state (line 58). Upon receiving this message (line 59), a process overwrites its state with the one provided, changes its status to FOLLOWER, and sets `cballot` to  $b$ , thereby recording the fact that it has synchronised with the leader of  $b$ . Note that the process will not accept transactions from the new leader until it receives the NEW\_STATE message. This ensures that Invariant 2 is preserved when the process receives the first ACCEPT message in the new ballot.

**Coordinator recovery.** If a process that accepted a transaction  $t$  does not receive the final decision on it, this may be because the coordinator of  $t$  has failed. In this case the process may decide to become a new coordinator by executing the `retry` function (line 67). For this the process just re-sends the PREPARE( $t$ ) message to the shards of  $t$ . A leader handles the PREPARE( $t$ ) message received from another process  $p_j$  similarly to one received from a client. If it has already certified the transaction  $t$ , it re-sends the corresponding ACCEPT message to the shard members, asking them to reply to  $p_j$  (line 14). Otherwise, it handles  $t$  as before. In the end, a quorum of processes in each shard will reply to the new coordinator (line 28), which will then broadcast the final decision (lines 30-31). Note that the check at line 14 ensures Invariants 4 and 5: in a given ballot  $b$ , a transaction  $t$  can only be assigned to a single slot in the certification order, and all transactions in the `txn` array are distinct.

Our protocol allows any number of processes to become coordinators of a transaction at the same time: unlike in the vanilla protocol of Figure 1b, coordinators are not consistently replicated. Nevertheless, the protocol ensures that they will all reach the same decision, even in case of leader changes. We formalise this in Invariant 6: part (a) ensures an agreement on the decision on the  $k$ -th transaction in the certification order at a given shard; part (b) ensures a system-wide agreement on the decision on a given transaction  $t$ . The latter part establishes that the fault-tolerant protocol computes a unique decision on each transaction.

By the structure of the handler at line 29, Invariant 6 follows from Invariant 7, since, if a coordinator has computed the final decision on a transaction, then a quorum of processes in each relevant shard has accepted a corresponding vote. We next prove Invariant 7.

**Proof of Invariant 7.** (a) Let us assume that quorums of processes in  $s$  have sent `ACCEPT_ACK`( $s, b_1, k, t_1, d_1$ ) and `ACCEPT_ACK`( $s, b_2, k, t_2, d_2$ ). Then `ACCEPT`( $b_1, k, t_1, d_1, \_$ ) and `ACCEPT`( $b_2, k, t_2, d_2, \_$ ) have been sent to  $s$ . Assume without loss of generality that  $b_1 \leq b_2$ . If  $b_1 = b_2$ , then by Invariant 1 we must have  $t_1 = t_2$  and  $d_1 = d_2$ . Assume now that  $b_1 < b_2$ . By Invariant 3, when `leader`( $b_2$ ) sends the ACCEPT message, it has `txn`[ $k$ ] =  $t_1$ . But then due to the check at line 14, we again must have  $t_1 = t_2$  and  $d_1 = d_2$ .

(b) Assume that quorums of processes in  $s$  have sent `ACCEPT_ACK`( $s, b_1, k_1, t, d_1$ ) and `ACCEPT_ACK`( $s, b_2, k_2, t, d_2$ ). Then `ACCEPT`( $b_1, k_1, t, d_1, \_$ ) and `ACCEPT`( $b_2, k_2, t, d_2, \_$ ) have been sent to  $s$ . Without loss of generality, we can assume  $b_1 \leq b_2$ . We first show that  $k_1 = k_2$ . If  $b_1 = b_2$ , then we must have  $k_1 = k_2$  by Invariant 4. Assume now that  $b_1 < b_2$ . By Invariant 3, when `leader`( $b_2$ ) sends the ACCEPT message, it has `txn`[ $k_1$ ] =  $t$ . But then due to the check at line 14 and Invariant 5, we again must have  $k_1 = k_2$ . Hence,  $k_1 = k_2$ . But then by Invariant 7a we must also have  $d_1 = d_2$ . ◀

**Protocol correctness.** We only establish the safety of the protocol (in the sense of the correctness condition in §2) and leave guaranteeing liveness to standard means, such as assuming either an oracle that is eventually able to elect a consistent leader in every shard [5], or that the system eventually behaves synchronously for sufficiently long [12].

► **Theorem 2.** *The fault-tolerant commit protocol in Algorithms 2–3 simulates the multi-shot 2PC protocol in Algorithm 1.*

We give the proof in [7, §B]. Its main idea is to show that, in an execution of the fault-tolerant protocol, each shard produces a single certification order on transactions from which votes and final decisions are computed. These certification orders determine the desired execution of the multi-shot 2PC protocol. We prove the existence of a single per-shard certification order using Invariant 3, showing that certification orders and votes used to compute decisions persist across leader changes. However, this property does not hold of final decisions, and it is this feature that necessitates adding transitions for forgetting and recalling final decisions to the protocol in Algorithm 1 (lines 21 and 24).

For example, assume that the leader of a ballot  $b$  at a shard  $s$  receives the decision ABORT on a transaction  $t$ . The leader will then take this decision into account in its vote computations, e.g., allowing transactions conflicting with  $t$  to commit. However, if the leader fails, a new leader may not find out about the final decision on  $t$  if this decision has not yet reached other shard members. This leader will not be able to take the decision into account in its vote computations until it reconstructs the decision from the votes at the relevant shards (line 67). Forgetting and recalling the final decisions in the multi-shot 2PC protocol captures how such scenarios affect vote computations.

**Optimisations.** Our protocol allows the client and the relevant servers to learn the decision on a transaction in four message delays, including communication with the client (Figure 1c). As in standard Paxos, this can be further reduced to three message delays at the expense of increasing the number of messages sent by eliminating the coordinator: processes can send their ACCEPT\_ACK messages for a transaction directly to all processes in the relevant shards and to the client. Each process can then compute the final decision independently. The resulting time complexity matches the lower bounds for consensus [24, 6] and non-blocking atomic commit [13].

In practice, the computation of a shard-local function for  $s$  depends only on the objects managed by  $s$ : e.g.,  $\text{Obj}_s$  for (4) and (5). Hence, once a process at a shard  $s$  receives the final decision on a transaction  $t$ , it may discard the data of  $t$  irrelevant to  $s$ . Note that the same cannot be done when  $t$  is only prepared, since the complete information about it may be needed to recover from coordinator failure (line 67).

## 5 Related Work

The existing work on the Atomic Commit Problem (ACP) treats it as a one-shot problem with the votes being provided as the problem inputs. The classic ACP solution is the Two-Phase Commit (2PC) protocol [15], which blocks in the event of the coordinator failure. The *non-blocking* variant of ACP known as Non-Blocking Atomic Commit (NBAC) [38] has been extensively studied in both the distributed computing and database communities [38, 13, 33, 20, 23, 17, 18, 16]. The Three-Phase Commit (3PC) family of protocols [38, 3, 13, 2, 23] solve NBAC by augmenting 2PC with an extra message exchange round in the failure-free case. Paxos Commit [16] and Guerraoui et al. [18] avoid extra message delays by instead replicating



the 2PC participants through consensus instances. While our fault-tolerant protocol builds upon similar ideas to optimise the number of failure-free message delays, it nonetheless solves a more general problem (TCS) by requiring the output decisions to be compatible with the given isolation level.

Recently, Guerraoui and Wang [19] have systematically studied the failure-free complexity of NBAC (in terms of both message delays and number of messages) for various combinations of the correctness properties and failure models. The complexity of certifying a transaction in the failure-free runs of our crash fault-tolerant TCS implementation (provided the coordinator is replaced with all-to-all communication) matches the tight bounds for the most robust version of NBAC considered in [19], which suggests it is optimal. A comprehensive study of the TCS complexity in the absence of failures is the subject of future work.

Our multi-shot 2PC protocol is inspired by how 2PC is used in a number of systems [32, 31, 34, 39, 37, 9, 14]. Unlike prior works, we formalise how 2PC interacts with concurrency control in such systems in a way that is parametric in the isolation level provided and give conditions for its correctness, i.e., (6)-(8). A number of systems based on deferred update replication [30] used non-fault-tolerant 2PC for transaction commit [32, 31, 34, 39]. Our formalisation of the TCS problem should allow making them fault-tolerant using protocols of the kind we presented in §4.

Multiple researchers have observed that implementing transaction commit by layering 2PC on top of Paxos is suboptimal and proposed possible solutions [41, 26, 28, 37, 11]. In comparison to our work, they did not formulate a stand-alone certification problem, but integrated certification with the overall transaction processing protocol for a particular isolation level and corresponding optimisations.

In more detail, Kraska et al. [26] and Zhang et al. [41] presented sharded transaction processing systems, respectively called MDCC and TAPIR, that aim to minimise the latency of transaction commit in a geo-distributed setting. The protocols used are leaderless: to compute the vote, the coordinator of a transaction contacts processes in each relevant shard directly; if there is a disagreement between the votes computed by different processes, additional message exchanges are needed to resolve it. This makes the worst-case failure-free time complexity of the protocols higher than that of our fault-tolerant protocol. The protocols were formulated for particular isolation levels (a variant of Read Committed in MDCC and serializability in TAPIR). Both MDCC and TAPIR are significantly more complex than our fault-tolerant commit protocol and lack rigorous proofs of correctness.

Sciascia et al. proposed Scalable Deferred Update Replication [37] for implementing serializable transactions in sharded systems. Like the vanilla protocol in §4, their protocol keeps shards consistent using black-box consensus. It avoids executing consensus to persist a final decision by just not taking final decisions into account in vote computations. This solution, specific to their conflict check for serializability, is suboptimal: if a prepared transaction  $t$  aborts, it will still cause conflicting transactions to abort until their read timestamp goes above the write timestamp of  $t$ .

Dragojević et al. presented a FaRM transactional processing system based on RDMA [11]. Like in our fault-tolerant protocol, in the FaRM atomic commit protocol only shard leaders compute certification votes. However, recovery in FaRM is simplified by the use of leases and an external reconfiguration engine.

Mahmoud et al. proposed Replicated Commit [28], which reduces the latency of transaction commit by layering Paxos on top of 2PC, instead of the other way round. This approach relies on 2PC deciding ABORT only in case of failures, but not because of concurrency control. This requires integrating the transaction commit protocol with two-phase locking and does not allow using it with optimistic concurrency control.

Schiper et al. proposed an alternative approach to implementing deferred update replication in sharded systems [35]. This distributes transactions to shards for certification using genuine atomic multicast [10], which avoids the need for a separate fault-tolerant commit protocol. However, atomic multicast is more expensive than consensus: the best known implementation requires 4 message delays to deliver a message, in addition to a varying convoy effect among different transactions [8]. The resulting overall latency of certification is 5 message delays plus the convoy effect.

Our fault-tolerant protocol follows the primary/backup state machine replication approach in imposing the leader order on transactions certified within each shard. This is inspired by the design of some total order broadcast protocols, such as Zab [22] and Viewstamped Replication [29]. Kokocinski et al. [25] have previously explored the idea of delegating the certification decision to a single leader in the context of deferred update replication. However, they only considered a non-sharded setting, and did not provide full implementation details and a correctness proof. In particular, it is unclear how correctness is maintained under leader changes in their protocol.

## 6 Conclusion

In this paper we have made the first step towards building a theory of distributed transaction commit in modern transaction processing systems, which captures interactions between atomic commit and concurrency control. We proposed a new problem of transaction certification service and an abstract protocol solving it among reliable processes. From this, we have systematically derived a provably correct optimised fault-tolerant protocol.

For conciseness, in this paper we focussed on transaction processing systems using optimistic concurrency control. We hope that, in the future, our framework can be generalised to systems that employ pessimistic concurrency control or a mixture of the two. The simple and leader-driven nature of our optimised protocol should also allow porting it to the Byzantine fault-tolerant setting by integrating ideas from consensus protocols such as PBFT [4].

---

## References

- 1 Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O’Neil, and Patrick O’Neil. A critique of ANSI SQL isolation levels. In *Conference on Management of Data (SIGMOD)*, 1995.
- 2 Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., 1986.
- 3 Andrea J. Borr. Transaction monitoring in ENCOMPASS: reliable distributed transaction processing. In *International Conference on Very Large Data Bases (VLDB)*, 1981.
- 4 Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 1999.
- 5 Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4), 1996.
- 6 Bernadette Charron-Bost and André Schiper. Uniform consensus is harder than consensus. *J. Algorithms*, 51(1), 2004.
- 7 Gregory Chockler and Alexey Gotsman. Multi-shot distributed transaction commit (extended version). *arXiv CoRR*, 1808.00688, 2018. Available from <http://arxiv.org/abs/1808.00688>.
- 8 Paulo R. Coelho, Nicolas Schiper, and Fernando Pedone. Fast atomic multicast. In *Conference on Dependable Systems and Networks (DSN)*, 2017.

- 9 James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.
- 10 Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4), 2004.
- 11 Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *Symposium on Operating Systems Principles (SOSP)*, 2015.
- 12 Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2), 1988.
- 13 Cynthia Dwork and Dale Skeen. The inherent cost of nonblocking commitment. In *Symposium on Principles of Distributed Computing (PODC)*, 1983.
- 14 Lisa Glendenning, Ivan Beschastnikh, Arvind Krishnamurthy, and Thomas Anderson. Scalable consistency in Scatter. In *Symposium on Operating Systems Principles (SOSP)*, 2011.
- 15 Jim Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, 1978.
- 16 Jim Gray and Leslie Lamport. Consensus on transaction commit. *ACM Trans. Database Syst.*, 31(1), 2006.
- 17 Rachid Guerraoui. Revisiting the relationship between non-blocking atomic commitment and consensus. In *Workshop on Distributed Algorithms (WDAG)*, 1995.
- 18 Rachid Guerraoui, Mikel Larrea, and André Schiper. Reducing the cost for non-blocking in atomic commitment. In *International Conference on Distributed Computing Systems (ICDCS)*, 1996.
- 19 Rachid Guerraoui and Jingjing Wang. How fast can a distributed transaction commit? In *Symposium on Principles of Database Systems (PODS)*, 2017.
- 20 V. Hadzilacos. On the relationship between the atomic commitment and consensus problems. In *Asilomar Workshop on Fault-Tolerant Distributed Computing*, 1990.
- 21 Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3), 1990.
- 22 Flavio Paiva Junqueira, Benjamin C. Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *Conference on Dependable Systems and Networks (DSN)*, 2011.
- 23 Idit Keidar and Danny Dolev. Increasing the resilience of atomic commit at no additional cost. In *Symposium on Principles of Database Systems (PODS)*, 1995.
- 24 Idit Keidar and Sergio Rajsbaum. A simple proof of the uniform consensus synchronous lower bound. *Inf. Process. Lett.*, 85(1), 2003.
- 25 Maciej Kokocinski, Tadeusz Kobus, and Pawel T. Wojciechowski. Make the leader work: Executive deferred update replication. In *Symposium on Reliable Distributed Systems (SRDS)*, 2014.
- 26 Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. MDCC: Multi-data center consistency. In *European Conference on Computer Systems (EuroSys)*, 2013.
- 27 Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2), 1998.

- 28 Hatem Mahmoud, Faisal Nawab, Alexander Pucher, Divyakant Agrawal, and Amr El Abadi. Low-latency multi-datacenter databases using replicated commit. *Proc. VLDB Endow.*, 6(9), 2013.
- 29 Brian M. Oki and Barbara H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Symposium on Principles of Distributed Computing (PODC)*, 1988.
- 30 Fernando Pedone, Rachid Guerraoui, and André Schiper. The database state machine approach. *Distributed and Parallel Databases*, 14(1), 2003.
- 31 Sebastiano Peluso, Paolo Romano, and Francesco Quaglia. Score: A scalable one-copy serializable partial replication protocol. In *International Middleware Conference (Middleware)*, 2012.
- 32 Sebastiano Peluso, Pedro Ruivo, Paolo Romano, Francesco Quaglia, and Luís E. T. Rodrigues. GMU: genuine multiversion update-serializable partial data replication. *IEEE Trans. Parallel Distrib. Syst.*, 27(10), 2016.
- 33 K. V. S. Ramarao. Complexity of distributed commit protocols. *Acta Informatica*, 26(6), 1989.
- 34 Masoud Saeida Ardekani, Pierre Sutra, and Marc Shapiro. G-DUR: A middleware for assembling, analyzing, and improving transactional protocols. In *International Middleware Conference (Middleware)*, 2014.
- 35 Nicolas Schiper, Pierre Sutra, and Fernando Pedone. P-store: Genuine partial replication in wide area networks. In *Symposium on Reliable Distributed Systems (SRDS)*, 2010.
- 36 Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4), 1990.
- 37 Daniele Sciascia, Fernando Pedone, and Flavio Junqueira. Scalable deferred update replication. In *Conference on Dependable Systems and Networks (DSN)*, 2012.
- 38 Dale Skeen. Nonblocking commit protocols. In *Conference on Management of Data (SIGMOD)*, 1981.
- 39 Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Symposium on Operating Systems Principles (SOSP)*, 2011.
- 40 Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann Publishers Inc., 2001.
- 41 Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. Building consistent transactions with inconsistent replication. In *Symposium on Operating Systems Principles (SOSP)*, 2015.
- 42 Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. When is operation ordering required in replicated transactional storage? *IEEE Data Eng. Bull.*, 39(1), 2016.

# Deterministic Blind Radio Networks

**Artur Czumaj**

University of Warwick, Coventry, UK  
A.Czumaj@warwick.ac.uk

**Peter Davies**

University of Warwick, Coventry, UK  
P.W.Davies@warwick.ac.uk

---

## Abstract

Ad-hoc radio networks and multiple access channels are classical and well-studied models of distributed systems, with a large body of literature on deterministic algorithms for fundamental communications primitives such as broadcasting and wake-up. However, almost all of these algorithms assume knowledge of the number of participating nodes and the range of possible IDs, and often make the further assumption that the latter is linear in the former. These are very strong assumptions for models which were designed to capture networks of weak devices organized in an ad-hoc manner. It was believed that without this knowledge, deterministic algorithms must necessarily be much less efficient.

In this paper we address this fundamental question and show that this is not the case. We present *deterministic* algorithms for *blind* networks (in which nodes know only their own IDs), which match or nearly match the running times of the fastest algorithms which assume network knowledge (and even surpass the previous fastest algorithms which assume parameter knowledge but not small labels).

Specifically, in multiple access channels with  $k$  participating nodes and IDs up to  $L$ , we give a wake-up algorithm requiring  $O(\frac{k \log L \log k}{\log \log k})$  time, improving dramatically over the  $O(L^3 \log^3 L)$  time algorithm of De Marco et al. (2007), and a broadcasting algorithm requiring  $O(k \log L \log \log k)$  time, improving over the  $O(L)$  time algorithm of Gaşieniec et al. (2001) in most circumstances. Furthermore, we show how these same algorithms apply directly to multi-hop radio networks, achieving even larger running time improvements.

**2012 ACM Subject Classification** Theory of computation → Distributed algorithms, Networks → Network algorithms

**Keywords and phrases** Broadcasting, Deterministic Algorithms, Radio Networks

**Digital Object Identifier** 10.4230/LIPIcs.DISC.2018.15

**Funding** Research partially supported by the Centre for Discrete Mathematics and its Applications (DIMAP), by EPSRC award EP/D063191/1, and by EPSRC award EP/N011163/1.

## 1 Introduction

In this paper we address the fundamental question in distributed computing of whether basic communication primitives can be efficiently performed in networks in which the participating nodes have no knowledge about the network structure. Our focus is on *deterministic* algorithms.



© Artur Czumaj and Peter Davies;  
licensed under Creative Commons License CC-BY  
32nd International Symposium on Distributed Computing (DISC 2018).  
Editors: Ulrich Schmid and Josef Widder; Article No. 15; pp. 15:1–15:17  
Leibniz International Proceedings in Informatics



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

## 1.1 Models and problems

We consider the two classical, and related, models of distributed communication: *multiple access channels* (cf. [19, 28]) and *ad-hoc multi-hop radio networks* (cf. [2, 8, 14, 27]).

### 1.1.1 Multiple access channels

A set of  $k$  nodes, with unique identifiers (IDs) from  $\{1, \dots, L\}$ , share a communication channel. Time is divided into discrete steps, and in every step each node chooses to either transmit a message to the channel or listen for messages. A transmission is only successful if exactly one node chooses to transmit in a given time-step; otherwise all nodes hear silence.

### 1.1.2 Ad-hoc multi-hop radio networks

The network is modeled by a *directed* graph  $\mathfrak{N} = (V, E)$ , with  $|V| = n$ , where nodes correspond to transmitter-receiver stations. The nodes have unique identifiers from  $\{1, \dots, L\}$ . A directed edge  $(v, u) \in E$  means that node  $v$  can send a message directly to node  $u$ . To make propagation of information feasible, we assume that every node in  $V$  is reachable in  $\mathfrak{N}$  from any other. Time is divided into discrete steps, and in every step each node chooses to either transmit a message to all neighbors or listen for messages. A listening node only hears a transmission if exactly one neighbor transmitted; otherwise it hears silence.

It can be seen that multiple access channels are equivalent to *single-hop radio networks* (that is, radio networks in which the underlying graph is a clique).

### 1.1.3 Node knowledge

We study *blind* versions of these models, by which we mean that the minimum possible assumptions about node knowledge are made (and this is where our work differs most significantly from previous work): we assume nodes do not know the parameters  $k$ ,  $L$ , or  $n$ , or any upper bounds thereof. In accordance with the standard model of ad-hoc radio networks (for more elaborate discussion about the model, see, e.g., [1, 2, 6, 9, 10, 16, 21, 23, 27]), we also make the assumption that a node does not have any prior knowledge about the topology of the network, its in-degree and out-degree, or the set of its neighbors. In our setting, *the only prior knowledge nodes have is their own unique ID*.

### 1.1.4 Tasks

In both models we consider the fundamental communication tasks of *broadcasting* (see, e.g., the survey [27] and the references therein) and *wake-up* (cf. [3, 8, 15]).

In the task of *wake-up*, nodes begin in a dormant state, and some non-empty subset of nodes spontaneously ‘wake up’ at arbitrary (adversarially chosen) time-steps. Nodes are also woken up if they receive messages. Nodes cannot participate (by transmitting) until they are woken up, and our goal is to ensure that eventually all nodes are awake. We assume nodes have access only to a *local clock*: they can count the number of time-steps since they woke up, but there is no global awareness of an absolute time-step number.

The task of *broadcasting* is usually described as follows: one node begins with a message, and it must inform all other nodes of this message via transmissions. However, to enable our results to transfer from multiple access channels (single-hop radio networks) to multi-hop radio networks, we will instead use broadcasting to refer to a more generalized task. Our broadcasting task will be defined similarly to wake-up, with the only difference being that

nodes have access to a *global clock*, informing them of the absolute time-step number. (In multiple access channels, this task is usually also referred to as wake-up, specifying global clock access, but here we will call it broadcasting to better differentiate.)

Notice that the standard broadcasting task in radio networks is a special case of this task, in which only one node spontaneously wakes up. A global clock can be simulated by appending the current global time-step to each transmitted message (and since all message chains originate from the same source node, these time-steps will agree).

For both tasks, we wish to minimize the number of time-steps that elapse between the first node waking up, and all nodes being woken. We are not concerned with the computation performed by nodes within time-steps.

## 1.2 Related work

As fundamental communications primitives, the tasks of designing efficient deterministic algorithms for *broadcasting* and *wake-up* have been extensively studied for various network models for many decades.

### 1.2.1 Wake-up

The wake-up problem (with only local clocks) has been studied in both multiple access channels and multi-hop radio networks (often separately, though the results usually transfer directly from one to the other). It has been commonly assumed in the literature that network parameters are known, and that IDs are small ( $L = n^{O(1)}$ ).

The first sub-quadratic deterministic wake-up protocol for radio networks was given in by Chrobak et al. [8], who introduced the concept of *radio synchronizers* to abstract the essence of the problem. They give an  $O(n^{5/3} \log n)$ -time protocol for the wake-up problem. Since then, there have been several improvements in running time, making use of the radio synchronizer machinery: firstly to  $O(n^{3/2} \log n)$  [4], and then to  $O(n \log^2 n)$  [3]. The current fastest protocol is  $O(\frac{n \log^2 n}{\log \log n})$  [13]. However, without the assumption of small labels, all of these running times are increased. The algorithm of [13] as analyzed would give  $O(\frac{n \log L \log(n \log L)}{\log \log(n \log L)})$  time, and similar time with  $k$  replacing  $n$  in multiple access channels. All of these algorithms, like those we present here, are non-explicit.

There has been some work on wake-up in multiple access channels without knowledge of network parameters: firstly an  $O(L^4 \log^5 L)$  algorithm [15], and then an improvement to  $O(L^3 \log^3 L)$  [26]. It was believed that this algorithms in this setting were necessarily much slower than those for when parameters were known; for example, [26] states “a crucial assumption is whether the processors using the shared channel are aware of the total number  $n$  of processors sharing the channel, or some polynomially related upper bound to such number. When such number  $n$  is known, much faster algorithms are possible.”

There are no direct results for wake-up in radio networks with unknown parameters, but the algorithm of [26] can be applied to give  $O(nL^3 \log^3 L)$  time.

We note that randomized algorithms for wake-up have also been studied, both with and without parameter knowledge; see [15, 19].

### 1.2.2 Broadcasting

Broadcasting is possibly the most studied problem in radio networks, and has a wealth of literature in various settings. For the model studied in this paper, *directed* radio networks with *unknown structure* and *without collision detection*, the first sub-quadratic *deterministic*



broadcasting algorithm was proposed by Chlebus et al. [6], who gave an  $O(n^{11/6})$ -time broadcasting algorithm. After several small improvements (cf. [7, 25]), Chrobak et al. [9] designed an almost optimal algorithm that ns the task in  $O(n \log^2 n)$  time, the first to be only a poly-logarithmic factor away from linear dependency. Kowalski and Pelc [21] improved this bound to obtain an algorithm of complexity  $O(n \log n \log D)$  and Czumaj and Rytter [14] gave a broadcasting algorithm running in time  $O(n \log^2 D)$ . Here  $D$  is the eccentricity of the network, i.e., the distance between the furthest pair of nodes. De Marco [24] designed an algorithm that completes broadcasting in  $O(n \log n \log \log n)$  time steps, beating [14] for general graphs. Finally, the  $O(n \log D \log \log D)$  algorithm of [13] came within a log-logarithmic factor of the  $\Omega(n \log D)$  lower bound [10]. Again, however, these results generally assume *small node labels* ( $L = O(n)$ , though some of the earlier results only require  $L = O(n^c)$  for some constant  $c$ ), and their running time results do not hold otherwise. The situation where node labels can be large is less well-studied, though it is easy to see that the algorithm of [9] still works, requiring  $O(n \log^2 L)$  time. In multiple access channels, a  $O(k \log \frac{L}{k})$  time algorithm exists [10]. Again, all of these algorithms are, like those presented here, non-explicit.

All of these results also *intrinsically require parameter knowledge*. Without knowledge of  $n$ ,  $L$ ,  $k$ , or  $D$ , the fastest algorithm known is the  $O(L)$  time algorithm of [15] for multiple access channels. This algorithm is explicit, but has the strong added restriction that the first node wakes up at global time-step 0. It also does not transfer to multi-hop radio networks, so the best running time therein is the  $O(DL^3 \log^3 L)$  given by the algorithm of [26]. Concurrently with this work, randomized algorithms for broadcasting without parameter knowledge are presented in [12], achieving a nearly optimal running time of  $O(D \log \frac{n}{D} \log^2 \log \frac{n}{D} + \log^2 n)$  in the model we study here (that is, the model without collision detection).

Broadcasting, as a fundamental communication primitive, has been also studied in various related models, including undirected networks, randomized broadcasting protocols, models with collision detection, and models in which the entire network structure is known. For example, if the underlying network is undirected, then an  $O(n \log D)$ -time algorithm due to Kowalski [20] exists. If spontaneous transmissions are allowed and a global clock available, then deterministic broadcast can be performed in  $O(L)$  time in undirected networks [6]. Randomized broadcasting has been also extensively studied, and in a seminal paper, Bar-Yehuda et al. [2] designed an almost optimal broadcasting algorithm achieving the running time of  $O((D + \log n) \cdot \log n)$ . This bound has been later improved by Czumaj and Rytter [14], and independently Kowalski and Pelc [22], who gave optimal randomized broadcasting algorithms that complete the task in  $O(D \log \frac{n}{D} + \log^2 n)$  time with high probability, matching a known lower bound from [23]. Haeupler and Wajc [17] improved this bound for undirected networks in the model that allows spontaneous transmissions and designed an algorithm that completes broadcasting in time  $O\left(\frac{D \log n \log \log n}{\log D} + \log^{O(1)} n\right)$  with high probability, improved to  $O\left(\frac{D \log n}{\log D} + \log^{O(1)} n\right)$  in [11]. In the model with collision detection for undirected networks, an  $O(D + \log^6 n)$ -time randomized algorithm due to Ghaffari et al. [16] is the first to exploit collisions and surpass the algorithms (and lower bound) for broadcasting without collision detection.

For more details about broadcasting algorithms in various models, see e.g., [11, 14, 16, 20, 27] and the references therein.



### 1.3 New results

We present algorithms for the fundamental tasks of broadcasting and wake-up in multiple access channels (single-hop radio networks) and multi-hop radio networks which require no knowledge of network parameters: nodes need know only their own unique ID.

Our wake-up algorithm takes  $O(\frac{k \log L \log k}{\log \log k})$  time in multiple access channels and  $O(\frac{n \log L \log n}{\log \log n})$  time in multi-hop radio networks, improving dramatically over the previous best  $O(L^3 \log^3 L)$  and  $O(DL^3 \log^3 L)$  respective running times of [26] (recall that  $k \leq n \leq L$ ). This is particularly significant in the case of large labels, since dependency on  $L$  has been improved from cubic to logarithmic. Furthermore, our running time matches the  $O(\frac{n \log L \log n}{\log \log n})$  time of [13], the fastest algorithm with parameter knowledge and small node labels.

Our broadcasting algorithm takes  $O(k \log L \log \log k)$  time in multiple access channels and  $O(n \log L \log \log n)$  time in multi-hop radio networks. This improves over the previous best  $O(L)$  multiple access channel bound [15] in most cases. In radio networks the improvement is even more pronounced, beating not only the  $O(DL^3 \log^3 L)$  result of [26] but also the  $O(n \log^2 L)$ -time algorithm of [9], which was the fastest result for large labels even when network parameters are known. When labels are small (i.e.,  $L = n^{O(1)}$ ), our result matches the best running time for known parameters ( $O(n \log D \log \log D)$  from [13]) for networks of polynomial eccentricity.

We believe the primary value of our work is in challenging the *long-standing assumption that parameter knowledge is necessary for efficient deterministic algorithms* in radio networks and multiple access channels. We show that in fact, deterministic algorithms which do not assume this knowledge can reach the fastest running times for those that do.

### 1.4 Previous approaches

Almost all deterministic broadcasting protocols with sub-quadratic complexity (that is, since [6]) have relied on the concept of *selective families* (or some similar variant thereof, such as selectors). These are families of sets for which one can guarantee that any subset of  $[k] := \{1, 2, \dots, k\}$  below a certain size has an intersection of size exactly 1 with some member of the family [6]. They are useful in the context of radio networks because if the members of the family are interpreted to be the set of nodes which are allowed to transmit in a particular time-step, then after going through each member, any node with a participating in-neighbor and an in-neighborhood smaller than the size threshold will be informed. Most of the recent improvements in broadcasting time have been due to a combination of proving smaller selective families exist, and finding more efficient ways to apply them (i.e., choosing which size of family to apply at which time) [6, 7, 9, 14].

Applying such constructs requires coordination between nodes, which relies on a global clock, making them unsuitable for wake-up. To tackle this issue, Chrobak et al. [8] introduced the concept of *radio synchronizers*. These are a development of selective families which allow nodes to begin their behavior at different times. A further extension to *universal synchronizers* in [4] allowed effectiveness across all in-neighborhood sizes.

Another similar extension of selective families came in 2010 with a paper by De Marco [24], which used a *transmission matrix* to schedule node transmissions for broadcasting. Like radio synchronizers, the application of this matrix allowed nodes to begin their own transmission sequence at any time, and still provided a ‘selective’ property that guaranteed broadcasting progress. The synchronization afforded by the assumption of a global clock allowed this method to beat the time bounds given by radio synchronizers (and previous broadcasting algorithms using selective families).

The proofs of *existence* for selective families, synchronizers, and transmission matrices follow similar lines: a probabilistic candidate object is generated by deciding on each element independently at random with certain carefully chosen probabilities, and then it is proven that the candidate satisfies the desired properties with positive probability, and so *such an object must exist*. These types of proofs are all *non-constructive* (and therefore all resulting algorithms non-explicit; cf. [5, 18] for an explicit construction of selective families with significantly weaker size bounds).

In contrast, results on multiple access channels without parameter knowledge (notably [15, 26]) have not used these types of combinatorial objects, and instead rely on some results from number theory. The algorithm of [26], for instance, is to have nodes transmit periodically, a node with ID  $v$  waiting  $p_v$  steps between transmissions, where  $p_v$  is the  $v^{\text{th}}$  smallest prime number. A number-theoretic result is then employed to show that eventually one node will transmit alone. As a result, these algorithms have the advantage of being explicit, but the disadvantage of slower running times.

## 1.5 Novel approach

We aim to apply the approach of using combinatorial objects proven by the probabilistic method to the setting where network parameters are not known. One way to do this would be to apply selectors (for example) of continually increasing size, until one succeeds. However, since there are two parameters which must meet the correct values for a successful application ( $k$  and  $L$  in the case of medium access channels), running times for this approach are poor. Instead, we define, and prove the existence of, *a single, infinite combinatorial object, which can accommodate all possible values of parameters at the same time*.

Another difference is that for all previous works using selective families or variants thereof, the candidate object has been generated with the same sequence of probabilities for each node. Here, however, in order to achieve good running times we need to have these probabilities depending on the node ID. In essence, this means that nodes effectively use their own ID as an estimate of the maximum ID in the network.

## 1.6 A note on non-explicitness

As mentioned, almost all deterministic broadcasting protocols with sub-quadratic complexity have relied on selective families or variants thereof, and have been non-explicit results. Our work here is also non-explicit, but while this is standard for deterministic radio network algorithms, it may present more of an issue here, since our combinatorial structures are infinite. It is not clear how the protocols we present could be performed by devices with bounded memory, and as such this work is more of a proof-of-concept than a practical algorithm. However, it is possible that our method could be adapted so that nodes' behavior could be generated by a finite-size (i.e., a function of ID) program; this would be a natural and interesting extension to our work, and would overcome the problem.

Another issue which would remain is that nodes must perform the protocol indefinitely, and never become aware that broadcasting has been successfully completed. However, this is unavoidable in the model: Chlebus et al. [6] prove that *acknowledged* broadcasting without parameter knowledge is impossible.

## 2 Combinatorial objects

In this section we present the two combinatorial objects that we wish to use in our algorithms: *unbounded universal synchronizers* and *unbounded transmission schedules*. After defining them in Sections 2.1 and 2.2, we present their main properties in Theorems 3 and 12, and then show how to apply them to obtain new deterministic algorithms for wake-up and broadcasting in multiple access channels and in radio networks (Theorems 19, 20, 22, 23).

### 2.1 Unbounded universal synchronizers

For the task of wake-up, i.e., in the absence of a global clock, we will define an object called an **unbounded universal synchronizer** for use in our algorithm.

We begin by defining the sets of circumstances our algorithm must account for:

► **Definition 1.** An  $(r, k)$ -instance  $X$  is a set  $K$  of  $k$  nodes with

$$\sum_{v \in K} \log v = r$$

and wake-up function  $\omega : K \rightarrow \mathbb{N}$ .

(By using  $v$  as an integer here, we are abusing notation to mean the ID of node  $v$ .)

Here  $r$  is the main parameter we will use to quantify how ‘large’ our input instance is. By using the sum of logarithms of IDs (which approximates the total number of bits needed to write all IDs in use), we capture both the number of participating nodes and the length of IDs in a single parameter. We require  $r$  to be an integer, so we round down accordingly, but we omit floor functions for clarity since they do not affect the asymptotic result.

The *wake-up function*  $\omega$  maps each node to the time-step it wakes up (either spontaneously or by receiving a transmission) when our algorithm is run on this instance. This means that the wake-up function depends on the algorithm, but there is no circular dependency: whether nodes wake-up in time-step  $j$  only depends on the algorithm’s behavior in previous time-steps, and the algorithm’s behavior at time-step  $j$  only depends on the wake-up function up to  $j$ . We will also extend  $\omega$  to sets of nodes in the instance by  $\omega(K) := \min_{v \in K} \omega(v)$ .

We now define the combinatorial object that will be the basis of our algorithm:

► **Definition 2.** For a function  $g : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ , an **unbounded universal synchronizer of delay  $g$**  is a function  $\mathcal{S} : \mathbb{N} \rightarrow \{0, 1\}^{\mathbb{N}}$  such that for any  $(r, k)$ -instance, there is some time-step  $j \leq \omega(K) + g(r, k)$  with  $\sum_{v \in K} \mathcal{S}(v)_{j - \omega(v)} = 1$ .

The *unbounded universal synchronizer*  $\mathcal{S}$  is a function mapping node IDs to a sequence of 0s and 1s, which tell nodes when to listen and transmit respectively. The function  $g$ , which we will call the *delay function*, tells us how many time-steps we must wait before a successful transmission is guaranteed, so this is what we want to asymptotically minimize.

We will apply this object to perform wake-up as follows: each node  $v$  transmits a message in time-step  $j$  (with its time-step count starting upon waking up) iff  $\mathcal{S}(v)_j = 1$ . Then, the property guarantees that at some time-step  $j$  within  $g(r, k)$  time-steps of the first node waking up, any  $(r, k)$ -instance will have a successful transmission. We call this  $\mathcal{S}$  ‘*hitting*’ the  $(r, k)$ -instance at time-step  $j$ . So, our aim is to show the existence of such an object, with  $g$  growing as slowly as possible.

Our main technical result in this section is the following:

► **Theorem 3.** *There exists an unbounded universal synchronizer of delay  $g$ , where*  

$$g(r, k) = O\left(\frac{r \log k}{\log \log k}\right).$$

Our approach to proving Theorem 3 will be to randomly generate a candidate synchronizer, and then prove that with positive probability it does indeed satisfy the required property. Then, for this to be the case, at least one such object must exist.

Our candidate  $S : \mathbb{N} \rightarrow \{0, 1\}^{\mathbb{N}}$  will be generated by independently choosing each  $S(v)_j$  to be  $\mathbf{1}$  with probability  $\frac{c \log v}{j + 2c \log v}$  and  $\mathbf{0}$  otherwise, where  $c$  is some sufficiently large constant to be chosen later.

While  $S$  is drawn from an uncountable set, we will only be concerned with events that depend upon a finite portion of it, and countable unions and intersections thereof. Therefore, we can use as our underlying  $\sigma$ -algebra that generated by the set of all events  $E_{v,j} = \{S : S(v)_j = 1\}$ , where  $v, j \in \mathbb{N}$ , with the corresponding probabilities  $\mathbb{P}[E_{v,j}] = \frac{c \log v}{j + 2c \log v}$ .

We set delay function  $g(r, k) = \frac{c^2 r \log k}{\log \log k}$ .

To simplify our task, we begin with some useful observations:

First we note that since we only care about the asymptotic behavior of  $g$ , we can assume that  $r$  is larger than a sufficiently large constant.

We also note that we need not consider all  $(r, k)$ -instances. For a given  $(r, k)$ -instance and time-step  $j$ , let  $K_j$  be the set of nodes woken up by time  $j$  (with  $k_j := |K_j|$ ), and  $r_j$  be defined as  $r$  but restricted to the nodes in  $K_j$ . Let  $t$  be the earliest time-step such that  $t > g(r_t, k_t)$ , and curtail the  $(r, k)$ -instance to the corresponding  $(r_t, k_t)$ -instance of nodes in  $K_t$ . It is easy to see that if we hit all curtailed  $(r_t, k_t)$ -instances within  $g(r_t, k_t)$  time, we must hit all  $(r, k)$ -instances within  $g(r, k)$  time, so henceforth we will only consider curtailed instances (i.e., we can assume that  $j \leq g(r_j, k_j)$  for all  $j < g(r, k)$ ).

Finally, we observe that, since nodes' behavior is not dependent on the global clock, we can shift all  $(r, k)$ -instances to begin at time-step 0.

To analyze the probability of hitting  $(r, k)$ -instances, define the *load* of a time-step  $f(j)$  to be the expected number of transmissions in that time-step:

► **Definition 4.** For a fixed  $(r, k)$ -instance, the **load  $f(j)$  of a time-step  $j$**  is defined as

$$\sum_{v \in K_j} \mathbb{P}[v \text{ transmits}] = \sum_{v \in K_j} \frac{c \log v}{j - \omega(v) + 2c \log v} .$$

We now seek to bound the load from above and below, since when the load is close to constant we have a good chance of hitting.

► **Lemma 5.** *All time-steps  $j \leq g(r, k)$  have  $f(j) \geq \frac{\log \log k}{2c \log k}$ .*

**Proof.** Fix a time-step  $j \leq g(r, k)$ , let  $K_j$  be the set of nodes awake by time-step  $j$ , and let  $k_j = |K_j|$  and  $r_j = \sum_{v \in K_j} \log v$ , analogous to  $r$  and  $k$ . If  $k_j = k$ , then

$$f(j) \geq \sum_{v \in K} \frac{c \log v}{j + 2c \log v} \geq \frac{cr}{j + 2cr} \geq \frac{cr}{\frac{2c^2 r \log k}{\log \log k}} \geq \frac{\log \log k}{2c \log k} .$$

If  $k_j < k$ , then due to our curtailing of instances, we have  $j \leq g(r_j, k_j)$ . So,

$$f(j) \geq \sum_{v \in K_j} \frac{c \log v}{j + 2c \log v} \geq \frac{cr_j}{j + 2cr_j} \geq \frac{cr_j}{\frac{2c^2 r_j \log k_j}{\log \log k_j}} \geq \frac{\log \log k_j}{2c \log k_j} \geq \frac{\log \log k}{2c \log k} . \quad \blacktriangleleft$$

Having bounded load from below, we also seek to bound it from above. Unfortunately, the load in any particular time-step can be very high, but we can get a good bound on at least a constant fraction of the columns.

► **Lemma 6.** *Let  $F$  denote the set of time-steps  $j \leq g(r, k)$  such that  $\frac{\log \log k}{2c \log k} \leq f(j) \leq \frac{\log \log k}{3}$ . Then  $|F| \geq \frac{cr \log k}{2 \log \log k}$ .*

**Proof.** The total load over all time-steps can be bounded as follows:

$$\begin{aligned} \sum_{j \leq g(r, k)} f(j) &= \sum_{j \leq g(r, k)} \sum_{v \in K_j} \frac{c \log v}{j - \omega(v) + 2c \log v} \leq \sum_{v \in K} \sum_{\omega(v) < j \leq g(r, k)} \frac{c \log v}{j - \omega(v) + 2c \log v} \\ &\leq \sum_{v \in K} c \log v \sum_{j \leq g(r, k)} \frac{1}{j + 2c \log v} \leq \sum_{v \in K} c \log v \ln \frac{2g(r, k)}{4c \log v}. \end{aligned}$$

Let  $K_i = \{v \in K : \frac{r}{k \cdot 2^i} \leq \log v < \frac{r}{k \cdot 2^{i-1}}\}$ , for  $i \geq 1$ , and  $K' = \{v \in K : \log v \geq \frac{r}{k}\}$

If  $\sum_{v \in K_i} \log v > \frac{r}{2^i}$  then  $r < 2^i \sum_{v \in K_i} \log v \leq 2^i \sum_{v \in K_i} \frac{r}{k \cdot 2^i} \leq r$ . This gives a contradiction, so we must have  $\sum_{v \in K_i} \log v \leq \frac{r}{2^i}$ . Then,

$$\begin{aligned} \sum_{j \leq g(r, k)} f(j) &\leq \sum_{v \in K} c \log v \ln \frac{2g(r, k)}{4c \log v} \leq \sum_{i \geq 1} \sum_{v \in K_i} c \log v \ln \frac{g(r, k)}{2c \log v} + \sum_{v \in K'} c \log v \ln \frac{g(r, k)}{2c \log v} \\ &\leq \sum_{i \geq 1} \sum_{v \in K_i} c \log v \ln \frac{g(r, k)}{2c \frac{r}{k \cdot 2^i}} + \sum_{v \in K'} c \log v \ln \frac{g(r, k)}{2c \frac{r}{k}} \\ &= \sum_{i \geq 1} \sum_{v \in K_i} c \log v \ln \frac{ck2^{i-1} \log k}{\log \log k} + \sum_{v \in K'} c \log v \ln \frac{ck \log k}{2 \log \log k} \\ &\leq \sum_{i \geq 1} cr2^{-i} (2 \ln k + (i-1) \ln 2) + 2cr \ln k \leq 5cr \ln k \leq 8cr \log k. \end{aligned}$$

Therefore, at most  $\frac{24cr \log k}{\log \log k}$  time-steps have load higher than  $\frac{\log \log k}{3}$ . Since by Lemma 5 all time-steps have load at least  $\frac{\log \log k}{2c \log k}$ , we have  $|F| \geq g(r, k) - \frac{24cr \log k}{\log \log k} \geq \frac{c^2 r \log k}{2 \log \log k}$  (provided we pick  $c \geq 7$ ). ◀

Now that we have bounded load, we show how it relates to hitting probability. The following lemma, or variants thereof, has been used in several previous works such as [24], but we prove it here for completeness.

► **Lemma 7.** *Let  $x_i$ ,  $i \in [n]$ , be independent  $\{0, 1\}$ -valued random variables with  $\mathbb{P}[x_i = 1] \leq \frac{1}{2}$ , and let  $f = \sum_{i \in [n]} \mathbb{P}[x_i = 1]$ . Then  $\mathbb{P}\left[\sum_{i \in [n]} x_i = 1\right] \geq f4^{-f}$ .*

**Proof.**

$$\begin{aligned} \mathbb{P}\left[\sum_{i \in [n]} x_i = 1\right] &= \sum_{j \in [n]} \mathbb{P}[x_j = 1 \wedge \forall_{i \neq j} x_i = 0] \geq \sum_{j \in [n]} \mathbb{P}[x_j = 1] \cdot \mathbb{P}[\forall_i x_i = 0] \\ &\geq f \cdot \mathbb{P}[\forall_i x_i = 0] = f \cdot \prod_{i \in [n]} (1 - \mathbb{P}[x_i = 1]) \geq f \cdot \prod_{i \in [n]} 4^{-\mathbb{P}[x_i = 1]} \\ &= f \cdot 4^{-\sum_{i \in [n]} \mathbb{P}[x_i = 1]} = f4^{-f}. \end{aligned}$$

We can use Lemma 7 to show that the probability that we hit on our ‘good’ time-steps (those in  $F$ ) is high:

► **Lemma 8.** *For any time-step  $j \in F$ , the probability that  $j$  hits is at least  $\frac{\log \log k}{3c \log k}$ .*

**Proof.**  $\frac{\log \log k}{2c \log k} \leq f(j) \leq \frac{\log \log k}{3}$ , and so  $f(j)4^{-f(j)}$  is minimized at  $f(j) = \frac{\log \log k}{2c \log k}$  and is therefore at least  $\frac{\log \log k}{2c \log k} 4^{-\frac{\log \log k}{2c \log k}} \geq \frac{\log \log k}{3c \log k}$ . ◀

We now bound the number of possible instances we must hit:

► **Lemma 9.** *For any (sufficiently large)  $r$ , the number of unique  $(r, k)$ -instances is at most  $2^{5r}$ .*

**Proof.** The total number of bits used in all node IDs in the instance is at most  $r$ . There are at most  $2^{r+1}$  possible bit-strings of length at most  $r$ , and at most  $2^r$  ways of dividing each of these into substrings (for individual IDs), giving at most  $2^{2r+1}$  sets of node IDs. The number of possible wake-up functions  $\omega : K \rightarrow \mathbb{N}$  is at most  $g(r, k)^k$ , since all nodes must be awake by  $g(r, k)$  time or the instance would have been curtailed.

$$g(r, k)^k = 2^{k \log g(r, k)} \leq 2^{1.1k \log r} = 2^{1.1(k \log k + k \log \frac{r}{k})} \leq 2^{1.3(k \log(k^{0.9}) + r)} \leq 2^{2.9r}.$$

So, the total number of possible  $(r, k)$ -instances is at most  $2^{2r+1+2.9r} \leq 2^{5r}$ . ◀

► **Lemma 10.** *For any (sufficiently large)  $r$ , the probability that  $S$  does not hit all  $(r, k)$ -instances is at most  $2^{-3r}$ .*

**Proof.** Fix some  $(r, k)$ -instance. The probability that it is not hit within  $g(k, r)$  time-steps is at most

$$\prod_{j \in F} \left(1 - \frac{\log \log k}{3c \log k}\right) \leq e^{-|F| \frac{\log \log k}{3c \log k}} \leq e^{-\frac{2}{3}cr} = 2^{-\frac{2cr}{3 \ln 2}},$$

by Lemma 8. Hence, if we set  $c = 9$ , by a union bound the probability that any  $(r, k)$ -instance is not hit is at most  $2^{5r} \cdot 2^{-\frac{18r}{3 \ln 2}} \leq 2^{-3r}$ . ◀

We can now prove our main theorem on unbounded universal synchronizers (Theorem 3):

**Proof.** By Lemma 10 and a union bound over  $r$ , the probability that  $S$  does not hit all instances is at most  $\sum_{r \in \mathbb{N}} 2^{-3r} < 1$ . Therefore  $S$  is an unbounded universal synchronizer of delay  $g$  with non-zero probability, so such an object must exist. ◀

## 2.2 Unbounded transmission schedules

For the task of broadcasting, i.e., when a global clock is available, we can make use of the global clock to improve our running time. We again define an infinite combinatorial object, which we will call an **unbounded transmission schedule**. We use the same definition of  $(r, k)$ -instances as in the previous section.

► **Definition 11.** For a function  $h : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ , an **unbounded transmission schedule of delay  $h$**  is a function  $T : \mathbb{N} \times \mathbb{N} \rightarrow \{0, 1\}^{\mathbb{N}}$  such that  $T(v, \omega(v))_j = 0$  for any  $j < \omega(v)$ , and for any  $(r, k)$ -instance there is some time-step  $j \leq \omega(K) + h(r, k)$  with  $\sum_{v \in K} T(v, \omega(v))_j = 1$ .

The difference here from an unbounded universal synchronizer is that nodes now know the global time-step in which they wake up, and so their transmission patterns can depend upon it. This is the second argument of the function  $T$ . The other difference in the meaning of the definition is that the output of  $T$  now corresponds to absolute time-step numbers, rather than being relative to each node's wake-up time as for unbounded universal synchronizers. That is, the  $j^{\text{th}}$  entry of a node's output sequence tells it how it should behave in global time-step  $j$ , rather than  $j$  time-steps after it wakes up.

Our existence result for unbounded transmission schedules is the following:

► **Theorem 12.** *There exists an **unbounded transmission schedule of delay  $h$** , where  $h(r, k) = O(r \log \log k)$ .*

Our method will again be to randomly generate a candidate unbounded transmission schedule  $T$ , and then prove that it satisfies the required property with positive probability, so some such object must exist.

Let  $d$  be a constant to be chosen later. Our candidate object  $T$  will be generated as follows: for each node  $v$ , we generate a transmission sequence  $s_{v,j}$ ,  $j \in \mathbb{N}$ , with  $s_{v,j}$  independently chosen to be 1 with probability  $\frac{d \log v \log \log j}{j + 2d \log v \log \log j}$  and 0 otherwise.

However, these will not be our final probabilities for generating  $T$ . To make use of our global clock, we will also divide time into short *phases* during which transmission probability will decay exponentially. The lengths of these phases will be based on a function  $z(j) := 2^{\lceil 1 + \log \log \log j \rceil}$ , i.e.,  $\log \log j$  rounded up to the next-plus-one power of 2. An important property to note is that for all  $i$ ,  $z(i) | z(i+1)$ . We also generate a sequence  $p_{v,j}$ ,  $j \in \mathbb{N}$  of *phase values*, each chosen independently and uniformly at random from the real interval  $[0, 1]$ . These, combined with the global time-step number and current phase length, will give us our final generation probabilities.

We set  $T(v, \omega(v))_j$  to equal 1 iff  $s_{v,j-\omega(v)} = 1$  and  $p_{v,j-\omega(v)} \leq 2^{-j \bmod z(j-\omega(v))}$ .

It can then be seen that

$$\mathbb{P}[T(v, \omega(v))_j = 1] = \frac{d \log v \log \log(j - \omega(v))}{(j - \omega(v) + 2d \log v \log \log(j - \omega(v))) 2^{j \bmod z(j - \omega(v))}} .$$

The reason we split the process of random generation into two steps (using our phase values) is that now, if we shift all wake-up times in an  $(r, k)$ -instance by the same multiple of  $z(x)$ , then node behavior in the first  $x$  time-steps after  $\omega(K)$  does not change. We will require this property when analyzing  $T$ .

Our probabilistic analysis is with respect to the  $\sigma$ -algebra generated by all events  $E_{v, \omega(v), j} = \{T : T(v, \omega(v))_j = 1\}$ , with  $v, \omega(v), j \in \mathbb{N}$ , and with the corresponding probabilities given above.

Let **load  $f(j)$  of a time-step  $j$**  be again defined as the expected number of transmissions in that time-step:

$$f(j) := \sum_{v \in K_j} \frac{d \log v \log \log(j - \omega(v))}{(j - \omega(v) + 2d \log v \log \log(j - \omega(v))) 2^{j \bmod z(j - \omega(v))}} .$$

We will set our delay function  $h(r, k) = d^2 r \log \log k$ .

Again we make some observations that allow us to narrow the circumstances we must consider: firstly that we can again assume that  $r$  is larger than a sufficiently large constant, and secondly that we need only look at curtailed instances (i.e., we can assume that  $j - \omega(K) \leq h(r_j, k_j)$  for all  $j < h(r, k)$ ). This time, however, we cannot shift instances to begin at time-step 0, because node behavior is dependent upon global time-step number.

We follow a similar line of proof as before, except with some extra complications in dealing with phases. We first consider only time-steps at the beginning of each phase, i.e., time-steps  $\omega(K) < j \leq \omega(K) + h(r, k)$  with  $j \bmod z(h(r, k)) \equiv 0$ , and we will call these *basic* time-steps. Notice that for basic time-steps,

$$f(j) = \sum_{v \in K_j} \frac{d \log v \log \log(j - \omega(v))}{j - \omega(v) 2d \log v \log \log(j - \omega(v))} .$$

We bound the load of basic time-steps from below:

► **Lemma 13.** *All basic time-steps  $j$  have  $f(j) \geq \frac{1}{2d}$ .*

**Proof.** Fix a basic time-step  $j$ , let  $K_j$  be the set of nodes awake by time-step  $j$ , and let  $k_j = |K_j|$  and  $r_j = \sum_{v \in K_j} \log v$ , analogous to  $r$  and  $k$ . If  $k_j = k$ , then

$$\begin{aligned} f(j) &\geq \sum_{v \in K} \frac{d \log v \log \log(j - \omega(v))}{j - \omega(v) + 2d \log v \log \log(j - \omega(v))} \geq \sum_{v \in K} \frac{d \log v \log \log h(r, k)}{h(r, k) + 2d \log v \log \log h(r, k)} \\ &\geq \sum_{v \in K} \frac{d \log v \log \log k}{2d^2 r \log \log k} \geq \frac{dr}{2d^2 r} = \frac{1}{2d} . \end{aligned}$$

If  $k_j < k$ , then due to our curtailing of instances, we have  $j - \omega(K) \leq h(r_j, k_j)$ . So,

$$\begin{aligned} f(j) &\geq \sum_{v \in K_j} \frac{d \log v \log \log(j - \omega(v))}{j - \omega(v) + 2d \log v \log \log(j - \omega(v))} \geq \sum_{v \in K} \frac{d \log v \log \log h(r_j, k_j)}{h(r_j, k_j) + 2d \log v \log \log h(r, k)} \\ &\geq \sum_{v \in K} \frac{d \log v \log \log k_j}{2d^2 r_j \log \log k_j} \geq \frac{dr_j}{2d^2 r_j} = \frac{1}{2d} . \end{aligned} \quad \blacktriangleleft$$

We next examine time-steps at the end of phases, i.e., with  $\omega(K) < j \leq \omega(K) + h(r, k)$  and  $j \bmod z(h(r, k)) \equiv -1$ , and call these *ending* time-steps. Note that for ending time-steps,

$$f(j) = \sum_{v \in K_j} \frac{d \log v \log \log(j - \omega(v))}{(j - \omega(v) + 2d \log v \log \log(j - \omega(v)))2^{z(j - \omega(v)) - 1}} .$$

We bound the load of (a constant fraction of) ending time-steps from above:

► **Lemma 14.** *Let  $\mathcal{F}$  denote the set of ending time-steps  $j$  such that  $f(j) \leq 1$ . Then  $|\mathcal{F}| \geq \frac{d^2 r}{2}$ .*

**Proof.** Let  $t$  be the first ending time-step. The total load over all ending time-steps can be bounded as follows:

$$\sum_{\text{ending timestep } j} f(j) \leq \sum_{i=0}^{h(r, k)/z(h(r, k))} f(t + iz(h(r, k))) \leq \sum_{i=0}^{d^2 r} f(t + iz(h(r, k))) .$$

Applying the definition of  $f$ ,  $f(t + iz(h(r, k)))$  is equal to:

$$\sum_{v \in K_{t+iz(h(r, k))}} \frac{d \log v \log \log(t + iz(h(r, k)) - \omega(v))2^{-z(t+iz(h(r, k)) - \omega(v)) - 1}}{(t + iz(h(r, k)) - \omega(v) + 2d \log v \log \log(t + iz(h(r, k)) - \omega(v)))}$$

which is bounded from above when  $t - \omega(v) = 0$ :

$$\begin{aligned} f(t + iz(h(r, k))) &\leq \sum_{v \in K_{t+iz(h(r, k))}} \frac{d \log v \log \log(iz(h(r, k)))}{(iz(h(r, k)) + 2d \log v \log \log(iz(h(r, k))))2^{z(iz(h(r, k)))}} \\ &\leq \sum_{v \in K_{t+iz(h(r, k))}} \frac{d \log v \log \log(iz(h(r, k)))}{iz(h(r, k))2^{z(iz(h(r, k)))}} . \end{aligned}$$



So,

$$\begin{aligned}
\sum_{\text{ending timestep } j} f(j) &\leq \sum_{i=0}^{d^2 r} \sum_{v \in K_{t+iz(h(r,k))}} \frac{d \log v \log \log(iz(h(r,k)))}{iz(h(r,k)) 2^{z(iz(h(r,k)))}} \\
&\leq \sum_{v \in K} \sum_{i=0}^{d^2 r} \frac{d \log v \log \log(iz(h(r,k)))}{iz(h(r,k)) 2^{z(iz(h(r,k)))}} \\
&\leq \sum_{v \in K} \sum_{i=0}^{d^2 r} \frac{2d \log v \log \log(iz(h(r,k)))}{iz(h(r,k)) \log^2(iz(h(r,k)))} \\
&\leq \sum_{v \in K} 2d \log v \sum_{i=0}^{\infty} \frac{\log \log(iz(h(r,k)))}{iz(h(r,k)) \log^2(iz(h(r,k)))} \leq 10dr .
\end{aligned}$$

Here the last inequality follows since the second sum converges to a constant less than 5, which can be seen by inspection of the first three terms and using the integral bound  $\int_2^{\infty} \frac{\log \log x}{x \log^2 x} < 2$

Therefore, at most  $10dr$  ending time-steps have load higher than 1, and so at least  $d^2 r - 10dr \geq \frac{d^2 r}{2}$  (provided we set  $d \geq 5$ ) ending time-steps have  $f(j) \leq 1$ . ◀

We can use Lemma 14 to show that we have sufficiently many time-steps with load within the interval  $[\frac{1}{2d}, 1]$ :

► **Lemma 15.** *Let  $\mathcal{F}$  be the set of time-steps  $\omega(K) < j \leq \omega(K) + h(r, k)$  with  $\frac{1}{2d} \leq f(j) \leq 1$ . Then  $|\mathcal{F}| \geq \frac{d^2 r}{2}$ .*

**Proof.** It can be seen that load decreases by at most a multiplicative factor of 3 between consecutive time-steps (since the contribution of each node decreases by at most a factor of 3). So, since every base time-step has load at least  $\frac{1}{2d}$ , for every ending timestep  $j$  with  $f(j) \leq 1$ , there is some  $j'$  in the preceding phase with  $\frac{1}{2d} \leq f(j') \leq 1$ . ◀

Since these time-steps have constant load, they have constant probability of hitting:

► **Lemma 16.** *For any time-step  $j \in \mathcal{F}$ , the probability that  $j$  hits is at least  $\frac{1}{3d}$ .*

**Proof.** By Lemma 7, the probability that  $j$  hits is at least  $f(j)4^{-f(j)}$ . This is minimized over the range  $[\frac{1}{2d}, 1]$  at  $f(j) = \frac{1}{2d}$  and is therefore at least  $\frac{4^{-\frac{1}{2d}}}{2d} \geq \frac{1}{3d}$ . ◀

We now need to know how many unique  $(r, k)$ -instances we must hit. Since we are only concerned with the first  $h(r, k)$  time-steps after the first node wakes up, we need only consider  $(r, k)$ -instances which are unique within this time range.

► **Lemma 17.** *For any (sufficiently large)  $r$ , the number of unique (up to the first  $h(r, k)$  steps after activation)  $(r, k)$ -instances is at most  $2^{5r}$ .*

**Proof.** As before (in Lemma 9) there are at most  $2^{2r+1}$  sets of node IDs. The number of possible wake-up functions  $\omega : K \rightarrow \mathbb{N}$  for a fixed  $\omega(K)$  is again at most  $h(r, k)^k$ , and though we are using a different delay function to the previous section, the calculations used to prove Lemma 9 still hold.

$$h(r, k)^k = 2^{k \log h(r, k)} \leq 2^{1.1k \log r} = 2^{1.1(k \log k + k \log \frac{r}{k})} \leq 2^{1.3(k \log(k^{0.9}) + r)} \leq 2^{2.9r} .$$

However, now our object does depend on  $\omega(K)$ , though as we noted we can shift all activation times by a multiple of  $z(h(r, k))$  and behavior during the time-steps we analyze is

---

**Algorithm 1** Wake-up at a node  $v$ .

---

**for**  $j$  from 1 to  $\infty$ , in time-step  $\omega(v) + j$ , **do**  
 $v$  transmits iff  $S(v)_j = 1$   
**end for**

---



---

**Algorithm 2** Broadcasting at a node  $v$ .

---

**for**  $j$  from 1 to  $\infty$ , in time-step  $j$ , **do**  
 $v$  transmits iff  $T(v, \omega(v))_j = 1$   
**end for**

---

unchanged. So our total number of instances to consider is multiplied by  $z(h(r, k))$ , and is upper bounded by  $2^{2r+1+2.9r} z(h(r, k)) \leq 2^{5r}$ . ◀

► **Lemma 18.** *For any (sufficiently large)  $r$ , the probability that  $T$  does not hit all  $(r, k)$ -instances is at most  $2^{-3r}$ .*

**Proof.** Fix some  $(r, k)$ -instance. The probability that it is not hit within  $h(r, k)$  time-steps is at most

$$\prod_{j \in \mathcal{F}} \left(1 - \frac{1}{3d}\right) \leq e^{-\frac{|\mathcal{F}|}{3d}} \leq e^{-\frac{dr}{6}} = 2^{-\frac{dr}{6 \ln 2}}.$$

Hence, if we set  $d = 34$ , by a union bound the probability that any  $(r, k)$ -instance is not hit is at most  $2^{5r} \cdot 2^{-\frac{34r}{6 \ln 2}} \leq 2^{-3r}$ . ◀

We can now prove our main theorem on unbounded transmission schedules (Theorem 12):

**Proof.** By Lemma 18 and a union bound over  $r$ , the probability that  $T$  does not hit all instances is at most  $\sum_{r \in \mathbb{N}} 2^{-3r} < 1$ . Therefore  $T$  is an unbounded transmission schedule of delay  $h$  with non-zero probability, so such an object must exist. ◀

### 3 Algorithms for multiple access channels

Armed with our combinatorial objects, our algorithms are now extremely simple, and are the same for multiple access channels as for multi-hop radio networks.

Let  $S$  be an unbounded universal synchronizer of delay  $g$ , where  $g(r, k) = O\left(\frac{r \log k}{\log \log k}\right)$ , and  $T$  be an unbounded transmission schedule of delay  $h$ , where  $h(r, k) = O(r \log \log k)$ .

Our algorithms are simply applications of these objects.

► **Theorem 19.** *Algorithm 1 performs wake-up in multiple access channels in time  $O\left(\frac{k \log L \log k}{\log \log k}\right)$ , without knowledge of  $k$  or  $L$ .*

**Proof.** By the definition of an unbounded universal synchronizer, there is some time-step within

$$g(r, k) = O\left(\frac{r \log k}{\log \log k}\right) = O\left(\frac{k \log L \log k}{\log \log k}\right)$$

time-steps of the first activation in which only one node transmits, and this completes wake-up. ◀

► **Theorem 20.** *Algorithm 2 performs broadcasting in multiple access channels in time  $O(k \log L \log \log k)$ , without knowledge of  $k$  or  $L$ .*

**Proof.** By the definition of an unbounded transmission schedule, there is some time-step within  $h(r, k) = O(r \log \log k) = O(k \log L \log \log k)$  time-steps of the first activation in which only one node transmits, and this completes broadcasting. ◀

## 4 Algorithms for radio networks

To see how our results on multiple access channels (Theorems 19 and 20) transfer directly to multi-hop radio networks, we apply the analysis method of [13] for radio network protocols. The idea is that we fix a shortest path  $p = (p_0, p_1, \dots, p_d)$  from some *source* node to some *target* node, and then classify all nodes into *layers* depending on the furthest node along the path to which they are an in-neighbor, i.e., layer  $L_i = \{v : \max j \text{ such that } (v, p_j) \in E = i\}$ . We wish to quantify how long a layer can remain *leading*, that is, the furthest layer to contain awake nodes. The key point is that we can regard these layers as multiple access channels: though they are not necessarily cliques, all we need is for a single node in the layer to transmit and then the layer ceases to be leading. Once the final layer ceases to be leading, the target node must be informed, and since this node was chosen arbitrarily we obtain a time-bound for informing the entire network. Thereby the problem is reduced to a sequence of at most  $D$  single-hop instances, whose sizes sum to at most  $n$ . For full details of this analysis method see [13].

Therefore we can use the following lemma from [13] (paraphrased to fit our notation) to analyze how our algorithms perform in radio networks.

► **Lemma 21** (Lemma 10 of [13]). *Let  $X : [n] \rightarrow \mathbb{N}$  be a non-decreasing function, and define  $Y(n)$  to be the supremum of the function  $\sum_{i=1}^n X(\ell_i)$ , where non-negative integers  $\ell_i$  satisfy the constraint  $\sum_{i=1}^n \ell_i \leq n$ . If a broadcast or wake-up protocol ensures that any layer of size  $\ell$  remains leading for no more than  $X(\ell)$  time-steps, then all nodes wake up within  $Y(n)$  time-steps.*

► **Theorem 22.** *Algorithm 1 performs wake-up in radio networks in time  $O(\frac{n \log L \log n}{\log \log n})$ , without knowledge of  $n$  or  $L$ .*

**Proof.** By Theorem 19, any layer of size  $\ell$  remains leading for no more than  $X(\ell)$  time-steps, where  $X(\ell) = O(\frac{\ell \log L \log \ell}{\log \log \ell})$ .  $Y(n, h)$  is then maximized by setting  $\ell_1 = n$  and  $\ell_i = 0$  for every  $i > 1$ . So, by Lemma 21, wake-up is performed for the entire radio network in  $O(\frac{n \log L \log n}{\log \log n})$  time. ◀

► **Theorem 23.** *Algorithm 2 performs broadcasting in radio networks in  $O(n \log L \log \log n)$  time, without knowledge of  $n$  or  $L$ .*

**Proof.** By Theorem 20, any layer of size  $\ell$  remains leading for no more than  $X(\ell)$  time-steps, where  $X(\ell) = O(\ell \log L \log \log \ell)$ .  $Y(n, h)$  is then maximized by setting  $\ell_1 = n$  and  $\ell_i = 0$  for  $i > 1$ . So, by Lemma 21, broadcasting is performed for the entire radio network in  $O(n \log L \log \log n)$  time. ◀

## 5 Conclusions

We have shown that *deterministic algorithms* for communications primitives in multiple access channels and multi-hop radio networks *need not assume parameter knowledge, or small IDs*, to be efficient. One possible next step would be to show a means by which nodes could generate efficient transmission schedules based on some finite (i.e., with size bounded by some function of ID) advice string; this would go some way towards making the algorithm practical.

---

### References

- 1 N. Alon, A. Bar-Noy, N. Linial, and D. Peleg. A lower bound for radio broadcast. *Journal of Computer and System Sciences*, 43(2):290–298, 1991.
- 2 R. Bar-Yehuda, O. Goldreich, and A. Itai. On the time-complexity of broadcast in multi-hop radio networks: An exponential gap between determinism and randomization. *Journal of Computer and System Sciences*, 45(1):104–126, 1992.
- 3 B. Chlebus, L. Gasieniec, D. R. Kowalski, and T. Radzik. On the wake-up problem in radio networks. In *Proceedings of the 32nd Annual International Colloquium on Automata, Languages and Programming (ICALP)*, pages 347–359, 2005.
- 4 B. Chlebus and D. R. Kowalski. A better wake-up in radio networks. In *Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 266–274, 2004.
- 5 B. Chlebus and D. R. Kowalski. Almost optimal explicit selectors. In *Proceedings of the 15th International Symposium on Fundamentals of Computation Theory (FCT)*, pages 270–280, 2005.
- 6 B. S. Chlebus, L. Gasieniec, A. Gibbons, A. Pelc, and W. Rytter. Deterministic broadcasting in unknown radio networks. *Distributed Computing*, 15(1):27–38, 2002.
- 7 B. S. Chlebus, L. Gasieniec, A. Östlin, and J. M. Robson. Deterministic radio broadcasting. In *Proceedings of the 27th Annual International Colloquium on Automata, Languages and Programming (ICALP)*, pages 717–728, 2000.
- 8 M. Chrobak, L. Gasieniec, and D. R. Kowalski. The wake-up problem in multihop radio networks. *SIAM Journal on Computing*, 36(5):1453–1471, 2007.
- 9 M. Chrobak, L. Gasieniec, and W. Rytter. Fast broadcasting and gossiping in radio networks. *Journal of Algorithms*, 43(2):177–189, 2002.
- 10 A. E. F. Clementi, A. Monti, and R. Silvestri. Distributed broadcasting in radio networks of unknown topology. *Theoretical Computer Science*, 302(1–3):337–364, 2003.
- 11 A. Czumaj and P. Davies. Exploiting spontaneous transmissions for broadcasting and leader election in radio networks. In *Proceedings of the 36th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 3–12, 2017.
- 12 A. Czumaj and P. Davies. Brief announcement: Randomized blind radio networks. In *Proceedings of the 32nd International Symposium on Distributed Computing (DISC)*, pages 44:1–44:3, 2018.
- 13 A. Czumaj and P. Davies. Deterministic communication in radio networks. *SIAM Journal on Computing*, 47(1):218–240, 2018.
- 14 A. Czumaj and W. Rytter. Broadcasting algorithms in radio networks with unknown topology. In *Proceedings of the 44th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 492–501, 2003.
- 15 L. Gasieniec, A. Pelc, and D. Peleg. The wakeup problem in synchronous broadcast systems. *SIAM Journal on Discrete Mathematics*, 14(2):207–222, 2001.

- 16 M. Ghaffari, B. Haeupler, and M. Khabbaziyan. Randomized broadcast in radio networks with collision detection. In *Proceedings of the 32nd Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 325–334, 2013.
- 17 B. Haeupler and D. Wajc. A faster distributed radio broadcast primitive. In *Proceedings of the 35th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 361–370, 2016.
- 18 P. Indyk. Explicit constructions of selectors and related combinatorial structures, with applications. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 697–704, 2002.
- 19 T. Jurdziński and G. Stachowiak. Probabilistic algorithms for the wakeup problem in single-hop radio networks. *Theory of Computing Systems*, 38(3):347–367, 2005.
- 20 D. Kowalski. On selection problem in radio networks. In *Proceedings of the 24th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 158–166, 2005.
- 21 D. Kowalski and A. Pelc. Faster deterministic broadcasting in ad hoc radio networks. *SIAM Journal on Discrete Mathematics*, 18(2):332–346, 2004.
- 22 D. Kowalski and A. Pelc. Broadcasting in undirected ad hoc radio networks. *Distributed Computing*, 18(1):43–57, 2005.
- 23 E. Kushilevitz and Y. Mansour. An  $\Omega(D \log(N/D))$  lower bound for broadcast in radio networks. *SIAM Journal on Computing*, 27(3):702–712, 1998.
- 24 G. De Marco. Distributed broadcast in unknown radio networks. *SIAM Journal on Computing*, 39(6):2162–2175, 2010.
- 25 G. De Marco and A. Pelc. Faster broadcasting in unknown radio networks. *Information Processing Letters*, 79(2):53–56, 2001.
- 26 G. De Marco, M. Pelegrini, and G. Sburlati. Faster deterministic wakeup in multiple access channels. *Discrete Applied Mathematics*, 155(8):898–903, 2007.
- 27 D. Peleg. Time-efficient broadcasting in radio networks: A review. In *Proceedings of the 4th International Conference on Distributed Computing and Internet Technology (ICDCIT)*, pages 1–18, 2007.
- 28 D. E. Willard. Log-logarithmic selection resolution protocols in a multiple access channel. *SIAM Journal on Computing*, 15(2):468–477, 1986.



# Detecting Cliques in CONGEST Networks

Artur Czumaj<sup>1</sup>

Department of Computer Science and Centre for Discrete Mathematics and its Applications (DIMAP), University of Warwick, UK  
A.Czumaj@warwick.ac.uk

Christian Konrad<sup>2</sup>

Department of Computer Science, University of Bristol, UK  
christian.konrad@bristol.ac.uk

---

## Abstract

The problem of detecting network structures plays a central role in distributed computing. One of the fundamental problems studied in this area is to determine whether for a given graph  $H$ , the input network contains a subgraph isomorphic to  $H$  or not. We investigate this problem for  $H$  being a clique  $K_\ell$  in the classical distributed CONGEST model, where the communication topology is the same as the topology of the underlying network, and with limited communication bandwidth on the links.

Our first and main result is a lower bound, showing that detecting  $K_\ell$  requires  $\Omega(\sqrt{n}/b)$  communication rounds, for every  $4 \leq \ell \leq \sqrt{n}$ , and  $\Omega(n/(\ell b))$  rounds for every  $\ell \geq \sqrt{n}$ , where  $b$  is the bandwidth of the communication links. This result is obtained by using a reduction to the set disjointness problem in the framework of two-party communication complexity. We complement our lower bound with a two-party communication protocol for listing all cliques in the input graph, which up to constant factors communicates the same number of bits as our lower bound for  $K_4$  detection. This demonstrates that our lower bound cannot be improved using the two-party communication framework.

**2012 ACM Subject Classification** Theory of computation → Distributed algorithms, Networks → Network algorithms

**Keywords and phrases** Lower bounds, CONGEST, subgraph detection, two-party communication

**Digital Object Identifier** 10.4230/LIPIcs.DISC.2018.16

## 1 Introduction

We study the problem of detecting network structures in a distributed environment, which is a fundamental problem in modern computing. Our focus is on the *subgraph detection problem*, in which for a given graph  $H$ , one wants to determine whether the network graph  $G$  contains a subgraph isomorphic to  $H$  or not. We investigate this problem for  $H$  being a clique  $K_\ell$  for  $\ell \geq 4$ .

The nowadays classical distributed *CONGEST model* (see, e.g., [18]) is a variant of the classical LOCAL model of distributed computation (where in each round network nodes can send through all incident links messages of unrestricted size) with limited communication bandwidth. The distributed system is represented as a network (undirected graph)  $G = (V, E)$

---

<sup>1</sup> Research partially supported by the Centre for Discrete Mathematics and its Applications (DIMAP), by EPSRC award EP/D063191/1, by EPSRC award EP/N011163/1, and by an IBM Faculty Award.

<sup>2</sup> Most of work on this paper has been carried out while the author was at the University of Warwick, where he was supported by the Centre for Discrete Mathematics and its Applications (DIMAP) and by EPSRC award EP/N011163/1.



with  $n = |V|$  nodes, where network nodes execute distributed algorithms in synchronous rounds, and the nodes collaborate to solve a graph problem with input  $G$ . Each node is assumed to have a unique identifier from  $\{0, \dots, \text{poly}(n)\}$ . In any single round, all nodes can:

- (i) perform an unlimited amount of local computation,
- (ii) send a possibly different  $\mathfrak{b}$ -bit message to each of their neighbors, and
- (iii) receive all messages sent to them.

We measure the *complexity* of an algorithms by the number of synchronous rounds required.

In accordance with the standard terminology in the literature, we assume  $\mathfrak{b} = \mathcal{O}(\log n)$ ; we note though that our analysis generalizes to other settings of  $\mathfrak{b}$  in a straightforward manner. (We note that in our lower bound for detecting  $K_4$  and  $K_\ell$  in Section 2, to ensure full generality of presentation, we will make the analysis parametrized by the message size  $\mathfrak{b}$ , in which case we will refer to such model of distributed computation as  $\text{CONGEST}_\mathfrak{b}$ , the CONGEST model with messages of size  $\mathfrak{b}$ .)

Our goal is, for a given network  $G = (V, E)$  and  $\ell \geq 4$ , to solve the *subgraph detection problem* for a clique  $K_\ell$ , that is, to design an algorithm in the CONGEST model such that

- (i) if  $G$  contains a copy of  $K_\ell$ , then with probability  $\geq \frac{2}{3}$  at least one node outputs 1, and
- (ii) if  $G$  does not contain a copy of  $K_\ell$ , then with probability  $\geq \frac{2}{3}$  no node outputs 1.

The subgraph detection problem is a local problem: it can be solved efficiently solely on the basis of local information. In particular, in the CONGEST model, the problem of finding  $K_\ell$  in a graph can be trivially solved in  $\mathcal{O}(n)$  rounds, or in fact, in  $\mathcal{O}(\max_{u \in V} \deg_G(u))$  rounds, where  $\deg_G(u)$  denotes the degree of node  $u$  in  $G$ . Indeed, if each node sends its entire neighborhood to all its neighbors, then afterwards, each node will be aware of all its neighbors and of their neighbors. Therefore, in particular, each node will be able to detect all cliques it belongs to. Since for each node  $u$ , the task of sending its entire neighborhood to all its neighbors can be performed in  $\mathcal{O}(\deg_G(u))$  rounds in the CONGEST model, the total number of rounds for the entire network is  $\mathcal{O}(\max_{u \in V} \deg_G(u)) = \mathcal{O}(n)$  rounds. In view of this simple observation, the main challenge in the clique  $K_\ell$  detection problem is whether this task can be performed in a *sublinear number of rounds*.

## 1.1 Our results

In this paper, we give the first non-trivial lower bound for the complexity of detecting a clique  $K_\ell$  in the  $\text{CONGEST}_\mathfrak{b}$  model, for  $\ell \geq 4$ . In Theorem 5, we prove that every algorithm in the  $\text{CONGEST}_\mathfrak{b}$  model that with probability at least  $\frac{2}{3}$  detects  $K_\ell$ , for  $\ell \geq 4$  and  $\ell = \mathcal{O}(\sqrt{n})$ , requires  $\Omega(\sqrt{n}/\mathfrak{b})$  rounds. Further, if  $\ell = \omega(\sqrt{n})$ , then  $\Omega(n/(\ell \mathfrak{b}))$  rounds are required. We are not aware of any other non-trivial (super-constant) lower bound for this problem in the  $\text{CONGEST}_\mathfrak{b}$  model.

We complement our lower bound with a two-party communication protocol for listing all cliques in the input graph (see Theorem 10), which up to constant factors communicates the same number of bits as our lower bound for  $K_4$  detection. This demonstrates that our lower bound is essentially tight in this framework, and cannot be improved using the two-party communication approach.

## 1.2 Techniques: Framework of two-party communication complexity

Our main results, the lower bound of clique detection in Theorem 5 and the upper bound in Theorem 10, rely on the *two-party communication complexity* framework and the use of a tight lower bound for the set disjointness problem in this framework.



We consider the classical two-party communication complexity setting (cf. [16]) in which two players, Alice and Bob, each have some private input  $X$  and  $Y$ . The players' goal is to compute a joint function  $f(X, Y)$ , and the complexity measure used is the number of bits Alice and Bob must exchange to compute  $f(X, Y)$ . In the two-party communication problem of *set disjointness*, Alice's input is  $X \in \{0, 1\}^n$  and Bob holds  $Y \in \{0, 1\}^n$ , and their goal is to compute  $\text{DISJ}_n(X, Y) := \bigvee_{i=1}^n X_i \wedge Y_i$ . In a seminal work, Kalyanasundaram and Schnitger [14] showed that in any randomized communication protocol, the players must exchange  $\Omega(n)$  bits to solve the set disjointness problem with constant success probability.

► **Theorem 1** ([14]). *The randomized two-party communication complexity of set disjointness is  $\Omega(n)$ . That is, for any constant  $p > 0$ , any randomized two-party communication protocol that computes  $\text{DISJ}_n(X, Y)$  with probability at least  $p$ , has two-party communication complexity  $\Omega(n)$ .*

Our main result, the lower bound for detecting  $K_\ell$  in the CONGEST model, relies on a reduction from the two-party communication problem of set disjointness. The two-party communication framework, and, in particular, the two-party set disjointness problem, have been frequently used in the past to construct lower bounds for the CONGEST model, see, e.g., [4, 7, 9, 11, 15]. A typical approach relies on a construction of a special graph  $G = (V, E)$  with some fixed edges and some edges depending on the input of Alice and Bob. One partitions the nodes of  $G$  into two disjoint sets  $V_A$  and  $V_B$ . Let  $\mathcal{C}$  be the  $(V_A, V_B)$ -cut, that is, the set of edges in  $G$  with one endpoint in  $V_A$  and one endpoint in  $V_B$ . Let  $E_A$  be the edge set of  $G[V_A]$  (subset of  $E$  on vertex set  $V_A$ ) and  $E_B$  be the edge set of  $G[V_B]$ . We consider a scenario where Alice's input is represented by the subgraph  $G_A = (V, E_A \cup \mathcal{C}) \subseteq G$  and Bob's input is represented by  $G_B = (V, E_B \cup \mathcal{C}) \subseteq G$ . (We denote this way of distributing the vertex and edge sets as the *vertex partition model*.) In order to learn any information about the structure of  $G[A] \setminus \mathcal{C}$  and  $G[B] \setminus \mathcal{C}$ , and hence about the input of the other player, Alice and Bob must communicate through the edges of the cut  $\mathcal{C}$ . Therefore, in order to obtain a lower bound for a problem in the  $\text{CONGEST}_b$  model, one wants to construct  $G$  to ensure that it has some property (in our case, contains a copy of  $K_\ell$ ) if and only if the corresponding instance of set disjointness is such that  $\text{DISJ}_n(X, Y) = 1$ , and in order to determine the required property, one has to communicate a large part of (essentially the entire graph)  $G[A]$  through  $\mathcal{C}$ . With this approach, if the cut  $\mathcal{C}$  has size  $|\mathcal{C}|$ , and the private inputs of Alice and Bob (edges in  $G[A] \setminus \mathcal{C}$  or  $G[B] \setminus \mathcal{C}$ ) are of size  $\mathfrak{s}$ , one can apply Theorem 1 to argue that the round complexity of any distributed algorithm in the  $\text{CONGEST}_b$  model for a given problem is  $\Omega(\frac{\mathfrak{s}}{|\mathcal{C}| \cdot b})$ . The central challenge is to ensure that for the encoded set disjointness instance of size  $\mathfrak{s}$  and the cut of size  $|\mathcal{C}|$ , the ratio  $\frac{\mathfrak{s}}{|\mathcal{C}|}$  is as large as possible.

For example, Drucker et al. [7] incorporated a similar approach to obtain a lower bound for the subgraph detection problem in a *broadcast* variant of the  $\text{CONGEST}_b$  model (in fact, even for a (stronger) broadcast variant of the **CONGESTED CLIQUE** model), where nodes are required to send the same message through all their incident edges. The lower bound construction requires sending  $\Omega(n^2)$  bits through the cut of size  $\mathcal{O}(n^2)$ , but the fact that in the broadcast variant of the  $\text{CONGEST}_b$  model every node is required to send the same message via all incident edges, at most  $\mathcal{O}(n \cdot b)$  bits can be transmitted through the cut, yielding a lower bound of  $\Omega(\frac{n}{b})$ . (In particular, for the *broadcast* variant of the  $\text{CONGEST}_b$  model, Drucker et al. [7, Theorem 15] proved that detecting a clique  $K_\ell$ ,  $\ell \geq 4$ , requires  $\Omega(\frac{n}{b})$  rounds.) Note however that in the (non-broadcast)  $\text{CONGEST}_b$  model, this construction does not give any non-trivial bound, since  $\frac{\mathfrak{s}}{|\mathcal{C}|} = \mathcal{O}(1)$ .

Our main building block for our lower bound is the construction of  $(\Omega(n^2), \mathcal{O}(n^{3/2}))$ -*lower-bound graphs* (see Section 3.1 for the precise definition) that can be used to encode a set disjointness instance of size  $\mathfrak{s} = \Omega(n^2)$  such that the cut is of size  $|\mathcal{C}| = \mathcal{O}(n^{3/2})$ . By

incorporating these bounds in the framework described above, this construction leads to the first non-trivial lower bound of  $\Omega(\frac{\sqrt{n}}{b})$  for the subgraph detection problem in the  $\text{CONGEST}_b$  model for the clique  $K_4$ . This construction can also be extended to detect larger cliques, yielding the lower bound of  $\Omega(\frac{n}{(\ell+\sqrt{n})b})$  for detecting any  $K_\ell$  with  $\ell \geq 4$ .

Since these are the first superconstant lower bounds for detecting a clique in the  $\text{CONGEST}$  model and since the best upper bound for these problems is still  $\mathcal{O}(n)$ , the next goal is to understand to what extent these bounds could be improved and whether the existing approach could be used for that task. Do we need  $\Omega(\frac{\sqrt{n}}{b})$  communication rounds to detect any clique  $K_\ell$  (with  $\ell \geq 4$ ,  $\ell = \mathcal{O}(\sqrt{n})$ ) in the  $\text{CONGEST}_b$  model, or maybe we need as many as a linear number of rounds? While we do not know the answer to this question, and in fact, this question is the main open problem left by this paper, we can prove that any better lower bound would require a significantly different approach, going beyond the two-party communication framework in the vertex partition model.

Indeed, let us consider the vertex partition model in the two-party communication framework, as defined above. The input consists of an undirected  $G = (V, E)$  with an arbitrary vertex partition  $V = V_A \dot{\cup} V_B$ . We consider a scenario where Alice is given the subgraph  $G_A = (V, E_A \cup \mathcal{C}) \subseteq G$  and Bob is given  $G_B = (V, E_B \cup \mathcal{C}) \subseteq G$ , where  $\mathcal{C}$  is the  $(V_A, V_B)$ -cut in  $G$ . The arguments in our construction of lower-bound graphs in Theorem 9 imply that for some inputs, any two-party communication protocol in the vertex partition model for the problem of listing all cliques in a given graph with  $n$  nodes requires communication of  $\Omega(\sqrt{n}|\mathcal{C}|)$  bits between Alice and Bob. We will prove in Section 4 (Theorem 10) that this lower bound is asymptotically tight in the two-party communication framework in the vertex partition model. We show that there is a two-party communication protocol in the vertex partition model for listing *all cliques* that uses  $\mathcal{O}(\sqrt{n}|\mathcal{C}|)$  communication rounds, where  $\mathcal{C}$  is the set of shared edges between Alice and Bob. This shows that we cannot obtain stronger lower bounds for the  $K_\ell$ -detection problem, for  $\ell = \mathcal{O}(\sqrt{n})$ , in the  $\text{CONGEST}$  model using the two-party communication framework in the vertex partition model.

### 1.3 Related works

As a fundamental primitive, subgraph detection and listing in the  $\text{CONGEST}$  model has been recently receiving attention from multiple authors, focusing mainly on randomized complexity. However, despite major efforts, for the  $\text{CONGEST}$  model, relatively little is known about the complexity of the subgraph detection problem.

Prior to our work, no non-trivial results about the complexity of clique  $K_\ell$  ( $\ell \geq 4$ ) detection in the  $\text{CONGEST}$  model have been known. While there is a trivial lower bound of a constant number of rounds, and as we mentioned earlier, one can easily solve the problem in  $\mathcal{O}(n)$  rounds in the  $\text{CONGEST}$  model, no sublinear upper bounds nor superconstant lower bounds have been known.

In a recent breakthrough in this area, Izumi and Le Gall [12] raised some hopes that maybe these problems could be solved in a sublinear number of rounds in the  $\text{CONGEST}$  model. They considered the subgraph detection problem for the smallest interesting subgraph  $H$ , the triangle  $K_3$ , and presented a very clever algorithm that detects a triangle in  $\tilde{\mathcal{O}}(n^{2/3})$  rounds. Further, they also showed that the related problem of finding all triangles (triangle listing) can be solved in  $\tilde{\mathcal{O}}(n^{3/4})$  rounds. Very recently, these results were improved by Chang et al. [5], who showed that both triangle detection and enumeration can be solved in  $\tilde{\mathcal{O}}(\sqrt{n})$  rounds in the  $\text{CONGEST}$  model. There is no non-trivial lower bound for the triangle detection problem, though it is known (cf. [12, 17]) that the more complex triangle listing problem requires  $\Omega(n^{1/3}/\log n)$  rounds, even in the  $\text{CONGESTED CLIQUE}$  model. It can also

be shown that the problem of listing all triangles such that each node  $v$  learns all triangles that it is part of is significantly harder than the general triangle listing problem and requires  $\Omega(n/\log n)$  rounds [12, Proposition 4.4]. While rather disappointingly, we do not know how to extend any of these upper bounds to other cliques  $K_\ell$  with  $\ell \geq 4$ , the previously mentioned works for triangle detection raise hope that detecting cliques  $K_\ell$  could potentially be solved in a sublinear number of rounds. In fact, even for  $K_3$ , we do not even know whether detecting a triangle  $K_3$  can be solved in a polylogarithmic or even a constant number of rounds in the CONGEST model (the lower bound of  $\Omega(n^{1/3}/\log n)$  rounds in the CONGESTED CLIQUE model (cf. [12, 17]) holds only for a more complex problem of detecting *all triangles*).

Even et al. [8] noted that the problem of detecting trees is significantly simpler and designed a randomized color-coding algorithm that detects any constant-size *tree* on  $\ell$  nodes in  $\mathcal{O}(\ell^\ell)$  rounds.

As for lower bounds for the subgraph detection problem in the CONGEST model, until very recently, the only hardness results known in the literature have been for cycles. For any fixed  $\ell \geq 4$ , there is a polynomial lower bound for detecting the  $\ell$ -cycle  $C_\ell$  in the CONGEST model [7], where it has been shown that detecting  $C_\ell$  requires  $\Omega(\text{ex}(n, C_\ell)/\log n)$  rounds, where  $\text{ex}(n, C_\ell)$  is the Turán number for cycles, that is, the largest possible number of edges in a  $C_\ell$ -free graph over  $n$  vertices. In particular, for odd-length cycles (of length 5 or more), the lower bound of [7] is  $\Omega(n/\log n)$ , and it is  $\Omega(\sqrt{n}/\log n)$  for  $\ell = 4$ . Very recently, Korhonen and Rybicki [15] improved the lower bound for all even-length cycles to  $\Omega(\sqrt{n}/\log n)$ . Further, Gonen and Oshman [11] extended these lower bounds for  $C_\ell$ -freeness to some related classes of graphs, though still with some cyclic underlying structure. (As mentioned above, we note that Drucker et al. [7] presented lower bounds for other graphs, but this was in a *broadcast* variant of the CONGESTED CLIQUE model, where nodes are required to send the same message on all their edges. In particular, for the broadcast variant of the CONGESTED CLIQUE model, Drucker et al. [7] proved that detecting a clique  $K_\ell$ ,  $\ell \geq 4$ , requires  $\Omega(n/\log n)$  rounds.)

The only lower bound for the subgraph detection problem for  $H$  significantly other than cycles, is a very recent work of Fischer et al. [9], who demonstrated that the subgraph detection problem is hard even for some subgraphs  $H$  of constant size. In particular, for any constant  $\ell \geq 2$ , there is a graph  $H$  with a constant number of vertices and edges such that the problem of finding  $H$  in a network of size  $n$  requires time  $\Omega(n^{2-\frac{1}{\ell}}/\mathfrak{b})$  in the CONGEST model, where  $\mathfrak{b}$  is the bandwidth of each communication links.

There has also been some recent research for the *deterministic* subgraph detection problem in the CONGEST model. For example, Drucker et al. [7] designed an  $\mathcal{O}(\sqrt{n})$  round algorithm for  $C_4$  detection, and Even et al. [8] and Korhonen and Rybicki [15] obtained path and tree detection algorithms requiring only a constant number of rounds. Korhonen and Rybicki [15] considered also deterministic subgraph detection (for paths, cycles, trees, pseudotrees, and on  $d$ -degenerate graphs) in the weaker broadcast CONGEST model, where nodes send the same message to all neighbors in each communication round. In the CONGESTED CLIQUE model, deterministic subgraph detection algorithms were given by Dolev et al. [6] and Censor-Hillel et al. [3].

We summarize earlier results together with our new results in Table 1.

### 1.3.1 Property testing of $H$ -freeness

Since there have been so few positive results for the original subgraph detection problem, recently there have been some advances in a relaxation of this problem, a closely related (and significantly simpler) problem of *testing subgraphs freeness* in the *framework of property*

## 16:6 Detecting Cliques in CONGEST Networks

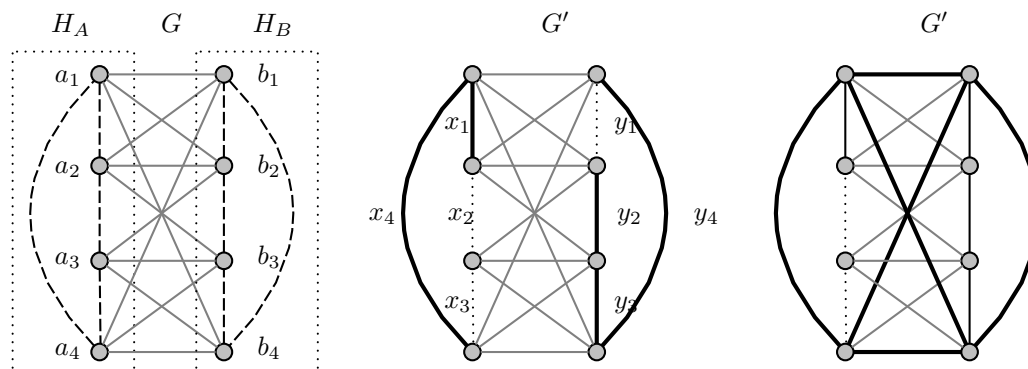
■ **Table 1** Prior (randomized) results for the problem of detecting a given subgraph  $H$ , or for listing all copies of  $H$ , in the CONGEST model (less relevant results (upper bounds) for the CONGESTED CLIQUE model are omitted; note that lower bounds for CONGESTED CLIQUE hold also for CONGEST and lower bounds for broadcast CONGESTED CLIQUE, abbreviated by br. CONGESTED CLIQUE in the table, do not imply any bounds for CONGEST).

Paper	Time bound	Problem	Model
[8]	$\mathcal{O}(\ell^\ell)$	Detecting a tree on $\ell$ nodes	CONGEST
folklore	$\mathcal{O}(n)$	Detecting $K_\ell$ , $\ell \geq 3$	CONGEST
[5]	$\tilde{\mathcal{O}}(\sqrt{n})$	Detecting triangle $K_3$	CONGEST
[5]	$\tilde{\mathcal{O}}(\sqrt{n})$	Triangle listing	CONGEST
[9]	$\Omega(n^{2-\frac{1}{\ell}}/\log n)$	Detecting some $H$ of size $\mathcal{O}(\ell)$	CONGEST
[7]	$\Omega(n/\log n)$	Detecting $C_\ell$ , $\ell \geq 5$ , $\ell$ odd	CONGEST
[7, 15]	$\Omega(\sqrt{n}/\log n)$	Detecting $C_\ell$ , $\ell \geq 4$ , $\ell$ even	CONGEST
[12, 17]	$\Omega(n^{1/3}/\text{poly-log}(n))$	Triangle listing	CONGESTED CLIQUE
[7]	$\Omega(n/\log n)$	Detecting $K_\ell$ for $\ell \geq 4$	br. CONGESTED CLIQUE
Thm. 4	$\Omega(\sqrt{n}/\log n)$	Detecting $K_4$	CONGEST
Thm. 5	$\Omega(\sqrt{n}/(\ell \log n))$	Detecting $K_\ell$ for $\ell \geq 4$	CONGEST

testing for distributed computations (see, e.g., [1, 8]). In the property testing setting, an algorithm has to decide, with probability at least  $\frac{2}{3}$ , if the input graph is (a)  $H$ -free (i.e., does not contain a subgraph isomorphic to  $H$ ) or (b)  $\varepsilon$ -far from being  $H$ -free (that is, the goal is to distinguish whether the input graph  $G$  is  $H$ -free or one needs to modify more than  $\varepsilon|E(G)|$  edges of  $G$  to obtain a graph that is  $H$ -free); in the intermediate case, the algorithm can perform arbitrarily (see e.g., [3, 8] for more details). Property testing of  $H$ -freeness in the CONGEST model has received a lot of attention lately (see, e.g., [1, 2, 8, 9, 10]). In particular, it has been shown [8] that testing  $H$ -freeness can be done in  $\mathcal{O}(1/\varepsilon)$  round in the CONGEST model for any constant-size graph  $H$  containing an edge  $(x, y)$  such that any cycle in  $H$  contains at least one of  $x, y$ . This implies testing in  $\mathcal{O}(1/\varepsilon)$  rounds of any cycle  $C_k$ , and of any subgraph  $H$  on five (or less) vertices except  $K_5$ . Further, for any  $\ell \geq 5$ ,  $K_\ell$ -freeness can be tested in  $\mathcal{O}((\varepsilon \cdot |E(G)|)^{\frac{1}{2} - \frac{1}{\ell-2}}/\varepsilon)$  rounds [8]. For trees, Even et al. [8] show that testing if the input graph is  $T$ -free for a tree  $T$  on  $\ell$  vertices can be done in  $\mathcal{O}(\ell^{1+\ell^2}/\varepsilon^\ell)$  rounds the CONGEST model.

### 2 Lower bound results: Detecting a clique requires $\tilde{\Omega}(\sqrt{n})$ rounds

In this section we prove our hardness results showing that any algorithm in the CONGEST<sub>b</sub> model that detects a  $K_\ell$  with probability at least  $\frac{2}{3}$  requires  $\Omega(\sqrt{n}/b)$  rounds, for every  $\ell = \mathcal{O}(\sqrt{n})$  and  $\ell \geq 4$ , and requires  $\Omega(\frac{n}{\ell b})$  rounds if  $\ell = \omega(\sqrt{n})$  (Theorems 4 and 5); or in short,  $\Omega(\frac{n}{(\ell + \sqrt{n})b})$  rounds, for every  $\ell \geq 4$ . Our lower bound for the complexity of detecting  $K_\ell$  in the CONGEST model relies on a reduction to the two-party communication complexity lower bound for the set disjointness problem (cf. Theorem 1 in Section 1.2), which we implement with the help of lower-bound graphs (cf. Section 2.1).



■ **Figure 1 Left:** Example of a  $(4, 12)$ -lower-bound graph  $G = (A, B, E)$ . The dotted edges are the edges of the associated graphs  $H_A$  and  $H_B$  (observe that  $H_A$  and  $H_B$  form cycles of length 4, which are bipartite). For  $1 \leq i \leq 4$ , let  $\mathcal{E}_i$  be the edge set of subgraph  $G[\{a_i, a_{(i \bmod 4)+1}, b_i, b_{(i \bmod 4)+1}\}]$ . Observe that  $E = \bigcup_{i \leq 4} \mathcal{E}_i$ , and, for every  $i$ ,  $G[\mathcal{E}_i]$  is isomorphic to  $K_{2,2}$ . Observe further that for  $i \neq j$ ,  $G[A(\mathcal{E}_i) \cup B(\mathcal{E}_j)]$  is not isomorphic to  $K_{2,2}$ . **Center:** Graph  $G'$  as in the proof of Theorem 3 obtained from the set disjointness instance with  $X = (1, 0, 0, 1)$  and  $Y = (0, 1, 1, 1)$ . Graph  $G'$  contains a  $K_4$  if and only if the set disjointness instance evaluates to 1. **Right:** The highlighted edges form a  $K_4$ .

## 2.1 Lower-bound graphs

Our reduction to the two-party communication complexity lower bound for the set disjointness problem relies on a notion of a *lower-bound graph* (cf. Figure 1).

► **Definition 2.** Let  $G = (A, B, E)$  be a bipartite graph with  $|A| = |B| = n$  and let  $k, m$  be integers. Then  $G$  is called a  $(k, m)$ -lower-bound graph if:

1.  $|E| \leq m$ .
2. The edge set  $E$  is the union of (not necessarily disjoint) sets  $\mathcal{E}_1, \mathcal{E}_2, \dots, \mathcal{E}_k$  such that, for every  $i$ ,  $1 \leq i \leq k$ , the edge-induced subgraph  $G[\mathcal{E}_i]$  is isomorphic to  $K_{2,2}$ .
3. For every  $i, j$ ,  $1 \leq i, j \leq k$ ,  $i \neq j$ , the vertex-induced subgraph  $G[A(\mathcal{E}_i) \cup B(\mathcal{E}_j)]$  is not isomorphic to  $K_{2,2}$  (For a set of edges  $E' \subseteq E$  we denote the set of incident  $A$ -vertices by  $A(E')$ . The set  $B(E')$  is defined similarly.).
4. Define two graphs associated with  $G$ ,  $H_A = (A, E_A)$  and  $H_B = (B, E_B)$ .  $H_A$  is the graph on vertex set  $A$ , where  $a_1, a_2 \in A$  are adjacent if and only if there exists an index  $i$  with  $A(\mathcal{E}_i) = \{a_1, a_2\}$ . Similarly,  $H_B$  is the graph on vertex set  $B$ , where  $b_1, b_2 \in B$  are adjacent if and only if there exists an index  $j$  with  $B(\mathcal{E}_j) = \{b_1, b_2\}$ . Then, we require that  $H_A$  and  $H_B$  are bipartite.

## 2.2 Using lower-bound graphs and set disjointness to prove the hardness of clique detection

With the notion of lower-bound graphs at hand, we can formalize our reduction to the two-party communication complexity lower bound for set disjointness to obtain the following central theorem.

► **Theorem 3.** Let  $G$  be a  $(k, m)$ -lower-bound graph. Then, detecting a  $K_4$  in the  $\text{CONGEST}_b$  model with probability at least  $\frac{2}{3}$  requires  $\Omega\left(\frac{k}{mb}\right)$  rounds.

**Proof.** Let  $\mathcal{A}$  be an algorithm in the  $\text{CONGEST}_b$  model for  $K_4$  detection, that is, such that with probability at least  $\frac{2}{3}$ , if  $G$  contains a  $K_4$  then at least one node outputs 1 and if  $G$

contains no copy of  $K_4$  then no node outputs 1. We will show that  $\mathcal{A}$  can be used to solve the two-party set disjointness problem for instances of size  $k$ .

Consider a set disjointness instance  $(X, Y)$  of size  $k$ . Let  $G = (A, B, E)$  be a  $(k, m)$ -lower-bound graph, let  $\mathcal{E}_1, \mathcal{E}_2, \dots, \mathcal{E}_k$  be the edge partition as in Item 2 of Definition 2, and let  $H_A = (A, E_A)$  and  $H_B = (B, E_B)$  be the graphs associated with  $G$  (Item 4 in Definition 2). Alice constructs the set  $E'_A \subseteq E_A$  such that for every  $i$  with  $X_i = 1$ , the edge between  $A(\mathcal{E}_i)$  is included in  $E'_A$ . Similarly, Bob constructs the set  $E'_B \subseteq E_B$  such that for every  $i$  with  $Y_i = 1$ , the edge between  $B(\mathcal{E}_i)$  is included in  $E'_B$ .

We first show that the graph  $G' := G \cup (E'_A \cup E'_B)$  contains a  $K_4$  if and only if  $\text{DISJ}_n(X, Y) = 1$ . Indeed, since by Item 4 of Definition 2, the graphs  $H_A$  and  $H_B$  are bipartite (and thus the subgraphs  $G'[A]$  and  $G'[B]$  are bipartite too), any copy of  $K_4$  in  $G'$  must consist of two vertices from  $A$  and two vertices from  $B$ . Let  $a_1, a_2$  be any pair of distinct vertices in  $A$  and  $b_1, b_2$  be any pair of distinct vertices in  $B$ . Observe that if there is no  $\mathcal{E}_i$  such that  $\{a_1, a_2\} = A(\mathcal{E}_i)$  or there is no  $\mathcal{E}_i$  such that  $\{b_1, b_2\} = B(\mathcal{E}_i)$  then it is impossible for the nodes  $a_1, a_2, b_1, b_2$  to form a  $K_4$ , since this would imply that either  $a_1 a_2 \notin E'_A$  or  $b_1 b_2 \notin E'_B$ . Assume therefore that  $\{a_1, a_2\} = A(\mathcal{E}_i)$  and  $\{b_1, b_2\} = B(\mathcal{E}_j)$ , for some  $i, j$ . Next, suppose that  $i \neq j$ . Then  $G[\{a_1, a_2, b_1, b_2\}]$  is not isomorphic to  $K_{2,2}$ , by Item 3 of Definition 2. Hence, assume that  $i = j$ . Then  $G[\{a_1, a_2, b_1, b_2\}]$  forms a  $K_{2,2}$  if and only if  $X_i = Y_i = 1$ , which in turn implies  $\text{DISJ}_n(X, Y) = 1$ .

The simulation of  $\mathcal{A}$  on  $G'$  is executed as follows. Suppose that  $\mathcal{A}$  runs in  $r$  rounds. Alice simulates vertices  $A$  and Bob simulates vertices  $B$ . In round  $i$ , Alice sends all messages from  $A$  with destinations in  $B$  to Bob, and Bob sends all messages from  $B$  with destinations in  $A$  to Alice. Since the cut between  $A$  and  $B$  is of size  $m$ , Alice and Bob exchange messages with overall  $m\mathfrak{b}$  bits per round. Thus, overall they communicate  $rm\mathfrak{b}$  bits. Since the algorithm allows them to solve set disjointness, by Theorem 1, we have  $rm\mathfrak{b} = \Omega(k)$ . Thus,  $\mathcal{A}$  requires  $\Omega(\frac{k}{m\mathfrak{b}})$  rounds.  $\blacktriangleleft$

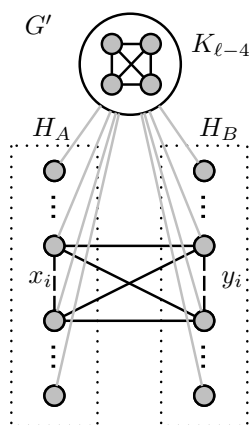
In Theorem 9 in Section 3, we prove the existence of a  $(\Omega(n^2), \mathcal{O}(n^{3/2}))$ -lower-bound graph. By combining Theorem 9 with Theorem 3, we obtain the following main result.

► **Theorem 4.** *Every algorithm in the  $\text{CONGEST}_{\mathfrak{b}}$  model that detects a  $K_4$  with probability at least  $\frac{2}{3}$  requires  $\Omega(\sqrt{n}/\mathfrak{b})$  rounds.*

### 2.3 Detection of $K_\ell$ for $\ell \geq 5$

The lower bound construction given in Theorem 3 can be extended to the task of detecting  $K_\ell$ , for  $\ell \geq 5$  (see also Figure 2). To this end, we add a clique on  $\ell - 4$  new nodes to graph  $G'$  (from the proof of Theorem 3) and connect each of these nodes to every vertex in  $A \cup B$ . Observe that this increases the cut between  $A$  and  $B$  by  $n(\ell - 4)$  edges. For  $\ell = \mathcal{O}(\sqrt{n})$ , there are only  $\mathcal{O}(n^{3/2})$  additional edges, which implies that the same lower bound as for  $K_4$  holds. If  $\ell = \omega(\sqrt{n})$ , then the number of additional edges is significant, since the size of the cut increases by more than a constant factor. In this case, the round complexity is  $\Omega(\frac{n^2}{n(\ell-4)\mathfrak{b}}) = \Omega(\frac{n}{\ell\mathfrak{b}})$ . Similarly as before, the encoded set disjointness instance evaluates to 1 if and only if  $G'$  contains a clique of size  $\ell$ . We thus conclude with the following theorem.

► **Theorem 5.** *Every algorithm in the  $\text{CONGEST}_{\mathfrak{b}}$  model that detects  $K_\ell$ , for  $\ell \geq 4$  and  $\ell = \mathcal{O}(\sqrt{n})$ , with probability at least  $\frac{2}{3}$  requires  $\Omega(\sqrt{n}/\mathfrak{b})$  rounds. If  $\ell = \omega(\sqrt{n})$ , then  $\Omega(n/(\ell\mathfrak{b}))$  rounds are required.*



■ **Figure 2** Extension of our lower bound for  $K_4$  detection to  $K_\ell$  detection, for  $\ell \geq 5$ . We add a clique  $K_{\ell-4}$  on  $\ell - 4$  new vertices to the graph  $G'$  and connect every vertex of the clique to every other vertex of  $G'$ . Then the resulting graph contains a clique on  $\ell$  vertices if and only if the encoded set disjointness instance evaluates to 1, i.e.,  $x_i = y_i = 1$ , for some  $i$ .

### 3 Lower-bound graph construction

In this section, we prove the existence of a  $(\Omega(n^2), \mathcal{O}(n^{3/2}))$ -lower-bound graph (see Definition 2), which is our main technical tool. We will show in Theorem 9 that Algorithm 1 below constructs a  $(\Omega(n^2), \mathcal{O}(n^{3/2}))$ -lower-bound graph with high probability (observe that a non-zero probability already suffices to prove the existence of such a graph).

#### 3.1 Construction of a $(\Omega(n^2), \mathcal{O}(n^{3/2}))$ -lower-bound graph

We proceed as follows. We start our construction with a bipartite random graph  $G = (A, B, E)$  with  $|A| = |B| = n$ , where every potential edge  $ab$  between  $a \in A$  and  $b \in B$  is included with probability  $p = \frac{1}{\sqrt{n}}$ . Observe that for any  $a_1, a_2 \in A$  ( $a_1 \neq a_2$ ) and  $b_1, b_2 \in B$  ( $b_1 \neq b_2$ ), the probability that  $G[\{a_1, a_2, b_1, b_2\}]$  is isomorphic to a  $K_{2,2}$  is  $p^4$ . We therefore expect  $G$  to contain  $\binom{n}{2}^2 p^4$  copies of  $K_{2,2}$ , and we prove in Lemma 6 below that, with high probability, the actual number of copies of  $K_{2,2}$  does not deviate significantly from its expectation. Let  $\mathcal{K}$  denote the set of copies of  $K_{2,2}$  in  $G$ .

In the peeling phase, we greedily compute a subset  $\mathcal{H} \subseteq \mathcal{K}$  such that at the end, the graph induced by the edges of  $\mathcal{H}$  is a  $(\Omega(n^2), \mathcal{O}(n^{3/2}))$ -lower bound graph. When inserting a set  $K = \{a_1, a_2, b_1, b_2\} \in \mathcal{K}$  into  $\mathcal{H}$ , we make sure that the following three properties are fulfilled:

1. We ensure that later on we will never add a  $K' = \{a'_1, a'_2, b'_1, b'_2\}$  such that either  $\{a_1, a_2, b'_1, b'_2\}$  or  $\{a'_1, a'_2, b_1, b_2\}$  form a  $K_{2,2}$ . To this end, when inserting  $K$  into  $\mathcal{H}$ , for every  $K' \in \mathcal{K}$  that contains the same pair of  $A$ -vertices (or  $B$ -vertices), we add its pair of  $B$  vertices (resp. pair of  $A$  vertices) to set  $F_B$  (resp.  $F_A$ ), indicating that this is a forbidden pair. Then, when inserting an element of  $\mathcal{K}$  into  $\mathcal{H}$ , we make sure that its pairs of  $A$  and  $B$  vertices are not forbidden.
2. We make sure that the insertion of  $K$  will not prevent too many other sets  $K'$  from being inserted into  $\mathcal{H}$ . To this end, we guarantee that there are at most six other sets in  $\mathcal{K}$  that share the same pair of  $A$  vertices and at most six other sets that share the same pair of  $B$  vertices. We prove in Lemma 7 that most  $K \in \mathcal{K}$  fulfill this property.



---

**Algorithm 1** Construction of a  $(\Omega(n^2), \mathcal{O}(n^{3/2}))$ -lower-bound graph:
 

---

**Input:** Integer  $n$ , let  $p = \frac{1}{\sqrt{n}}$ .

**1. Random Graph:**

 Let  $G = (A, B, E)$  with  $|A| = |B| = n$  be the bipartite random graph where  
 for every  $a \in A, b \in B$  the edge  $ab$  is included in  $E$  with probability  $p$ .

 Let  $\mathcal{K}$  be the family of sets  $\{a_1, a_2, b_1, b_2\}$  with  $a_1, a_2 \in A, a_1 \neq a_2, b_1, b_2 \in B, b_1 \neq b_2$   
 and  $G[\{a_1, a_2, b_1, b_2\}]$  isomorphic to  $K_{2,2}$ .

 For  $S \subseteq A \cup B$ , let  $\mathcal{K}(S) := \{K \in \mathcal{K} : S \subseteq K\}$ .

**2. Peeling Process:**

 Let  $A' \subseteq A$  and  $B' \subseteq B$  be a uniform random sample of  $A$  and  $B$ , respectively,  
 where every vertex is included with probability  $\frac{1}{2}$ .

 $\mathcal{H} \leftarrow \{\}, F_A \leftarrow \{\}, F_B \leftarrow \{\}$ .

**for** every  $K = \{a_1, a_2, b_1, b_2\} \in \mathcal{K}$  **do**
**if**  $|\mathcal{K}(\{a_1, a_2\})| \leq 6$  and  $|\mathcal{K}(\{b_1, b_2\})| \leq 6$  and  $|\{a_1, a_2\} \cap A'| = |\{b_1, b_2\} \cap B'| = 1$  and  
 $\{a_1, a_2\} \notin F_A$  and  $\{b_1, b_2\} \notin F_B$  **then**
 $\mathcal{H} \leftarrow \mathcal{H} \cup K$ .

 For every  $\{a_1, a_2, b_3, b_4\} \in \mathcal{K}(\{a_1, a_2\})$ , add  $\{b_3, b_4\}$  to  $F_B$ .

 For every  $\{a_3, a_4, b_1, b_2\} \in \mathcal{K}(\{b_1, b_2\})$ , add  $\{a_3, a_4\}$  to  $F_A$ .

**end if**
**end for**
**3. Lower Bound Graph  $H$ :**

 For  $K = \{a_1, a_2, b_1, b_2\} \in \mathcal{H}$ , let  $E_K$  be the edge set  $\{a_1b_1, a_1b_2, a_2b_1, a_2b_2\}$ .

**return**  $H := (A, B, \bigcup_{K \in \mathcal{H}} E_K)$ .
 

---

3. It is required that the graphs  $G_A$  and  $G_B$  as defined in Item 4 of Definition 2 are bipartite. We therefore partition the sets  $A$  and  $B$  randomly into subsets  $A'$  and  $A \setminus A'$ , and  $B'$  and  $B \setminus B'$ , and only add  $K$  to  $\mathcal{H}$  if exactly one of its  $A$  vertices is in  $A'$  and one of its  $B$  vertices is in  $B'$ .

In the last step of the algorithm, we assemble graph  $H$  as the union of the edges contained in the copies of  $K_{2,2}$  in  $\mathcal{H}$ .

### 3.2 Analysis of Algorithm 1

Our analysis relies on some basic properties of the structure of subgraphs of random graphs (for a more complete treatment of related problems, see, e.g., [13, Chapter 3]). We prove three high probability claims about the construction in Algorithm 1: that the random graph  $G$  contains many copies of  $K_{2,2}$  (Lemma 6), that only a small fraction of pairs of  $A$  vertices are contained in more than six copies of  $K_{2,2}$  (Lemma 7), and finally that the resulting graph  $H$  contains  $\Omega(n^2)$  copies of  $K_{2,2}$  (Lemma 8). With these three claims at hand, we will complete the analysis to prove in Theorem 9 that with high probability, the output of Algorithm 1 is a  $(\Omega(n^2), \mathcal{O}(n^{3/2}))$ -lower-bound graph.

We begin with a proof that in Algorithm 1, the random graph  $G$  contains many copies of  $K_{2,2}$ .



► **Lemma 6.** *Suppose that  $p \geq \frac{1}{n}$ . Then there is a constant  $C$  such that*

$$\mathbb{P} \left[ |\mathcal{K}| \leq \frac{9}{10} \binom{n}{2} p^4 \right] \leq C \cdot \frac{1}{n^2 p} .$$

**Proof.** We will compute the expectation and the variance of  $|\mathcal{K}|$  and then use Chebyshev's inequality to bound the probability that  $|\mathcal{K}|$  deviates substantially from its expectation.

Let  $\mathcal{X}$  be the family of all sets  $\{a_1, a_2, b_1, b_2\}$  with  $a_1, a_2 \in A$ ,  $a_1 \neq a_2$ ,  $b_1, b_2 \in B$ ,  $b_1 \neq b_2$ , and for  $X \in \mathcal{X}$  let  $\chi(X)$  be the indicator variable of the event “ $G[X]$  is isomorphic to  $K_{2,2}$ ”. Then:

$$\mathbb{E}|\mathcal{K}| = \sum_{X \in \mathcal{X}} \mathbb{P}[\chi(X) = 1] = |\mathcal{X}|p^4 = \binom{n}{2}^2 p^4 ,$$

since  $K_{2,2}$  contains 4 edges. To bound the variance  $\mathbb{V}|\mathcal{K}|$ , we use the identity  $\mathbb{V}|\mathcal{K}| = \mathbb{E}|\mathcal{K}|^2 - (\mathbb{E}|\mathcal{K}|)^2$ :

$$\mathbb{E}|\mathcal{K}|^2 = \mathbb{E} \left( \sum_{X \in \mathcal{X}} \chi(X) \right)^2 = \mathbb{E} \sum_{X, Y \in \mathcal{X}} \chi(X) \cdot \chi(Y) = \sum_{X, Y \in \mathcal{X}} \mathbb{E}(\chi(X) \cdot \chi(Y)) .$$

We distinguish the following cases:

- $|X \cap Y| = 0$ . Then,  $\mathbb{E}(\chi(X) \cdot \chi(Y)) = p^8$ . Observe that there are  $t_0 = \binom{n}{2}^2 \binom{n-2}{2}^2$  such pairs.
- $|X \cap Y| = 1$ . Then,  $\mathbb{E}(\chi(X) \cdot \chi(Y)) = p^8$ . There are  $t_1 = 4 \binom{n}{2}^2 \binom{n-2}{2} \binom{n-2}{1}$  such pairs.
- $|X \cap Y| = 2$  and the intersection consists of either two  $A$ -vertices or two  $B$ -vertices. Then,  $\mathbb{E}(\chi(X) \cdot \chi(Y)) = p^8$  and there are  $t_{2,1} = 2 \cdot \binom{n}{2}^2 \binom{n-2}{2}$  such pairs.
- $|X \cap Y| = 2$  and the intersection consists of one  $A$ -vertex and one  $B$ -vertex. Then,  $\mathbb{E}(\chi(X) \cdot \chi(Y)) = p^7$  and there are  $t_{2,2} = 4 \cdot \binom{n}{2}^2 \cdot (n-2)^2$  such pairs.
- $|X \cap Y| = 3$ . Then,  $\mathbb{E}(\chi(X) \cdot \chi(Y)) = p^6$ . There are  $t_3 = 4 \cdot \binom{n}{2}^2 \cdot (n-2)$  such pairs.
- $|X \cap Y| = 4$ . Then,  $\mathbb{E}(\chi(X) \cdot \chi(Y)) = p^4$ . There are  $t_4 = \binom{n}{2}^2$  such pairs.

A quick sanity check shows that  $t_0 + t_1 + t_{2,1} + t_{2,2} + t_3 + t_4 = \binom{n}{2}^4$ . We thus obtain:

$$\begin{aligned} \mathbb{V}|\mathcal{K}| &= \mathbb{E}|\mathcal{K}|^2 - (\mathbb{E}|\mathcal{K}|)^2 = p^8(t_0 + t_1 + t_{2,1}) + p^7 t_{2,2} + p^6 t_3 + p^4 t_4 - \binom{n}{2}^4 p^8 \\ &\leq p^7 t_{2,2} + p^6 t_3 + p^4 t_4 = \mathcal{O}(p^7 n^6) , \end{aligned}$$

where the last equality holds for every  $p \geq \frac{1}{n}$ . We apply Chebyshev's inequality and obtain:

$$\mathbb{P} \left[ \left| |\mathcal{K}| - \mathbb{E}|\mathcal{K}| \right| \geq \frac{1}{10} \mathbb{E}|\mathcal{K}| \right] \leq \frac{100 \mathbb{V}|\mathcal{K}|}{(\mathbb{E}|\mathcal{K}|)^2} = C \cdot \frac{1}{n^2 p} ,$$

for some constant  $C$ . ◀

Next, we prove that only a small fraction of pairs of  $A$  vertices are contained in more than six copies of  $K_{2,2}$ .

► **Lemma 7.** *Let  $p = \frac{1}{\sqrt{n}}$ . For every constant  $\delta > 0$ , with high probability, there are at most  $(1 + \delta)n^2/10$  pairs of distinct vertices  $a_1, a_2 \in A$  with  $|\mathcal{K}(\{a_1, a_2\})| > 6$ .*

16:12 Detecting Cliques in CONGEST Networks

**Proof.** Let  $a_1, a_2 \in A$ ,  $a_1 \neq a_2$  be arbitrary vertices. Let  $B(\{a_1, a_2\}) \subseteq B$  be the set of vertices  $b$  such that  $a_1b, a_2b \in E$ . Observe that  $|\mathcal{K}(\{a_1, a_2\})| = \binom{|B(\{a_1, a_2\})|}{2}$ . By linearity of expectation,  $\mathbb{E}|B(\{a_1, a_2\})| = np^2 = 1$ .

Let  $\mathcal{X}$  be the family of all sets of vertices  $\{a_1, a_2\} \subseteq A$  with  $a_1 \neq a_2$ . Partition now  $\mathcal{X}$  into disjoint subsets such that  $\mathcal{X} = \mathcal{X}_1 \cup \mathcal{X}_2 \cup \dots \cup \mathcal{X}_{n-1}$ , where  $|\mathcal{X}_i| = n/2$  and, for every  $1 \leq i \leq n-1$ , all elements of  $\mathcal{X}_i$  are pairwise disjoint (such a partitioning corresponds to partitioning the complete graph  $K_n$  into  $n-1$  perfect matchings). For a pair of vertices  $P \in \mathcal{X}$ , let  $\chi(P)$  be the indicator variable of the event “ $|B(P)| \geq 5$ ”. Recall that  $\mathbb{E}|B(P)| = np^2 = 1$  (since  $p = 1/\sqrt{n}$ ). Hence, by Markov’s inequality, we have  $\mathbb{P}[\chi(P) = 1] \leq \frac{1}{5}$ .

For every  $1 \leq i \leq n-1$  we have  $\mathbb{E} \sum_{P \in \mathcal{X}_i} \chi(P) \leq \frac{1}{5} \frac{n}{2} = \frac{n}{10}$ . Observe further that for every  $P, Q \in \mathcal{X}_i$ ,  $P \neq Q$ , the random variables  $B(P)$  and  $B(Q)$  are independent. Thus, by a Chernoff bound (for  $\mu = \frac{n}{10}$ ):

$$\mathbb{P} \left[ \left| \sum_{S \in \mathcal{X}_i} \chi(S) - \mu \right| \geq \delta \mu \right] \leq 2 \exp(-\mu \delta^2 / 3) = e^{-\Theta(n)} ,$$

for any constant  $\delta$ . Thus, applying the union bound for every  $1 \leq i \leq n-1$ , with high probability, at most  $(1 + \delta) \frac{n}{10} \cdot (n-1) \leq (1 + \delta)n^2/10$  pairs of vertices are both connected to at least 5 vertices of  $B$ . Hence, at most  $(1 + \delta)n^2/10$  pairs of vertices  $\{a_1, a_2\}$  are such that  $\mathcal{K}(\{a_1, a_2\}) > \binom{4}{2} = 6$ . ◀

In the next lemma, we show that our resulting graph  $H$  contains  $\Omega(n^2)$  copies of  $K_{2,2}$ .

► **Lemma 8.** *With high probability, the number of copies of  $K_{2,2}$  in  $H$  is  $|\mathcal{H}| = \Omega(n^2)$ .*

**Proof.** By Lemma 6, we have  $|\mathcal{K}| \geq \frac{9}{40}(n-1)^2$  with high probability. Let  $\mathcal{K}' \subseteq \mathcal{K}$  be the subset of sets  $\{a_1, a_2, b_1, b_2\}$  with  $\mathcal{K}(\{a_1, a_2\}) \leq 6$  and  $\mathcal{K}(\{b_1, b_2\}) \leq 6$ . By Lemma 7, with high probability,  $|\mathcal{K}'| \geq |\mathcal{K}| - 2 \cdot (1 + \delta)n^2/10$ , for any small constant  $\delta$ .

Let  $\mathcal{K}'' \subseteq \mathcal{K}'$  be the subset of sets  $\{a_1, a_2, b_1, b_2\}$  with  $|\{a_1, a_2\} \cap A'| = |\{b_1, b_2\} \cap B'| = 1$ . Observe that every set  $X \in \mathcal{K}'$  is included in  $\mathcal{K}''$  with probability  $\frac{1}{4}$ . Thus, by a Chernoff bound,  $|\mathcal{K}''| \geq |\mathcal{K}'|/8$  with high probability.

We argue next that the insertion of any set  $K \in \mathcal{K}'$  can block at most  $2 \cdot 6^2 = 72$  other sets of  $\mathcal{K}'$  from being inserted into  $\mathcal{H}$ . Consider thus a set  $K = \{a_1, a_2, b_1, b_2\} \in \mathcal{K}'$  that is added to  $\mathcal{H}$ . This inserts at most six pairs  $\{a_3, a_4\}$  into  $F_A$  and six pairs  $\{b_3, b_4\}$  into  $F_B$ , since  $\mathcal{K}(\{a_1, a_2\}) \leq 6$  and  $\mathcal{K}(\{b_1, b_2\}) \leq 6$ . Since each pair in  $F_A$  or in  $F_B$  can block at most another six sets of  $\mathcal{K}'$ , overall at most  $2 \cdot 6^2 = 72$  sets of  $\mathcal{K}'$  can be blocked by the insertion of  $K$  into  $\mathcal{H}$ .

Hence:

$$|\mathcal{H}| \geq \frac{|\mathcal{K}''|}{72} \geq \frac{|\mathcal{K}'|}{8 \cdot 72} \geq \frac{(|\mathcal{K}| - 2 \cdot (1 + \delta)n^2/10)}{8 \cdot 72} \geq \frac{(\frac{9}{40}(n-1)^2 - (1 + \delta)n^2/5)}{8 \cdot 72} = \Omega(n^2) ,$$

for  $\delta < \frac{1}{8}$ . ◀

With Lemmas 6–8 at hand, we are now ready to complete the analysis and show that the graph  $H$  fulfills Definition 2 of a lower bound graph.

► **Theorem 9.** *With high probability, the output of Algorithm 1 is a  $(\Omega(n^2), \mathcal{O}(n^{3/2}))$ -lower-bound graph. In particular, for every natural  $n$ , there exists a  $(\Omega(n^2), \mathcal{O}(n^{3/2}))$ -lower-bound graph.*

**Proof.** We need to check that all items of Definition 2 are fulfilled with  $p = \frac{1}{\sqrt{n}}$ . Concerning Item 1, observe that graph  $G$  has  $\mathcal{O}(n^2p) = \mathcal{O}(n^{3/2})$  edges with high probability (by a Chernoff bound).

For each  $K \in \mathcal{H}$ , let  $E_K$  denote the edge set added to graph  $H$  as in Step 3 of the algorithm. Item 2 holds, since  $E(H) = \bigcup_{K \in \mathcal{H}} E_K$ , and  $H[E_K]$  is isomorphic to  $K_{2,2}$ , for every  $K$ , and by Lemma 8.

Concerning Item 3, observe that when  $K = \{a_1, a_2, b_1, b_2\}$  is inserted into  $\mathcal{H}$ , then every  $\{a_1, a_2, b_3, b_4\}$  such that  $G[\{a_1, a_2, b_3, b_4\}]$  is isomorphic to  $K_{2,2}$  will not be inserted at a later stage, since  $\{b_3, b_4\}$  is inserted into  $F_B$ . For the same reason, every  $\{a_3, a_4, b_1, b_2\}$  such that  $G[\{a_3, a_4, b_1, b_2\}]$  is isomorphic to  $K_{2,2}$  will not be inserted into  $\mathcal{H}$ . This proves Item 3.

Concerning Item 4, observe that for every  $\{a_1, a_2, b_1, b_2\}$  that is included in  $\mathcal{H}$ , we have  $|\{a_1, a_2\} \cap A'| = |\{b_1, b_2\} \cap B'| = 1$ . Hence,  $H_A$  and  $H_B$  as defined in Item 4 are bipartite. ◀

## 4 Two-party communication protocol for listing all cliques

We consider a two-party communication protocol in the vertex partition model for listing all cliques (of all sizes) in a given graph. The input consists of an undirected graph  $G = (V, E)$  with an arbitrary vertex partition  $V = V_A \dot{\cup} V_B$ . Let  $\mathcal{C}$  be the  $(V_A, V_B)$ -cut,  $E_A$  be the edge set of  $G[V_A]$ , and  $E_B$  be the edge set of  $G[V_B]$ . We consider a scenario where Alice is given the subgraph  $G_A = (V, E_A \cup \mathcal{C}) \subseteq G$  and Bob is given  $G_B = (V, E_B \cup \mathcal{C}) \subseteq G$ . The objective is for Alice and Bob to detect all cliques (of all sizes) of  $G$  and to minimize the number of bits communicated.

We show that in such framework, there is a two-party communication protocol for listing all cliques (of all sizes) that uses  $\mathcal{O}(\sqrt{n}|\mathcal{C}|)$  bits of communication, where  $\mathcal{C}$  are the edges shared by Alice and Bob. This shows that we cannot improve our lower bounds for the  $K_\ell$ -detection problem, for  $\ell = \mathcal{O}(\sqrt{n})$ , in the CONGEST model (cf. Theorem 5) using the two-party communication framework in the vertex partition model.

Observe that without any communication between the two players, Alice can detect every clique that contains at most one vertex of  $V_B$ , and, similarly, Bob can detect every clique that contains at most one vertex of  $V_A$  (in particular, listing all triangles does not require any communication). Our task is hence to detect every clique consisting of at least two  $V_A$  vertices and at least two  $V_B$  vertices. We consider two cases:

1. Suppose that  $|\mathcal{C}| \geq n^{3/2}$ . Then Alice sends all edges  $E_A$  to Bob by encoding all entries in the adjacency matrix of  $G[V_A]$ , which requires at most  $n^2 \leq \sqrt{n}|\mathcal{C}|$  bits. Since Bob then knows the entire graph  $G$ , he can detect all cliques.
2. Suppose that  $|\mathcal{C}| < n^{3/2}$ . For any vertex  $v \in V$ , let  $d_v$  be the number of edges of  $\mathcal{C}$  incident to  $v$ , let  $V_{\leq \sqrt{n}} \subseteq \{v \in V_A : d_v \leq \sqrt{n}\}$ , and let  $V_{> \sqrt{n}} = V_A \setminus V_{\leq \sqrt{n}}$ . We first show how to detect every clique that contains at least one vertex of  $V_{\leq \sqrt{n}}$ . Then, we show how to detect every clique that does not contain any vertex of  $V_{\leq \sqrt{n}}$ .
  - a. For every  $v \in V_{\leq \sqrt{n}}$ , Bob sends the induced subgraph  $G_B[\Gamma_G(v) \cap V_B]$  (its adjacency matrix) to Alice (observe that Bob knows the set  $V_{\leq \sqrt{n}}$  without communication). This requires at most  $\sqrt{n}|\mathcal{C}|$  bits, since

$$\sum_{v \in V_{\leq \sqrt{n}}} d_v^2 \leq \sqrt{n} \sum_{v \in V_{\leq \sqrt{n}}} d_v \leq \sqrt{n}|\mathcal{C}|.$$

Alice can thus detect any clique that contains at least one vertex of  $V_{\leq \sqrt{n}}$ .

- b. Observe that  $|V_{>\sqrt{n}}| \leq \frac{|\mathcal{C}|}{\sqrt{n}}$ . Alice sends the entire subgraph  $G_A[V_{>\sqrt{n}}]$  (again, its adjacency matrix) to Bob. This requires at most  $\sqrt{n}|\mathcal{C}|$  bits, since

$$|V_{>\sqrt{n}}|^2 \leq \left(\frac{|\mathcal{C}|}{\sqrt{n}}\right)^2 \leq |\mathcal{C}| \cdot \frac{|\mathcal{C}|}{n} \leq \sqrt{n}|\mathcal{C}|,$$

using the assumption  $|\mathcal{C}| \leq n^{3/2}$ . Bob can thus detect every clique that does not contain any vertex of  $V_{\leq\sqrt{n}}$ .

We thus obtain the following theorem:

► **Theorem 10.** *There is a two-party communication protocol in the vertex partition model for listing all cliques (of all sizes) that uses  $\mathcal{O}(\sqrt{n}|\mathcal{C}|)$  communication rounds, where  $\mathcal{C}$  is the set of shared edges between Alice and Bob.*

## 5 Conclusions

In this paper, we give the first non-trivial lower bound for the problem of detecting a clique  $K_\ell$ , for  $\ell \geq 4$ , in the classical distributed CONGEST model. We show that detecting  $K_\ell$  requires  $\Omega\left(\frac{n}{(\ell+\sqrt{n})\mathfrak{b}}\right)$  communication rounds, for every  $\ell \geq 4$ , where  $\mathfrak{b}$  is the bandwidth of the communication links. Our lower bound is complemented by a matching upper bound obtained by a two-party communication protocol in the vertex partition model for listing all cliques of all sizes. This demonstrates that our lower bound cannot be improved using the two-party communication framework.

We leave as a great open question whether the complexity of clique detection in the CONGEST model is sublinear, or one needs  $\tilde{\Theta}(n)$  communication rounds to detect even a copy of  $K_4$ . Since the two-party communication approach used in our lower bound cannot be improved further, we do not have any intuition whether the lower bound is tight, or could be improved significantly. On the other hand, the very recent  $\tilde{\mathcal{O}}(\sqrt{n})$ -communication rounds algorithm for detecting a triangle [5] raises some hopes that maybe also  $K_4$  could be detected in a sublinear number of rounds.

---

## References

- 1 Zvika Brakerski and Boaz Patt-Shamir. Distributed discovery of large near-cliques. *Distributed Computing*, 24(2):79–89, 2011.
- 2 Keren Censor-Hillel, Eldar Fischer, Gregory Schwartzman, and Yadu Vasudev. Fast distributed algorithms for testing graph properties. In *Proceedings of the 30th International Symposium on Distributed Computing (DISC)*, pages 43–56, 2016.
- 3 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 35th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 143–152, 2015.
- 4 Keren Censor-Hillel, Seri Khoury, and Ami Paz. Quadratic and near-quadratic lower bounds for the CONGEST model. In *Proceedings of the 31st International Symposium on Distributed Computing (DISC)*, pages 10:1–10:16, 2017.
- 5 Yi-Jun Chang, Seth Pettie, and Hengjie Zhang. Distributed triangle detection via expander decomposition. *CoRR*, abs/1807.06624, 2018. [arXiv:1807.06624](https://arxiv.org/abs/1807.06624).
- 6 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)*, pages 195–209, 2012.

- 7 Andrew Drucker, Fabian Kuhn, and Rotem Oshman. On the power of the congested clique model. In *Proceedings of the 33rd Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 367–376, 2014.
- 8 Guy Even, Orr Fischer, Pierre Fraigniaud, Tzlil Gonen, Reut Levi, Moti Medina, Pedro Montealegre, Dennis Olivetti, Rotem Oshman, Ivan Rapaport, and Ioan Todinca. Three notes on distributed property testing. In *Proceedings of the 31st International Symposium on Distributed Computing (DISC)*, pages 15:1–15:30, 2017.
- 9 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, New York, NY, USA, 2018. ACM.
- 10 Pierre Fraigniaud and Dennis Olivetti. Distributed detection of cycles. In *Proceedings of the 29th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 153–162, 2017.
- 11 Tzlil Gonen and Rotem Oshman. Lower bounds for subgraph detection in the CONGEST model. In *Proceedings of the 21st International Conference on Principles of Distributed Systems (OPODIS)*, pages 6:1–6:16, 2017.
- 12 Taisuke Izumi and François Le Gall. Triangle finding and listing in CONGEST networks. In *Proceedings of the 37th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 381–389, 2017.
- 13 Svante Janson, Tomasz Łuczak, and Andrzej Ruciński. *Random Graphs*. John Wiley & Sons, 2011.
- 14 Bala Kalyanasundaram and Georg Schnitger. The probabilistic communication complexity of set intersection. *SIAM Journal on Discrete Mathematics*, 5(4):545–557, 1992.
- 15 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)*, pages 4:1–4:16, 2017.
- 16 Eyal Kushilevitz and Noam Nisan. *Communication Complexity*. Cambridge University Press, 1997.
- 17 Gopal Pandurangan, Peter Robinson, and Michele Scquizzato. Tight bounds for distributed graph computations. *CoRR*, abs/1602.08481, 2016.
- 18 David Peleg. *Distributed Computing: A Locality-Sensitive Approach*. SIAM Monographs on Discrete Mathematics and Applications. SIAM, Philadelphia, PA, 2000.



# A Wealth of Sub-Consensus Deterministic Objects

**Eli Daian**

School of Computer Science, Tel-Aviv University, Israel  
eliyahud@post.tau.ac.il

**Giuliano Losa**

Computer Science Department, University of California, Los Angeles, CA, USA  
giuliano@cs.ucla.edu

**Yehuda Afek**

School of Computer Science, Tel-Aviv University, Israel  
afek@post.tau.ac.il

**Eli Gafni**

Computer Science Department, University of California, Los Angeles, CA, USA  
eli@ucla.edu

---

## Abstract

The consensus hierarchy classifies shared an object according to its consensus number, which is the maximum number of processes that can solve consensus wait-free using the object. The question of whether this hierarchy is precise enough to fully characterize the synchronization power of deterministic shared objects was open until 2016, when Afek et al. showed that there is an infinite hierarchy of deterministic objects, each weaker than the next, which is strictly between  $i$  and  $i + 1$ -processors consensus, for  $i \geq 2$ . For  $i = 1$ , the question whether there exist a deterministic object whose power is strictly between read-write and 2-processors consensus, remained open.

We resolve the question positively by exhibiting an infinite hierarchy of simple deterministic objects which are equivalent to set-consensus tasks, and thus are stronger than read-write registers, but they cannot implement consensus for two processes. Still our paper leaves a gap with open questions.

**2012 ACM Subject Classification** Theory of computation → Distributed computing models

**Keywords and phrases** shared memory, distributed algorithms, wait-free, set consensus

**Digital Object Identifier** 10.4230/LIPIcs.DISC.2018.17

**Funding** The work of Yehuda Afek and Eli Gafni was partially supported by the United States-Israel Binational Science Foundation (grant 2014226). This material is based upon work supported by the National Science Foundation under Grant No. 1655166.

## 1 Introduction

Shared memory objects have been classified by Herlihy [19] by their consensus number, where the consensus number of an object  $O$  is the maximum number of processes which can solve the consensus task in the wait-free model using any number of copies of  $O$ <sup>1</sup>. Herlihy also

---

<sup>1</sup> Read-write registers are also usually allowed, in addition to copies of  $O$ , but this is superfluous since any non-trivial object can implement bounded-use registers [7], and bounded-use suffices when solving a task.



showed that  $n$ -consensus objects are universal for  $n$  processes, meaning that, for  $n$  processes, any other object can be implemented wait-free using  $n$ -consensus objects.

Until recently, it was not known whether an object of consensus power  $n$  can be implemented wait-free using  $n$ -consensus objects (i.e., objects that can be used to solve consensus among at most  $n$  processes) in a system of more than  $n$  processes (as a special case, the Common2 [2, 5] conjecture stipulates that all objects of consensus number 2 can be implemented using consensus for 2 processes). If this were the case, then the consensus hierarchy would offer a complete characterization of the synchronization power of distributed objects.

Addressing this question, requires first to precisely define the computation model used and the notion of synchronization power. Several object binding models exist, e.g. with a notion of ports, such as in the hard-wired and soft-wired binding models [11], or without ports, such as in the oblivious model [20]. There are also several ways to compare synchronization power, such as using non-blocking implementations or wait-free implementations, and by restricting the comparison to the power to implement tasks.

In this paper, we work in the oblivious object model. Moreover, we are just concerned with the power of objects to wait-free solve task defined over finite number of processors. It is easy to see that for this if we have an implementation of the object the implementation does not need to be a wait-free implementation, it is enough that it will be non-blocking, or as called in other places lock-free.

In 2016, Afek et al. [1] constructed for every  $n \geq 2$  an infinite sequence of deterministic objects (in the oblivious model)  $O_{n,k}$ ,  $k \in \mathbb{N}$ , of consensus number  $n$ , and such that  $O_{n,k}$  cannot be used to obtain a non-blocking implementation of  $O_{n,k+1}$  in a system of  $nk + n + k$  processes. Thus, for every  $n$ , the  $O_{n,k}$  objects have strictly increasing synchronization power, as measured by the non-blocking implementation relation. This shows that consensus number alone is not sufficient to characterize the synchronization power of deterministic objects at levels  $n \geq 2$  of the consensus hierarchy. As a special case, this also refutes the Common2 conjecture.

However, the case for consensus number 1 remained an open question, and it was conjectured that any deterministic object of consensus number 1 is equivalent to read-write registers, meaning that the object can solve exactly the same tasks that are solvable with read-write registers, no more, no less.

Herlihy [18] presented a consensus number 1 object that cannot be implemented wait-free from read-write register. But nevertheless it was implemented non-blocking (lock-free) from read-write registers, thus it did not refute the conjecture that every consensus number 1 object can be implemented non-blocking from read-write registers. Chan et al. [12] showed that for every set-consensus task, there exists an equivalent soft-wired non-deterministic object.

The main result of this paper refutes the above conjecture by presenting a deterministic object, Write and Read Next ( $WRN_k$ ), in the oblivious binding model, satisfying:

► **Theorem 1.** *For all integers  $k \geq 3$ , there is a deterministic object,  $WRN_k$ , whose consensus number is 1 but which cannot be implemented non-blocking from registers in a system of  $n > k$  processes.*

The second result of this paper applies to a one-shot variant,  $1sWRN_k$ , of  $WRN_k$ . Assuming that the object may be accessed at most once by each process and that no two processes use the same argument in their invocation, we show the following theorem:

► **Theorem 2.**  *$1sWRN_k$  and  $(k, k - 1)$ -set consensus have equivalent synchronization power (i.e., each can implement the other).*



Since  $(k, k - 1)$ -set-consensus is strictly weaker than  $(k + 1, k)$ -set-consensus, this gives rise to an infinite hierarchy among the  $\text{WRN}_k$  objects, such that  $1\text{sWRN}_{k'}$  objects are stronger than (can implement but not be implemented from)  $1\text{sWRN}_k$  objects if  $k < k'$ . Since  $1\text{sWRN}_k$  objects have more synchronization power than simple read-write registers, and cannot solve the consensus task for 2 processes, this shows the existence of an infinite number of object classes between simple read-write registers and 2-consensus.

The rest of the paper is structured as follows. The model is given in section 2. The  $\text{WRN}_k$  object and its one-shot variant  $1\text{sWRN}_k$  are presented in Section 3. We show two set consensus implementations that use these objects in section 4. A construction of  $1\text{sWRN}_k$  from  $(k, k - 1)$ -set consensus object is presented in section 5.  $\text{WRN}_k$  is proved to be weaker than 2-consensus in Section 6. The implied infinite hierarchy is presented in Section 7. Finally, conclusions and open questions are discussed in section 8.

## 2 Model

We follow the standard asynchronous shared memory model with oblivious objects, as defined in [1], in which processes communicate with one another by applying atomic operations, called steps, to shared objects. Each object has a set of possible values or states. Each operation (together with its inputs) is a partial mapping, taking each state to a set of states. A shared object is *deterministic* if each operation takes each state to a single state and its associated response is a function of the state to which the operation is applied.

A *configuration* specifies the state of every process and the value of every shared object. An *execution* is an alternating sequence of configurations and steps, starting from an initial configuration. Processes behave in accordance with the algorithm they are executing. If  $C$  is a configuration and  $s$  is a sequence of steps, we denote by  $Cs$  the configuration (or in the case of nondeterministic objects, the set of possible configurations) when the sequence of steps  $s$  is performed starting from configuration  $C$ .

An *implementation* of a sequentially specified object  $O$  consists of a representation of  $O$  from a set of shared base objects and algorithms for each process to apply each operation supported by  $O$ . The implementation is *deterministic* if all its algorithms are deterministic. The implementation is *linearizable* if, in every execution, there is a sequential ordering of all completed operations on  $O$  and a (possibly empty) subset of the uncompleted operations on  $O$  such that:

1. If  $op$  is completed before  $op'$  begins, then  $op$  occurs before  $op'$  in this ordering.
2. The behavior of each operation in the sequence is consistent with its sequential specification (in terms of its response and its effect on shared objects).

An implementation of an object  $O$  is *wait-free* if, in every execution, each process that takes sufficiently many steps eventually completes each of its operations on  $O$ . The implementation is *non-blocking* if, starting from every configuration, if enough steps are taken, then there exists a process that completes its operation. Note that a wait-free implementation is also a non-blocking implementation. In the rest of this paper, we discuss only deterministic and linearizable wait-free implementations.

A *task* specifies what combinations of output values are allowed to be produced, given the input value of each process and the set of processes producing output values. A wait-free or non-blocking solution to a task (both notions are equivalent when considering algorithms that solve tasks) is an algorithm in which each process that takes sufficiently many steps eventually produces an output value, and such that the collection of output values satisfies the specification of the task given the input values of the process.

## 17:4 A Wealth of Sub-Consensus Deterministic Objects

A task is solvable wait-free if and only if it is solvable non-blocking. This is because, in a non-blocking implementation of a bounded problem, at least one processor eventually terminates. A processor that terminates stops participating, and thus, because the implementation is non-blocking, another process eventually terminates, and so on until all processes that take sufficiently many steps have terminated, which fulfills the wait-free requirement. More generally, for any problem in which there is a bound on the number of operations that processors must complete, there is no difference between non-blocking and wait-free.

In the *consensus task*, each process,  $p_i$ , has an input value  $x_i$  and must output a value  $y_i$  that satisfies the following two properties:

**Validity.** Every output is the input of some process.

**Agreement.** All outputs are the same.

We say that an execution of an algorithm solving consensus *decides a value* if that value is the output of some process.

The *k-set consensus task*, introduced by [14, 15], is defined in the same way, except that agreement is replaced by the following property:

**k-agreement.** There are at most  $k$  different output values.

Note that the 1-set consensus task is the same as the consensus task.

An object has *consensus number*  $n$  if there is a wait-free algorithm that uses only copies of this object and registers to solve consensus for  $n$  processes, but there is no such an algorithm for  $n + 1$  processes. An object has an infinite consensus number if there is such algorithm for each positive integer  $n$ .

For all positive integers  $k < n$ , an  $(n, k)$ -set consensus nondeterministic object [10] supports one operation, **propose**, which takes a single non-negative integer as input. The value of an  $(n, k)$ -set consensus object is a set of at most  $k$  values, which is initially empty, and a count of the number of **propose** operations that have been performed on it (to a maximum of  $n$ ). The first **propose** operation adds its input to the set. Any other **propose** operation can nondeterministically choose to add its input to the set, provided the set has size less than  $k$ . Each of the first  $n$  **propose** operations performed on the object *nondeterministically* returns an element from the set as its output. All subsequent **propose** operations hang the system in a manner that cannot be detected by the processes.

A variant of the consensus task is the election task, in which all participating processes propose their own identifiers (rather than proposing some value). It also has the variable of  $k$ -set election task, that is basically a  $k$ -set consensus task, in which the identifiers of the processes are proposed. It was shown in [3] that the  $k$ -set consensus task is computationally equivalent to the  $k$ -set election task.

The  $k$ -strong set election task is a  $k$ -set election task, with the following self election property:

**Self Election.** If some process  $p_i$  decides on  $p_j$ , then  $p_j$  also decides on  $p_j$ .

It was shown in [9] that the  $k$ -strong set election task can be implemented using  $k$ -set election implementations, and thus the  $k$ -set election and  $k$ -strong set election tasks are computationally equivalent.

---

**Algorithm 1** A sequential specification of the atomic WRN operation of a  $\text{WRN}_k$  object.

---

```

1: function WRN( $i, v$ ) ▷  $i \in \{0, \dots, k-1\}, v \neq \perp$ 
2:    $A[i] \leftarrow v$ 
3:   return  $A[(i+1) \bmod k]$ 
4: end function

```

---

**Algorithm 2**  $(k-1)$ -Set consensus using a  $\text{WRN}_k$  object.

---

```

1: function Propose( $v_i$ ) ▷ For process  $P_i, 0 \leq i < k$ 
2:    $t \leftarrow \text{WRN}(i, v_i)$  ▷  $t$  is a local variable.
3:   if  $t \neq \perp$  then return  $t$ 
4:   else return  $v_i$ 
5:   end if
6: end function

```

---

### 3 Write and Read Next Objects

For every  $k \geq 2$ , we introduce the WriteAndReadNext $_k$  (or  $\text{WRN}_k$ ) object, that has a single operation – WRN. This operation accepts an index  $i$  in the range  $\{0, \dots, k-1\}$ , and a value  $v \neq \perp$ . It returns the value  $v'$  that was passed in the previous invocation to WRN with the index  $(i+1) \bmod k$ , or  $\perp$  if there is no such previous invocation.

A possible implementation of  $\text{WRN}_k$  consists of  $k$  registers,  $A[0], \dots, A[k-1]$ , initially initialized to  $\perp$ . A sequential specification of the atomic WRN operation is presented in Algorithm 1.

The OneShot $\text{WRN}_k$  (or  $1\text{sWRN}_k$ ) object is similar to  $\text{WRN}_k$ , but any index can be used at most once. Any attempt to invoke  $1\text{sWRN}$  with the same index twice is illegal, and hangs the system in a manner that cannot be detected by any process.

Note that the requirement that processes do not use the same argument in their invocation is reminiscent of the soft-wired model, in which there cannot be concurrency on a port. We could have chosen to specify  $1\text{sWRN}_k$  in the soft-wired binding model. This would have avoided ad-hoc assumptions about how processes use of the  $1\text{sWRN}_k$  object. We opted not to do so in order to use the oblivious object-binding model exclusively.

For  $k = 2$ ,  $\text{WRN}_2$  is simply a SWAP object, whose consensus number is known to be 2 [19]. From now on, we assume  $k \geq 3$ , unless stated otherwise.

## 4 Solving $(k, k-1)$ -Set Consensus using $\text{WRN}_k$ Objects

### 4.1 Solution in a System of $k$ Processes

For any  $k \geq 3$ , a  $\text{WRN}_k$  object can solve the  $(k, k-1)$ -set consensus task for  $k$  processes with unique ids taken from  $\{0, \dots, k-1\}$ , using the following algorithm (also described in Algorithm 2): Assume the processes are  $P_0, \dots, P_{k-1}$ , and their values are  $v_0, \dots, v_{k-1}$ . Process  $P_i$  invokes a  $1\text{sWRN}$  with index  $i$  and value  $v_i$ . If the output of the operation,  $t$ , is  $\perp$ ,  $P_i$  decides  $v_i$ . Otherwise, it decides  $t$ .

Since it is illegal for a process to propose multiple values (with the same ID) in the set consensus task, WRN can be replaced by  $1\text{sWRN}$ , that is invoked at most once with each index.

► **Claim 3.** *Algorithm 2 is wait free.*

► **Claim 4.** *The first process to perform WRN decides its own proposed value.*

## 17:6 A Wealth of Sub-Consensus Deterministic Objects

**Proof.** Since it is the first one to invoke  $\text{WRN}$ , the output of  $\text{WRN}$  is  $\perp$ , and hence the process decides on its own proposed value.  $\blacktriangleleft$

► **Claim 5.** *Let  $P_i$  be the last process to perform  $1s\text{WRN}$ . So  $P_i$  decides the proposal of  $P_{(i+1) \bmod k}$ .*

**Proof.** Since  $P_i$  is the last one to invoke  $\text{WRN}$ ,  $P_{(i+1) \bmod k}$  has already completed its  $\text{WRN}$  invocation. Theretofore,  $P_i$  receives  $v_{(i+1) \bmod k}$  as the output from  $\text{WRN}$ . Hence,  $P_i$  decides the value of  $P_{(i+1) \bmod k}$ .  $\blacktriangleleft$

► **Claim 6 (Validity).** *A process  $P_i$  can decide its proposed value, or the proposed value of  $P_{(i+1) \bmod k}$ .*

► **Claim 7.** *A process  $P_i$  decides its own proposed value if  $P_{(i+1) \bmod k}$  have not invoked  $\text{WRN}$  yet.*

► **Corollary 8 (( $k - 1$ )-agreement).** *Assume the proposals are pairwise different (there are exactly  $k$  different proposals). So at most  $k - 1$  values can be decided.*

**Proof.** Let  $P_i$  be the first process to invoke  $\text{WRN}$ , and  $P_j$  be the last process to invoke  $\text{WRN}$ . From Claim 4,  $P_i$  decides its proposal. From Claim 5,  $P_j$  decides the proposal of  $P_{(j+1) \bmod k}$ . From claim 7, no process decides the proposal of  $P_j$ .  $\blacktriangleleft$

► **Corollary 9.** *Algorithm 2 solves the ( $k - 1$ )-set consensus task for  $k$  processes.*

► **Corollary 10.**  *$1s\text{WRN}_k$  and  $\text{WRN}_k$  cannot be implemented from atomic read-write registers. Hence,  $1s\text{WRN}_k$  and  $\text{WRN}_k$  are stronger than registers.*

### 4.2 Solution in a System with $k$ Participating Processes Out of Many

Assuming that each process has a unique name in  $\{0, \dots, k - 1\}$  might be a strong limitation in some models. In this section, we assume we have at most  $k$  participating processes, whose names are taken from  $\{0, \dots, M - 1\}$ , where  $M \gg k$ .

In [4, 6], wait-free algorithms have been shown that use registers only to rename  $k$  processes from  $\{0, \dots, M - 1\}$  to  $k$  unique names in the range  $\{0, \dots, 2k - 2\}$ . So we shall relax our assumption, and assume now we have at most  $k$  participating processes, whose names are in  $\{0, \dots, 2k - 2\}$ . Let us consider the set of functions  $\{0, \dots, 2k - 2\} \rightarrow \{0, \dots, k - 1\}$ , call it  $\mathcal{F}$ . So  $|\mathcal{F}| = (2k - 1)^k$  is finite, and we can fix an arbitrary ordering of  $\mathcal{F} = \{f_1, \dots, f_{(2k-1)^k}\}$ .

The ( $k - 1$ )-set consensus algorithm for  $k$  processes is described in Algorithm 3. It uses  $(2k - 1)^k$  instances of  $\text{WRN}_k$  objects,  $W[1], \dots, W[(2k - 1)^k]$ . First, the process name is renamed to be  $j \in \{0, \dots, 2k - 2\}$ . Then, for each  $\ell \in \{1, \dots, (2k - 1)^k\}$  (in this exact order for all processes), the process invokes  $W[\ell].\text{WRN}$  with the index  $f_\ell(j)$ , and the proposed value  $v_j$ . If the result of such a  $\text{WRN}$  operation returns a value different than  $\perp$ , the process immediately decides on this returned value, and returns without continuing to the following iterations. If the process received  $\perp$  from all the  $\text{WRN}$  operations on  $W[1], \dots, W[(2k - 1)^k]$ , it decides its own proposed value.

► **Claim 11 (Validity).** *Every decided value in Algorithm 3 was proposed by some process.*

---

**Algorithm 3**  $(k - 1)$ -Set consensus for  $k$  processes out of many using  $\text{WRN}_k$  objects.
 

---

```

1: shared array of  $\text{WRN}_k$  objects  $W[\ell]$ ,  $1 \leq \ell \leq (2k - 1)^k$ 
2: function  $\text{Propose}(v)$   $\triangleright$  For process whose name is in  $\{0, \dots, M - 1\}$ 
3:    $j \leftarrow \text{Rename}$   $\triangleright j \in \{0, \dots, 2k - 2\}$ 
4:   for  $\ell = 1, \dots, (2k - 1)^k$  do
5:      $i \leftarrow f_\ell(j)$   $\triangleright i \in \{0, \dots, k - 1\}$  is a local variable.
6:      $t \leftarrow W[\ell].\text{WRN}(i, v)$   $\triangleright t$  is a local variable.
7:     if  $t \neq \perp$  then return  $t$ 
8:     end if
9:   end for
10:  return  $v$   $\triangleright$  Reaching here means  $t$  was  $\perp$  in all iterations
11: end function

```

---

**Proof.** Each process can only write its proposal to the WRN objects, and hence only proposal values or  $\perp$  can be returned from the WRN operations. Therefore, if a WRN operation performed by the process  $P$  does not return  $\perp$ , it returns a proposal of some process  $Q$ , and hence  $P$  decides on the proposal of  $Q$ . If  $P$  gets only  $\perp$  from all the WRN invocations, it decides on its own proposal.  $\blacktriangleleft$

► **Claim 12.** For every iteration number  $1 \leq \ell \leq (2k - 1)^k$ , there is a process that invokes  $W[\ell].\text{WRN}$  in Algorithm 3, and the first such process returns  $\perp$ .

**Proof.** The first process to invoke  $W[\ell].\text{WRN}$  returns  $\perp$  by the definition of the WRN objects, and hence it also continues to the next iteration. Using induction, it is clear that a process gets to the first iteration and continues to the second one, and hence there is a process that accesses  $W[(2k - 1)^k]$ , and the first such process returns  $\perp$ .  $\blacktriangleleft$

► **Corollary 13.** There is a process that invokes  $W[(2k - 1)^k].\text{WRN}$  in Algorithm 3, and the first such process decides on its proposed value.

► **Claim 14.** Assume a process  $P$  got the output  $x \neq \perp$  from its invocation of  $W[(2k - 1)^k].\text{WRN}$ .  $x$  is the value of another process  $Q$ , that invoked  $W[(2k - 1)^k].\text{WRN}$  before  $P$ .

► **Corollary 15.** Assume exactly  $k$  inputs were proposed to Algorithm 3. Also assume the processes  $P$  and  $Q$  proposed the values  $x$  and  $y$ , respectively, and assume  $P$  decides on  $y$ .  $Q$  does not decide on  $x$ .

► **Claim 16.** Assume all  $k$  processes access the construction of algorithm 3, each with a different input. There is a process  $P$  that decides on the value of another process  $Q$ .

**Proof.** Let  $R$  be the set of new names of the processes after renaming them in line 3,  $|R| = k$ . Hence there is a mapping  $f_{\ell^*} \in \mathcal{F}$  such that  $\{f_{\ell^*}(i) \mid i \in R\} = \{0, 1, \dots, k - 1\}$ . Either some process returns before iteration  $\ell^*$ , or all of them reach iteration  $\ell^*$ .

In the former case, process  $P$  quits in iteration  $\ell' < \ell^*$ , and  $P$  gets a proposal  $v$  of another process from  $W[\ell']$ , and decides  $v$ .

In the latter case, let  $j_P$  be the name of  $P$  after the renaming in line 3. Let  $P$  be the last process to invoke  $W[\ell^*].\text{WRN}$ . So  $P$  invoked it with the index  $f_{\ell^*}(j_P)$ . Let  $Q$  be the process that invoked  $W[\ell^*].\text{WRN}$  with the index  $(f_{\ell^*}(j_P) + 1) \bmod k$  (there is such process because of the selection of  $\ell^*$ ).  $Q$  invoked  $W[\ell^*].\text{WRN}$  before  $P$ , and hence the  $P$ 's invocation of  $W[\ell^*].\text{WRN}$  results in the proposal of  $Q$ . Therefore,  $P$  decides on the proposal of  $Q$ .  $\blacktriangleleft$

---

**Algorithm 4** Implementing relaxed  $\text{WRN}_k$  using  $\text{1sWRN}_k$  and registers.
 

---

```

1: shared  $\text{1sWRN}_k$  object
2: shared array of registers  $A[i]$ ,  $0 \leq i < k$ , initialized to 0
3: function  $\text{R1xWRN}(i, v)$   $\triangleright 0 \leq i < k, v \neq \perp$ 
4:    $\text{Inc}(A[i])$   $\triangleright$  Increment  $A[i]$  by 1.
5:    $c \leftarrow \text{Read}(A[i])$   $\triangleright c$  is a local variable.
6:   if  $c = 1$  then return  $\text{1sWRN}(i, v)$ 
7:   else return  $\perp$ 
8:   end if
9: end function

```

---

► **Corollary 17** ( $(k-1)$ -agreement). *Assume exactly  $k$  inputs were proposed to Algorithm 3. So there is a process  $P$  whose proposal is not decided by any process.*

**Proof.** Let  $v_i$  and  $d_i$  be the proposal and decision values of process  $P_i$ . Let  $A$  be the set of processes  $P_i$  such that  $x_i \neq y_i$ . From Claim 16,  $A \neq \emptyset$ .

Each process  $P_i \in A$  has an iteration  $\ell_i$  in which  $d_i$  was returned from its invocation of  $W[\ell_i].\text{WRN}$ . Let  $\ell'$  be the minimal such iteration, and let  $P_i \in A$  be the last process to invoke  $W[\ell'].\text{WRN}$ .

No value was decided by any process in iteration  $\ell < \ell'$ , and hence  $v_i$  was not decided by any process in these iterations. The value  $v_i$  is unknown to  $W[\ell']$  before  $P_i$  invokes  $W[\ell'].\text{WRN}$ . Therefore,  $v_i$  cannot be returned by any  $W[\ell'].\text{WRN}$  invocation prior to  $P_i$ 's invocation. In  $P_i$ 's invocation the value  $d_i \neq v_i$  is returned. From the selection of  $i$ , every  $W[\ell_j].\text{WRN}$  invocation after  $P_i$ 's invocation returns  $\perp$ , and hence no process returns  $v_i$  in iteration  $\ell'$ .

$P_i$  have not participated in any latter iteration, and hence  $v_i$  was not seen by any  $\text{WRN}$  object in such an iteration, so it could not be returned from any  $\text{WRN}$  invocation. Therefore,  $v_i$  is not returned by any process also after iteration  $\ell'$ . ◀

► **Corollary 18.** *Algorithm 3 solves the  $(k-1)$ -set consensus task for  $k$  processes whose names are taken from  $\{0, \dots, M-1\}$ .*

Algorithm 3 uses  $\text{WRN}_k$  objects that cannot be trivially replaced by  $\text{1sWRN}_k$  objects, since after the renaming, processes  $P$  and  $Q$  get the new names  $0 \leq i < j < 2k-1$ , and there is a mapping  $f_\ell \in \mathcal{F}$  such that  $f_\ell(i) = f_\ell(j)$ . If both  $P$  and  $Q$  get to iteration  $\ell$ , both invoke  $W[\ell].\text{WRN}$  with the index  $f_\ell(i) = f_\ell(j)$ .

Although this fact might pose a problem, the correctness of the algorithm is based on the existence of an iteration  $\ell^*$  such that  $f_{\ell^*}$  maps all the renamed process names onto  $\{0, 1, \dots, k-1\}$ . This fact is being used in the proof of Claim 16 in order to show that there is a process that decides on the proposal of another process, and hence the  $(k-1)$ -agreement property is achieved.

A relaxed implementation of  $\text{WRN}_k$  using  $\text{1sWRN}_k$  is enough for implementing Algorithm 3. This relaxed implementation is described in Algorithm 4. The  $\text{1sWRN}_k$  object is protected by a counter for every legal index. This counter is a simple atomic register that can be incremented and read (each operation is a single step). When a process comes with the index  $i$ , it first increments the counter of index  $i$ , and then reads the value of that counter. If the read value is exactly 1, it is safe for the process to invoke  $\text{1sWRN}$  (in a similar manner to the flag principle [21]). Otherwise, the process cannot tell whether it is safe to invoke  $\text{1sWRN}$  or not, so it gives up, and returns  $\perp$  directly.

► **Claim 19 (Safety).** *At most one process invokes  $1sWRN$  with an index  $0 \leq i < k$  in Algorithm 4.*

**Proof.**  $1sWRN$  is invoked with an index  $i$  only by a process that read the value 1 (exactly) from  $A[i]$ . By contradiction, assume both  $P$  and  $Q$  read 1 from  $A[i]$ , and without loss of generality, let  $Q$  be the last process to increment  $A[i]$ . Since  $A[i]$  is initialized to 0 and  $Q$  is not the first process to increment it,  $Q$  must have read at least 2. ◀

► **Corollary 20.** *Algorithm 4 is using the  $1sWRN_k$  object legally.*

► **Claim 21.** *If exactly  $k$  processes arrive with  $k$  different indices,  $1sWRN$  is invoked by every participating process in Algorithm 4.*

**Proof.** Every process that comes with an index  $i$  is the only one that increments  $A[i]$ , so it is the only one to read the value 1 from  $A[i]$ , and hence it will invoke  $1sWRN$ . ◀

Algorithm 4 of a relaxed  $WRN_k$  object can be used as a substitution for the  $WRN_k$  objects in algorithm 3; lines 1 and 6 should be replaced by the following lines:

1: **shared array of  $WRN_k$  objects**  $W[\ell]$ ,  $1 \leq \ell \leq (2k-1)^k$   
 6:  $t \leftarrow W[\ell].R1xWRN(i, v)$  ▷  $t$  is a local variable.

If at round  $\ell$  two different processes access  $W[\ell]$  with the same index  $i$ , with the relaxed  $WRN_k$  the underlying  $1sWRN$  operation might not even get invoked, in which case both processes get  $\perp$  from their  $R1xWRN$  invocation, if a process accesses later  $W[\ell].R1xWRN$  with the index  $(i-1) \bmod k$ , this process might get  $\perp$  and continue to the next iteration, which is the opposite of the expected behavior with regular  $WRN_k$  objects.

However, in the proof of Claim 16, iteration  $\ell^*$  still exists, in which all  $k$  participating processes invoke  $W[\ell].R1xWRN$  with a different index, and claim 21 guarantees that in iteration  $\ell^*$ , the underlying  $1sWRN_k$  object gets accesses just like the regular  $WRN_k$  object. Hence Algorithm 3 solves the  $(k-1)$ -set consensus task for  $k$  processes using  $1sWRN_k$  objects as well.

## 5 Constructing $1sWRN_k$ from $(k, k-1)$ -Set Consensus Implementation

In this section we present an implementation of  $1sWRN_k$  object that uses  $(k, k-1)$ -strong set election (i.e., if process  $P_i$  decides in the proposal of  $P_j$ , then  $P_j$  also decides on its own proposal), which can be implemented using  $(k, k-1)$ -set consensus [9], and registers.

The base of the implementation is an array of registers, in which each process publishes its value (using the index), and reads the published value of its successor (by the index) if such a value is published, or  $\perp$  otherwise. Each process aims to return the read value of its successor, whether it is  $\perp$  or not. However, the first linearized operation must return  $\perp$ , and if the processes return their read value, the following execution has no first linearized operation: All processes write together their values, and then read together the values of their successors.

In order to avoid such cases, the implementation uses a doorway register. This doorway is initially open (i.e., the register value is *opened*), and once a process enters through the doorway (i.e., reads the value *opened*), it closes the doorway (i.e., writes the value *closed*). The processes that pass through the doorway use the strong set election implementation, and return the read published value of their successor only if they do not win the strong set



---

**Algorithm 5** Implementation of  $1sWRN_k$  using  $(k, k - 1)$ -Strong Set Election.
 

---

```

1: shared  $(k, k - 1)$ -strong set election implementation SSE
2: shared MWMR register Doorway, initially opened
3: shared SWMR register array  $R[i]$ ,  $0 \leq i < k$ ; initially  $R[i] = \perp$  for every  $i$ 
4: shared SWMR register array  $O[i]$ ,  $0 \leq i < k$ ; initially  $O[i] = \perp$  for every  $i$ 
5: function  $1sWRN(i, v)$   $\triangleright i \in \{0, \dots, k - 1\}$  is the index,  $v \notin \{\perp, \emptyset\}$  is the value.
6:    $R[i] \leftarrow v$   $\triangleright v$  is announced at the index  $i$ .
7:   if  $Read(Doorway) = opened$  then
8:      $Doorway \leftarrow closed$ 
9:     if  $SSE.Invoke(i) = i$  then
10:      return  $\perp$ 
11:     end if
12:   end if
13:    $SR \leftarrow Snapshot(R)$   $\triangleright SR$  is a local array.
14:    $O[i] \leftarrow SR$ 
15:    $SO \leftarrow Snapshot(O)$   $\triangleright SO$  is a local array.
16:   for  $j = 0, 1, \dots, k - 1$  do
17:     if  $SO[j][i] = v$  and  $SO[j][(i + 1) \bmod k] = \perp$  then
18:       return  $\perp$ 
19:     end if
20:   end for
21:   return  $SR[(i + 1) \bmod k]$ 
22: end function

```

---

election. If a process wins the strong set election, its  $1sWRN$  invocation returns  $\perp$ . Notice that using the strong set election without the doorway might result in a non-linearizable implementation: If a process completes its  $1sWRN$  invocation with the index  $(i + 1) \bmod k$  before another process issues its invocation with the index  $i$ , the latter is expected to return the value of the former. However, the latter invocation might win in the strong set election as well, in which case it would return  $\perp$ .

The described solution is not enough, since the result is non-linearizable. Consider the case in which the doorway has already been closed by an early invocation. Since the read and write operations are not atomic, the linearization might break between an invocation announces its value, and reads the value of its successor index.

For example, consider the following execution: (1) an invocation  $w_1$  with the index 1 can announce its value. (2) an invocation  $w_2$  with the index 2 announces its value. (3) The invocation  $w_1$  encounters a closed doorway, reads the value of  $w_2$  and returns it. (4) After  $w_1$  completes, an invocation  $w_3$  announces its value. (5)  $w_2$  reads the announces value of  $w_3$  and returns it. In this described execution,  $w_1$  would be linearized after  $w_2$ , that would be linearized after  $w_3$ . But  $w_3$  starts only after  $w_1$  has completed.

In order to overcome this kind of problem, two snapshots are being taken. The first snapshot reads the announced values, and the second one is used for announcing the snapshot every invocation observes, in order to detect scenarios similar to the one described above. If an invocation  $w_i$  observes the value of its successor invocation  $w_{(i+1) \bmod k}$ , but it also sees that there is another invocation  $w_j$  that saw the value of  $w_i$ , but did not see the value of  $w_{(i+1) \bmod k}$ , so  $w_i$  knows that it has started before  $w_{(i+1) \bmod k}$  finishes, and  $w_i$  returns  $\perp$ . A pseudo code of the implementation is presented in algorithm 5.

Let  $e$  be a legal execution that contains invocations to  $1sWRN$ , as described in Algorithm 5. Denote by  $\{w_i\}$  the invocations to  $1sWRN$ , such that  $w_i$  is the invocation with index  $i$  and input value  $v_i$ . Assume  $1sWRN$  was invoked for every index  $0 \leq i < k$  (otherwise, append



the missing invocations at the end of the execution). We will now see that Algorithm 5 is a linearizable implementation of  $1sWRN$ .

► **Claim 22.**  $w_i$  returns  $v_{(i+1) \bmod k}$  or  $\perp$ .

► **Claim 23.** There is an index  $0 \leq i < k$  such that  $w_i$  returns  $\perp$ .

**Proof.** The first invocation to check the doorway status (in line 7) invokes the strong set election, so the strong set election is invoked at least once. By definition, there is an invocation  $w_i$  that its strong set election invocation returns  $i$ , and then  $w_i$  returns  $\perp$  from Algorithm 5. ◀

► **Claim 24.** There is an index  $0 \leq i < k$  such that  $w_i$  return  $v_{(i+1) \bmod k}$ .

**Proof.** When some invocation takes a snapshot in line 13, all invocations that enter the doorway have already registered their values in  $R$ : Assume  $w_i$  does not read  $v_j$  in  $R$ . When  $w_i$  takes the snapshot in line 13, the doorway is already closed, and  $v_j$  is not written in  $R$ . So  $w_j$  writes  $v_j$  to  $R$  in line 6 after the doorway is closed. So  $w_j$  does not enter through the doorway.

At least one invocation reads in line 13, because an invocation reads  $R$  if it does not enter the doorway, or loses in the strong set election. Let  $w_i$  be the last invocation to write in line 6 that also reads in line 13. Claim by contradiction that  $w_i$  returns  $\perp$ .

So there is a an index  $0 \leq j < k$  such that  $w_i$  sees  $SO[j][i] = v_i$  and it also sees  $SO[j][(i+1) \bmod k] = \perp$ . In this case, when  $w_j$  takes a snapshot of  $R$  in line 13, it sees  $v_i$  in  $R$ , but not  $v_{(i+1) \bmod k}$ . So  $v_i$  is written to  $R$  before  $v_{(i+1) \bmod k}$ , and after the doorway is already closed. So  $w_{(i+1) \bmod k}$  writes to  $R$  after the doorway is closed, and after  $w_i$  writes to  $R$ , which is a contradiction to the selection of  $w_i$ . ◀

► **Lemma 25.** If  $w_i$  returns  $\perp$ , then  $w_{(i+1) \bmod k}$  finishes after  $w_i$  starts.

**Proof.** By a contradiction assume  $w_{(i+1) \bmod k}$  finishes before  $w_i$  starts. In this case, when  $w_i$  starts,  $v_{(i+1) \bmod k}$  is already written in  $R[(i+1) \bmod k]$  and the doorway is closed, and  $O[(i+1) \bmod k] \neq \perp$ .

Since  $w_i$  returns  $\perp$ , it must be done in line 18 in iteration  $0 \leq j < k$ , when  $w_j$  saw  $v_i$ , but not  $v_{(i+1) \bmod k}$ . Therefore,  $w_{(i+1) \bmod k}$  starts after  $w_i$  starts, that is after  $w_{(i+1) \bmod k}$  finishes, which is a contradiction. ◀

► **Lemma 26.** If  $w_i$  returns  $v_{(i+1) \bmod k}$ , then  $w_i$  finishes after  $w_{(i+1) \bmod k}$  starts.

**Proof.** Assume  $w_i$  finishes before  $w_{(i+1) \bmod k}$  starts. In this case, when  $w_i$  finishes, the value in  $R[(i+1) \bmod k]$  is  $\perp$ , so  $w_i$  returns  $\perp$  either if it wins the strong set election, or if it reads it from  $R[(i+1) \bmod k]$ . ◀

We now define a directed graph  $G = (V, E)$ , where  $V = \{w_i \mid 0 \leq i < k\}$ , and the set of edges is defined as follows:

- If  $w_i$  returns  $\perp$ , there is an edge from  $w_i$  to  $w_{(i+1) \bmod k}$ .
- If  $w_i$  returns  $v_{(i+1) \bmod k}$ , there is an edge from  $w_{(i+1) \bmod k}$  to  $w_i$ .

► **Claim 27.** There is an edge from  $w_i$  to  $w_{(i+1) \bmod k}$  if and only if there is no edge from  $w_{(i+1) \bmod k}$  to  $w_i$ .

► **Corollary 28.** There are no directed cycles in  $G$ .

**Proof.** The degree of each node in the graph is exactly 2, since the edges are between  $w_i$  and  $w_{(i+1) \bmod k}$ . Therefore, with the combination of Claim 27, if there is a cycle in  $G$ , its length is  $k$ .

Assume there is a cycle of length  $k$  in  $G$ . Using claim 23, let  $w_{i_1}$  be a **1sWRN** invocation using Algorithm 5 that returns  $\perp$ . Since  $w_{i_1}$  returns  $\perp$ , the cycle is in increasing order, e.g., for every  $0 \leq i < k$ , there is an edge from  $w_i$  to  $w_{(i+1) \bmod k}$ .

Using Claim 24, let  $w_{i_2}$  be a **1sWRN** invocation that returns  $v_{(i_2+1) \bmod k}$ . From the construction of  $G$ , there is an edge from  $w_{(i_2+1) \bmod k}$  to  $w_{i_2}$ , which is a contradiction to claim 27.  $\blacktriangleleft$

► **Corollary 29.** *There is a source and a sink in  $G$ .*

► **Corollary 30.** *The edges of  $G$  form a partial order.*

► **Lemma 31.** *Let  $p$  be an increasing indices directed path from  $w_i$  to  $w_j$ . That is:*

$$p = \langle w_i \rightarrow w_{(i+1) \bmod k} \rightarrow w_{(i+2) \bmod k} \rightarrow \cdots \rightarrow w_j \rangle$$

*Then  $w_j$  finishes after  $w_i$  starts.*

**Proof.** In this case, every  $w \in p \setminus \{w_j\}$  returns  $\perp$ . We use induction on  $p$  to show that  $w_j$  finishes after every  $w \in P$  starts. The base case is trivial:  $w_j$  finishes after it starts.

Inductively assume  $w_j$  finishes after  $w_{(i+x+1) \bmod k}$  starts. We now show that  $w_j$  finishes after  $w_{(i+x) \bmod k}$  starts. If  $w_{(i+x) \bmod k}$  enters through the doorway, it is impossible for  $w_j$  to finish before  $w_{(i+x) \bmod k}$  starts. Let us now consider the case in which  $w_{(i+x) \bmod k}$  encounters a closed doorway.

If  $w_{(i+x) \bmod k}$  reads  $\perp$  from  $R[(i+x+1)]$  in line 13, then it must have started before  $w_{(i+x+1) \bmod k}$  starts, which is before  $w_j$  finishes.

Consider the case in which  $w_{(i+x) \bmod k}$  reads  $v_{(i+x+1) \bmod k}$  from  $R[(i+x+1)]$  in line 13. Since  $w_{(i+x) \bmod k}$  returns  $\perp$ , it must have been in line 18 in iteration  $0 \leq \ell < k$  of the for loop of line 16. Therefore,  $w_\ell$  sees  $v_{(i+x) \bmod k}$  but not  $v_{(i+x+1) \bmod k}$ . Hence,  $w_{(i+x) \bmod k}$  writes to  $R$  in line 6 before  $w_{(i+x+1) \bmod k}$  does. It follows that  $w_{(i+x) \bmod k}$  starts before  $w_{(i+x+1) \bmod k}$  starts, that is before  $w_j$  finishes.

Hence  $w_i \in p$  starts before  $w_j$  finishes.  $\blacktriangleleft$

► **Lemma 32.** *Let  $p$  be a descending indices directed path from  $w_i$  to  $w_j$ . That is:*

$$p = \langle w_i \rightarrow w_{(i-1) \bmod k} \rightarrow w_{(i-2) \bmod k} \rightarrow \cdots \rightarrow w_j \rangle$$

*Then  $w_j$  finishes after  $w_i$  starts.*

**Proof.** In this case, every  $w \in p \setminus \{w_i\}$  does not return  $\perp$ . We use induction on the length of  $p$  to show that  $w_i$  starts before  $w_j$  finishes. The base case is trivial: lemma 26 shows that  $w_i$  starts before  $w_j$  finishes if the length of  $p$  is 1.

Assume the length of  $p$  is greater than 1. Inductively we assume that any decreasing indices path shorter than  $p$  satisfies the lemma. Also assume by contradiction that  $w_j$  finishes before  $w_i$  starts. Therefore,  $w_j$  does not read  $v_i$  from  $R$  in line 13. Since  $w_j$  returns  $v_{(j+1) \bmod k}$ , it has to read  $v_{(j+1) \bmod k}$  in  $R$  after  $w_{(j+1) \bmod k}$  writes it there. So there is an  $\ell$  such that  $w_\ell \in p$ , and  $w_j$  reads  $R[\ell] = v_\ell$  but  $R[(\ell+1) \bmod k] = \perp$ .

If operation  $w_\ell$  reads  $O[j] \neq \perp$ ,  $w_\ell$  would have to return  $\perp$  in line 18. Since  $w_\ell \in p$ , it returns  $v_{(\ell+1) \bmod k}$ . Therefore,  $w_\ell$  reads  $O[j] = \perp$  in line 15. So  $w_\ell$  reads  $O$  before  $w_j$  finishes. Reading  $O$  in line 15 is the last operation in the shared memory, so  $w_\ell$  finishes before  $w_j$  does. Since the path  $\langle w_i \rightarrow w_{(i-1) \bmod k} \rightarrow \cdots \rightarrow w_\ell \rangle$  is a decreasing path shorter than  $p$ , from the induction assumption,  $w_i$  starts before  $w_\ell$  finishes, that is before  $w_j$  finishes.  $\blacktriangleleft$

► **Corollary 33** (Transitivity). *Let  $p$  be a directed path from  $w_i$  to  $w_j$  in  $G$ . So  $w_j$  finishes after  $w_i$  starts.*

We build a total order of  $\{w_i \mid 0 \leq i < k\}$  inductively. For the base case, denote:  $S^0 = \emptyset$ ,  $T^0 = \{w_i \mid 0 \leq i < k\}$ .

Given  $S^j$  and  $T^j \neq \emptyset$ ,  $0 \leq j < k$ , we build  $S^{j+1}$  and  $T^{j+1}$  using the following construction: denote by  $\tilde{T}_j$  the set of invocations  $t \in T^j$ , such that  $t$  has no incoming edges in  $G$  from another invocation in  $T^j$ . Since  $T^j \neq \emptyset$  then also  $\tilde{T}_j \neq \emptyset$ , because there are no cycles in  $G$ . Let  $w^j$  be the first invocation in  $\tilde{T}_j$  to perform the write in line 6 (that is, to start running). We define  $S^{j+1}$  and  $T^{j+1}$  as follows:  $S^{j+1} = S^j \cup \{w^j\}$  and  $T^{j+1} = T^j \setminus \{w^j\}$ . Since  $|T^{j+1}| = |T^j| + 1$ , this construction is well defined for  $0 \leq j < k$ .

We define the total order  $\preceq$  as follows:  $w^i \preceq w^j$  if  $i \leq j$ .

► **Lemma 34.** *For every  $0 \leq j \leq k$ , there are no edges from  $T^j$  to  $S^j$ .*

**Proof.** We use induction on  $j$  for the proof. The base case is trivial, since  $S^0 = \emptyset$ .

Assume there are no edges from  $T^j$  to  $S^j$ . Since  $w^j \in \tilde{T}_j$ , there are no edges to  $w^j$  from  $T^j$  (and there is also no edge from  $w^j$  to itself). So there are no edges from  $T^{j+1} = T^j \setminus \{w^j\}$  to  $S^{j+1} = S^j \cup \{w^j\}$ . ◀

► **Corollary 35.**  $w^0$  returns  $\perp$ .

**Proof.** Assume  $w^0$  does not return  $\perp$ . Following the construction of  $G$ , there is an incoming edge to  $w^0$ . From lemma 34, there are no incoming edges to  $S^1 = \{w^0\}$ , in a contradiction. ◀

► **Corollary 36.**  $w_i$  returns  $\perp$  if and only if  $w_i \preceq w_{(i+1) \bmod k}$ .

**Proof.** Assume  $w_i$  returns  $\perp$ . Assume  $w_i = w^j$ . So  $w_i \in T^j$ , but  $w_i \in S^{j+1}$ . Since  $w_i$  returns  $\perp$ , following the construction of  $G$ , there is an edge from  $w_i$  to  $w_{(i+1) \bmod k}$ . Assuming that  $w_{(i+1) \bmod k} \in S^j$  would contradict lemma 34, so  $w_{(i+1) \bmod k} \in T^j$ , and therefore also  $w_{(i+1) \bmod k} \in T^{j+1}$ . So  $w_i \preceq w_{(i+1) \bmod k}$ . ◀

► **Corollary 37.**  $\preceq$  is a linearization of  $1sWRN$ . Therefore, algorithm 5 is a linearizable implementation of  $1sWRN_k$ .

Corollary 37 shows that  $1sWRN_k$  can be implemented using a  $(k, k-1)$ -set consensus implementation. This implies that  $1sWRN_k$  is equivalent to  $(k, k-1)$ -set consensus. In particular,  $1sWRN_k$  cannot solve the 2-process consensus task where  $k \geq 3$ .

## 6 WRN<sub>k</sub> is Weaker than 2-Consensus

Section 5 describes a linearizable construction of  $1sWRN_k$  using an implementation for  $(k, k-1)$ -set consensus. In this section we prove that neither  $WRN_k$  objects can solve the 2-process consensus task for  $k \geq 3$ , using a critical-state argument [17, 19].

We follow the standard definitions of *bivalent configuration*, *v-univalent configuration* and *critical configuration*, as defined in [17, 19].

► **Lemma 38.** *For each  $k \geq 3$ , there is no wait-free algorithm for solving the consensus task with 2 processes using only registers and  $WRN_k$  objects.*

**Proof.** Assume such an algorithm exists. Consider the possible executions of the processes  $P$  and  $Q$  of this algorithm, while proposing 0 and 1, respectively. Let  $C$  be a critical configuration of this run. Denote the next steps of  $P$  and  $Q$  from  $C$  as  $s_P$  and  $s_Q$ , respectively. Without loss of generality, we assume that  $Cs_P$  is a 0-univalent configuration, and  $Cs_Q$  is a 1-univalent configuration.

Following [19],  $s_P$  and  $s_Q$  both invoke a  $WRN$  operation on the same  $WRN_k$ .

---

**Algorithm 6**  $m$ -set consensus for  $n$  processes using  $\text{WRN}_k$  objects.

---

```

1: shared array  $W[j]$  of  $\text{WRN}_k$  objects,  $0 \leq j < \lceil \frac{n}{k} \rceil$ 
2: function Propose( $v_i$ )                                     ▷ For process  $P_i$ ,  $0 \leq i < n$ 
3:    $t \leftarrow W[\lfloor \frac{i}{k} \rfloor].\text{WRN}(i \bmod k, v_i)$        ▷  $t$  is a local variable.
4:   if  $t \neq \perp$  then return  $t$ 
5:   else return  $v_i$ 
6:   end if
7: end function

```

---

**Case 1.** Both  $s_P$  and  $s_Q$  perform WRN with the same index  $i$ .

The configurations  $C_{s_P}$  and  $C_{s_Q s_P}$  are indistinguishable for a solo run of  $P$ , but a solo run of  $P$  from  $C_{s_P}$  decides 0, while an identical solo run of  $P$  from  $C_{s_Q s_P}$  decides 1. This is a contradiction.

**Case 2.**  $s_P$  and  $s_Q$  perform WRN with different indices,  $i_P$  and  $i_Q$ , respectively.

Since  $k \geq 3$ , either  $i_P \neq i_Q + 1 \bmod k$  or  $i_Q \neq i_P + 1 \bmod k$ . Without loss of generality, assume that  $i_Q \neq i_P + 1 \bmod k$ . So the configurations  $C_{s_P s_Q}$  and  $C_{s_Q s_P}$  are indistinguishable for a solo run of  $P$ . However, the identical solo runs of  $P$  from the configurations  $C_{s_P s_Q}$  and  $C_{s_Q s_P}$  decide 0 and 1, respectively, which is a contradiction.

Both cases resulted in a contradiction, and therefore no such algorithm exists. ◀

## 7 Implications

### 7.1 Set Consensus Ratio

A trivial implication of Section 4 is that  $\text{WRN}_k$  objects can solve the  $m$ -set consensus task for  $n$  processes as long as  $\frac{k-1}{k} \leq \frac{m}{n}$  is satisfied. For instance,  $\text{WRN}_3$  objects can be used for implementing (12, 8)-set consensus.

Algorithm 6 describes an implementation of the  $m$ -set consensus task for  $n$  processes using  $\text{WRN}_k$  objects. It uses an array  $W$  of  $\lceil \frac{n}{k} \rceil$  shared  $\text{WRN}_k$  objects, where the process named  $i$ ,  $0 \leq i < n$  invokes the WRN operation of  $W[\lfloor \frac{i}{k} \rfloor]$  with its proposal and the index  $i \bmod k$ . If  $\perp$  is returned, the process decides on its own proposal. Otherwise, it decides on the returned value of the invocation.

Note that Algorithm 6 can be implemented using  $1s\text{WRN}_k$  objects instead of the  $\text{WRN}_k$  objects, since every index is accessed at most once.

► **Lemma 39.** For every  $0 \leq j < \lceil \frac{n}{k} \rceil$ , the set of processes  $\mathcal{P} = \{P_i \mid j \cdot k \leq i < (j+1) \cdot k\}$  solves the  $(k-1)$ -set consensus task using algorithm 6.

**Proof.** This algorithm is similar to Algorithm 2, and since  $|\mathcal{P}| \leq k$ , corollary 9 shows algorithm 6 solves the  $(k-1)$ -set consensus task for  $\mathcal{P}$ . ◀

► **Corollary 40.** Algorithm 6 solves the  $m$ -set consensus task for  $n$  processes.

### 7.2 Infinite Hierarchy

The combination of the results of Sections 4 and 5 imply that  $1s\text{WRN}_k$  objects have the same computational power as  $(k, k-1)$ -set consensus objects, e.g.  $1s\text{WRN}_k$  objects are computationally equivalent to  $(k, k-1)$ -set consensus objects.

The following relationship among set consensus objects is known [1, 16]:

► **Theorem 41.** *Let  $n > k$  and  $m > j$  be positive integers. Then there is a wait-free implementation of an  $(n, k)$ -set consensus object from  $(m, j)$ -set consensus objects and registers in a system of  $n$  or more processes if and only if  $k \geq j$ ,  $\frac{n}{k} \leq \frac{m}{j}$ , and either  $k \geq j \cdot \lceil \frac{n}{m} \rceil$  or  $k \geq j \cdot \lfloor \frac{n}{m} \rfloor + n - m \cdot \lfloor \frac{n}{m} \rfloor$ .*

► **Corollary 42** (Hierarchy of 1sWRN objects). *Let  $k < k'$  be two positive integers. So:*

1.  $1sWRN_k$  cannot be implemented using  $1sWRN_{k'}$  objects and registers.
2.  $1sWRN_{k'}$  can be implemented using  $1sWRN_k$  objects and registers.

This corollary forms an infinite hierarchy among the 1sWRN objects, such that  $1sWRN_{k'}$  objects are considered to have more computational power than  $1sWRN_k$  objects if  $k < k'$ . Since 1sWRN objects have more computational power than simple read-write registers, and cannot solve the consensus task for 2 processes, this hierarchy shows the existence of an infinite number of computational power classes between simple read-write registers and 2-consensus.

## 8 Conclusion

This paper advances our understanding of classification of deterministic shared objects. It was an open question whether there are deterministic objects that are stronger than registers, and yet incapable of solving the consensus task for two processes.

The answer to this question for nondeterministic objects is well known [18]. For the deterministic case, only recently [1] it has been shown that the consensus task alone is not enough for classifying the computational power of deterministic objects. It is suggested that the set consensus task gives a more fine grained granularity for deterministic objects power classification, however the layer of objects under 2-consensus was not discussed.

Our construction shows that set-consensus gives a more fine grained granularity in understanding the computational power of objects, even between atomic read/write registers and 2-consensus. Not only we show the existence of objects between both computational classes, we also provide an infinite hierarchy of computational classes between the two classes, defined by the set-consensus task, using the implications of [8, 9].

Even though we have a better understanding of the behavior of deterministic objects under 2-consensus, our research leaves some open questions. We have shown that for every  $k$ , there is a deterministic object that can solve the  $(k, k - 1)$ -set consensus task. This result is extended to the  $(n, m)$ -set consensus task, where  $\frac{m}{n} \geq \frac{k}{k-1} \geq \frac{2}{3}$ . We do not show the existence of deterministic objects that can solve the  $(n, m)$ -set consensus task where  $\frac{n}{k} < \frac{2}{3}$  without solving the 2-consensus task. More precisely, this paper does not show (or refutes) the existence of a deterministic object that can solve the 2-set consensus task for any number of processes, but is unable to solve the 2-consensus task. These questions remain open.

Finally, although the Consensus Hierarchy is not precise enough to characterize the synchronization power of objects, we may conjecture that a hierarchy based on set-consensus may be precise enough. Chan et al. [13] give an example in which set-consensus powers is not enough to characterize the ability of a deterministic object to solve the  $n$ -SLC problem. However, by definition, the  $n$ -SLC problem is not a problem in the wait-free model. Thus the conjecture that set-consensus is enough to characterize the synchronization power of deterministic shared objects in the wait-free model (in particular, their power to solve tasks wait-free) is still open.

## References

- 1 Yehuda Afek, Faith Ellen, and Eli Gafni. Deterministic objects: Life beyond consensus. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, PODC '16, pages 97–106, New York, NY, USA, 2016. ACM. doi:10.1145/2933057.2933116.
- 2 Yehuda Afek, Eli Gafni, and Adam Morrison. Common2 extended to stacks and unbounded concurrency. *Distributed Computing*, 20(4):239–252, Nov 2007. doi:10.1007/s00446-007-0023-3.
- 3 Yehuda Afek, Eli Gafni, Sergio Rajsbaum, Michel Raynal, and Corentin Travers. Simultaneous consensus tasks: A tighter characterization of set-consensus. In Soma Chaudhuri, Samir R. Das, Himadri S. Paul, and Srikanta Tirthapura, editors, *Distributed Computing and Networking*, pages 331–341, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.
- 4 Yehuda Afek and Michael Merritt. Fast, wait-free (2k-1)-renaming. In *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '99, pages 105–112, New York, NY, USA, 1999. ACM. doi:10.1145/301308.301338.
- 5 Yehuda Afek, Eytan Weisberger, and Hanan Weisman. A completeness theorem for a class of synchronization objects. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Distributed Computing*, PODC '93, pages 159–170, New York, NY, USA, 1993. ACM. doi:10.1145/164051.164071.
- 6 Hagit Attiya and Arie Fouren. Adaptive wait-free algorithms for lattice agreement and renaming (extended abstract). In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '98, pages 277–286, New York, NY, USA, 1998. ACM. doi:10.1145/277697.277749.
- 7 Rida A. Bazzi, Gil Neiger, and Gary L. Peterson. On the use of registers in achieving wait-free consensus. *Distributed Computing*, 10(3):117–127, Mar 1997. doi:10.1007/s004460050029.
- 8 Elizabeth Borowsky. *Capturing the Power of Resiliency and Set Consensus in Distributed Systems*. PhD thesis, University of California in Los Angeles, Los Angeles, CA, USA, 1995. UMI Order No. GAX96-10429.
- 9 Elizabeth Borowsky and Eli Gafni. Generalized flip impossibility result for t-resilient asynchronous computations. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing*, STOC '93, pages 91–100, New York, NY, USA, 1993. ACM. doi:10.1145/167088.167119.
- 10 Elizabeth Borowsky and Eli Gafni. *The implication of the Borowsky-Gafni simulation on the set-consensus hierarchy*. UCLA Computer Science Department, 1993.
- 11 Elizabeth Borowsky, Eli Gafni, and Yehuda Afek. Consensus power makes (some) sense! (extended abstract). In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '94, pages 363–372, New York, NY, USA, 1994. ACM. doi:10.1145/197917.198126.
- 12 David Yu Cheng Chan, Vassos Hadzilacos, and Sam Toueg. On the number of objects with distinct power and the linearizability of set agreement objects. In *31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria*, pages 12:1–12:14, 2017. doi:10.4230/LIPIcs.DISC.2017.12.
- 13 David Yu Cheng Chan, Vassos Hadzilacos, and Sam Toueg. On the classification of deterministic objects via set agreement power. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, PODC '18, pages 71–80, New York, NY, USA, 2018. ACM. doi:10.1145/3212734.3212775.
- 14 Soma Chaudhuri. Agreement is harder than consensus: Set consensus problems in totally asynchronous systems. In *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing*, PODC '90, pages 311–324, New York, NY, USA, 1990. ACM. doi:10.1145/93385.93431.

- 15 Soma Chaudhuri. More choices allow more faults: Set consensus problems in totally asynchronous systems. *Information and Computation*, 105(1):132–158, 1993.
- 16 Soma Chaudhuri and Paul Reiners. Understanding the set consensus partial order using the borowsky-gafni simulation. In Özalp Babaoğlu and Keith Marzullo, editors, *Distributed Algorithms*, pages 362–379, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.
- 17 Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.
- 18 Maurice Herlihy. Impossibility results for asynchronous pram (extended abstract). In *Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '91, pages 327–336, New York, NY, USA, 1991. ACM. doi:10.1145/113379.113409.
- 19 Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13:124–149, January 1991. doi:10.1145/114005.102808.
- 20 Prasad Jayanti. Wait-free computing. In Jean-Michel Hélary and Michel Raynal, editors, *Distributed Algorithms*, pages 19–50, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.
- 21 Leslie Lamport. Solved problems, unsolved problems and non-problems in concurrency. *SIGOPS Oper. Syst. Rev.*, 19(4):34–44, 1985. doi:10.1145/858336.858339.





# NUMASK: High Performance Scalable Skip List for NUMA

**Henry Daly**

Lehigh University, Bethlehem, PA, USA  
hwd219@lehigh.edu

**Ahmed Hassan**

Alexandria University, Alexandria, Egypt  
ahmed.hassan@alexu.edu.eg

**Michael F. Spear**

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

**Roberto Palmieri**

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

---

## Abstract

This paper presents NUMASK, a skip list data structure specifically designed to exploit the characteristics of Non-Uniform Memory Access (NUMA) architectures to improve performance. NUMASK deploys an architecture around a concurrent skip list so that all metadata accesses (e.g., traversals of the skip list index levels) read and write memory blocks allocated in the NUMA zone where the thread is executing. To the best of our knowledge, NUMASK is the first NUMA-aware skip list design that goes beyond merely limiting the performance penalties introduced by NUMA, and leverages the NUMA architecture to outperform state-of-the-art concurrent high-performance implementations. We tested NUMASK on a four-socket server. Its performance scales for both read-intensive and write-intensive workloads (tested up to 160 threads). In write-intensive workload, NUMASK shows speedups over competitors in the range of 2x to 16x.

**2012 ACM Subject Classification** Information systems → Data structures

**Keywords and phrases** Skip list, NUMA, Concurrent Data Structure

**Digital Object Identifier** 10.4230/LIPIcs.DISC.2018.18

**Funding** This material is based upon work supported by the Air Force Office of Scientific Research under award number FA9550-17-1-0367 and by the National Science Foundation under Grant No. CNS-1814974.

**Acknowledgements** Authors would like to thank anonymous reviewers for the insightful comments, Maged Michael and Dave Dice for the early feedback on the paper, and Vincent Gramoli for agreeing to integrate NUMASK into Synchrobench.

## 1 Introduction

Data structures are one of the most fundamental building blocks in modern software. The creation of performance-optimized data structures is a high-value task, both because of intellectual contributions related to algorithms' design and correctness proofs, and because



© Henry Daly and Ahmed Hassan and Michael F. Spear and Roberto Palmieri;  
licensed under Creative Commons License CC-BY

32nd International Symposium on Distributed Computing (DISC 2018).

Editors: Ulrich Schmid and Josef Widder; Article No. 18; pp. 18:1–18:19

Leibniz International Proceedings in Informatics



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

of the impact that even a single data structure can have on the performance of enterprise-level applications. For example, the use of a high-performance non-blocking skip list is *the* fundamental innovation in the MemSQL database [29].

Current and (likely) future generations of enterprise-level computing infrastructures deploy on a hardware design known as Non-Uniform Memory Access (or NUMA) [22, 24], which specifies that memory access latency varies depending on the distance between the processor performing the memory access and the memory chip currently holding the memory location. With NUMA, the memory hierarchy is more complex than before; if a system possesses multiple discrete CPU chips (i.e., physical processors installed on different CPU sockets), each will have faster access to a locally-attached coherent memory and slower (but still cache-coherent) access to the memories attached to other chips. This is mainly because the bandwidth of the hardware channel that connects these multiple chips is limited and its performance is generally poor. As a consequence of these considerations, we can claim that NUMA prefers locality; therefore, applications or systems should be (re)designed with this guideline in mind. Such a claim has been confirmed by a number of recent works [27, 4, 6, 10].

The performance penalty of NUMA architectures has been quantified by many recent efforts [4, 26, 3, 16]. A recurring, although conservative, guideline in those studies is to avoid (if possible) scheduling cooperative threads on different processors. Although this guideline is valid in some applications where there is a clear separation in data access pattern among application threads, it might not be easy to apply in other applications where data is maintained as a set of connected items in a linked data structure. For example, searching for an item usually forces a thread to traverse multiple elements of the data structure in order to reach the target item. Because of this, each operation might produce large traffic on the NUMA interconnection; this traffic is the main reason for degraded performance [9].

Caching will not completely solve the problem either, because concurrent updates mandate refreshing cached locations. From our experience, as we show later in the experimental results in Section 7, the presence of even a few percentage of update operations results in a significant performance drop on NUMA. We conclude that data structures not designed for NUMA do not perform well on modern enterprise-level architectures when concurrent updates mandate refreshing cached locations.

In this paper we present NUMASK, a novel concurrent skip list data structure [20] tailored to a NUMA organization. Unlike existing NUMA-aware solutions for data structures (e.g., [6] see Section 2 for details), our design does not limit parallelism to cope with NUMA; rather, it leverages NUMA characteristics to improve performance. What makes our proposal unique is that its advantages hold even for high update rates and contention. We adhered to the following considerations throughout the development of NUMASK:

- (a) *local* memory accesses (i.e. memory close to the executing thread’s processor) are favored;
- (b) traffic across NUMA zones, often produced by synchronization primitives, is avoided.

In a nutshell, our design produces redundant metadata to be placed on different NUMA zones (which meets requirement (a)) and avoids the need of synchronizing this metadata across NUMA zones (which satisfies requirement (b)). The final design is a data structure that never limits concurrency and at the same time primarily accesses NUMA local memory (in our evaluation study, > 80% of memory accesses are local).

The simple observation that motivated our work is that in a skip list, the actual data resides in the lowest level of the skip list, and the other levels form an index layer whose task is only to accelerate execution of operations. In NUMASK, we exploit this fact in two ways:

- We define independent index layers (one per NUMA zone) for the skip list. Each operation traverses the index layer that is local to the thread that executes it. This way, operations do not need to traverse the interconnection between NUMA zones during the index layer

traversal. Importantly, we do not keep these index layers consistent with each other; we allow them to be different. In fact, having different index layers in different NUMA zones does not affect correctness because the actual data (which resides in the lowest level of the skip list) is still synchronized.

- We isolate updates on the index layers in separate (per-NUMA) helper threads instead of performing those updates in the critical path of the insert/remove operations. Although this isolation may delay the synchronization of the index layers, the (probabilistic) logarithmic complexity of the skip list operations can be eventually maintained even with lazy index layer updates [18].

Former designs [8, 12] proposed the isolation of index layer updates in helper threads, but none of them defined per-NUMA index layers. That is why in those proposals, the NUMA overhead is still significant due to traversing a single index layer. NUMASK inherits the idea of applying replication to data structure in order to improve its performance in NUMA architectures, as done by [6], but NUMASK targets only metadata and updates such metadata lazily.

We implement NUMASK in C++ and integrate into Sychrobench [17], a comprehensive suite of data structures implemented in the same optimized software infrastructure. The implementation of NUMASK has been enriched with specific optimizations, such as an efficient NUMA memory allocator, developed on top of `libnuma` [1], to avoid bottleneck. We compare the performance of NUMASK with three state-of-the-art concurrent skip lists: Fraser [15], No Hotspot [8], and Rotating Skip List [12]. Performance shows up to 16x speed up for write workloads and improvements up to 40% in read-intensive workloads. In summary, NUMASK hits an important performance goal: in low-contention workloads, NUMASK adds no overhead to the high-performance concurrent data structures; and in high-contention workloads, NUMASK outperforms all other competitors and keeps scaling (we tested up to 160 threads) while other competitors stop earlier (at 64 threads in our experiments).

NUMASK is part of the core release of Sychrobench [17] available at <https://github.com/gramoli/synchrobench>.

## 2 Related Work

Many concurrent variants of the original sequential skip list [28] data structure have been proposed in the last decade. Some of them are blocking [6, 21, 19, 20], and others are non-blocking [14, 15, 8, 12]. Among the non-blocking designs, which often demonstrate improved performance over blocking designs [17], Fraser [15] proposed the use of a CAS primitive to create a non-blocking skip list. Crain et al. [8] proposed a contention friendly skip list, called No Hotspot, which serves as the foundation of our NUMASK design. The main innovation in No Hotspot is that it isolates bookkeeping operations (e.g., updating index levels) in a helper thread. The rotating skip list was proposed by Dick et al. [12] to further improve No Hotspot's poor locality of references in order to reduce cache misses. However, none of the above designs is optimized for NUMA architectures and thus they all generate significant NUMA interconnect traffic.

Recent uses of skip lists include ordered maps, priority queues, heaps, and database indexes (e.g., [29]). The NUMASK design can be applied to these data structures, improving their performance through data and index layer separation when deployed in NUMA architectures.

The impact of NUMA organization on the performance of software components (e.g. data structures and thread synchronization) is an important topic. Interestingly, the last decade saw the proposal of many NUMA-aware building blocks to improve application performance.

Examples include NUMA-aware lock implementations [11, 5], thread placement policy [23], and smart data arrays [27]. Although helpful, the applicability of these components in linked data structures is limited due to the memory organization required by data structures in order to implement their operations while preserving the asymptotic complexity.

Few specialized NUMA-aware techniques for data structures have been proposed [6, 4]. The most relevant to NUMASK is the method proposed by Calciu et al. [6], wherein data structures can be made NUMA aware. Using a technique called NR (Node Replication), replicas are created across NUMA zones. However, replica synchronization across zones forces significant NUMA interconnect traffic. In fact, since synchronous updates of the whole data structure (including the searching layer) are assumed, the authors needed a shared synchronization log to save and replay update operations on each replica of the data structure. Moreover, a read operation would wait for the replay of pending updates in order to guarantee its linearization. On the bright side, this approach is a general technique that applies to different data structure designs, whereas NUMASK can exploit specific optimizations because its goal is to provide a high-performance NUMA-aware skip list. In fact, NUMASK relaxes the need of synchronizing different index layer instances; thus, it does not suffer from the above overheads which impede scalability.

Brown et al. [4] proposed a simple design, effective in small-scale deployments, that maintains the entire index layer in a single NUMA zone. This solution's pitfall is its limited parallelism. For operations to access NUMA-local memory addresses, either the application thread's execution must be migrated to the processor attached to the desired NUMA zone, or the operation must be delegated to one or more serving threads in the target NUMA zone. This inherently limits parallelism to a single processor's maximum computing capability. Our new design overcomes all the above limitations: all application and background threads operate primarily on NUMA-local memory and perform a negligible number of NUMA-remote accesses, eliminating the need for migration or delegation.

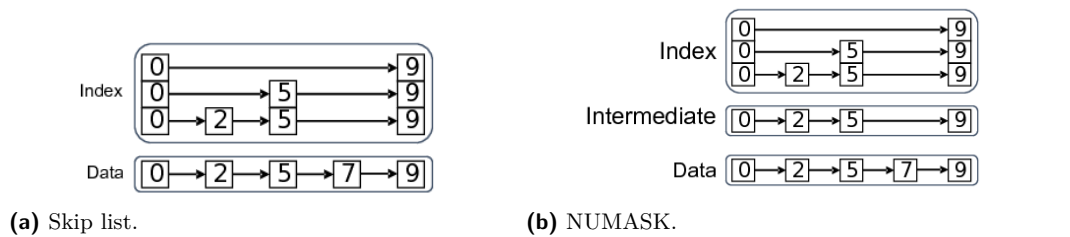
Orthogonal to our NUMASK approach, in [27, 25] partitioning techniques have been used for targeting the hardware organization of NUMA architectures to improve the performance of array representations [27] and in-memory transaction processing [25].

### 3 Terminology, NUMA & Linked Data Structures

In NUMA, each (multicore) CPU is physically connected to a partition of the whole memory available in the system, called a NUMA zone. A hardware interconnection exists between NUMA zones (the NUMA interconnection). The hardware provides applications (including the OS) with the abstraction of a single consistent global memory address space; therefore, threads can access the entire memory range in a manner that is oblivious to the NUMA zone in which each virtual address resides. However, this transparency comes with performance costs associated with having an interconnection between NUMA zones.

This interconnection has limited bandwidth, is slow to traverse, and saturates when many threads attempt to use it. Thus, if a thread executing on one CPU accesses a memory location stored in a NUMA zone physically connected with another CPU (called a *remote* NUMA zone hereafter), it incurs a latency that is significantly higher than the latency needed to access a memory location in the NUMA zone connected with the CPU where the thread executes (called *local* NUMA zone hereafter). In short, we use the term NUMA-local memory when the memory is in the local NUMA zone and the term NUMA-remote memory otherwise.

Linked data structures are particularly affected by the memory latency variation introduced by NUMA. This is because traversing the data structure through pointers can easily lead threads to access memory locations physically maintained in remote NUMA zones.



■ **Figure 1** Separation of layers in base skip list Vs. NUMASK. In 1b, the Intermediate layer has not been updated with key 7 yet.

NUMA-aware memory allocation (e.g., `libnuma` [1], which is supported by most Operating System distributions) cannot eliminate this problem because even if threads allocate memory in their local NUMA zone, they might still need to traverse many other nodes to accomplish their operation, and these nodes might be added by threads running on remote NUMA zones.

#### 4 NUMASK: A Concurrent Skip List Designed for NUMA

In this section we illustrate the design of NUMASK. In order to retain decades of high performance skip list results, NUMASK deploys a modular design that re-uses the fundamental operations of an existing concurrent skip-list and wraps these operations around a NUMA-aware architecture. The result is a data structure whose performance improves upon the selected concurrent skip list implementation when deployed on NUMA architectures. Another benefit of our modular design is that the correctness of the resulting NUMA-aware skip list is easy to prove since the wrapping architecture does not modify the core operations of the selected concurrent skip list implementation, which is assumed to be correct.

In the rest of the paper we will use the term *base skip list* to indicate an implementation of a skip list that is wrapped (and improved) by the NUMASK architecture. The *base skip list* is a concurrent skip list whose API are *insert*, *remove*, and *contains* operations, with their default signatures [20]. The only requirement we add to this concurrent skip list is that bookkeeping operations (e.g., updating the searching layers and physical removal of logically deleted nodes) are decoupled from the critical path of the data structure operations (i.e., *insert/remove/contains*) and executed lazily by a *helper* thread. It is worth noting that the features we require in the base skip list have been successfully deployed in many existing data structure implementations [18, 7, 12] and do not diminish the applicability of our proposal.

In this paper we use Crain et al.'s No Hotspot skip list [8] as the *base skip list* because it defines a helper thread responsible for updating the skip list, and it is one of the state-of-the-art concurrent skip list implementations (as studied in [17]). For completeness, it is worth mentioning that No Hotspot, and thus our NUMASK skip list implementation, is lock-free.

All skip list implementations share one key observation that motivates our design: elements in the data structure, representing the abstract state of the skip list, are reached through an index layer. This index layer is composed of metadata that does not belong to the abstract state of the data structure, and which is used to improve performance by minimizing the number of traversed nodes. Leveraging the above observation, we can split the memory space used by a skip list into a *data layer*, which stores the abstract state of the data structure, and an *index layer*, which includes the metadata exploited to reach the data layer. Figure 1a illustrates this separation.

Managing the data layer and index layer independently is the crucial intuition behind the NUMASK design, for it exploits the different consistency requirements they have to improve performance in NUMA architectures. None of the existing designs of NUMA-aware data structures, when applied to skip lists (e.g., [6]), accounts for such separation.

In a nutshell, in order to improve performance in NUMA architectures, the primary design choice of NUMASK is to create as many index layers as the number of NUMA zones in the system. These index layers are not updated immediately after successful insert/remove operations. Instead, they will be updated independently to avoid (unnecessary) synchronization and traffic on the NUMA interconnection. The ultimate goal of having NUMA-local index layers is to let operations on the data structure only access NUMA-local memory before reaching the data layer. Once there, the (probabilistic) logarithmic complexity of the skip list allows for the traversal of only few nodes in the data layer before finalizing the operation. We empirically demonstrate that traversing these few nodes (possibly NUMA-remote) does not have a significant impact on performance. NUMASK accomplishes the above goal by deploying the following design around a *base skip list*.

#### 4.1 Per-NUMA zone index layers

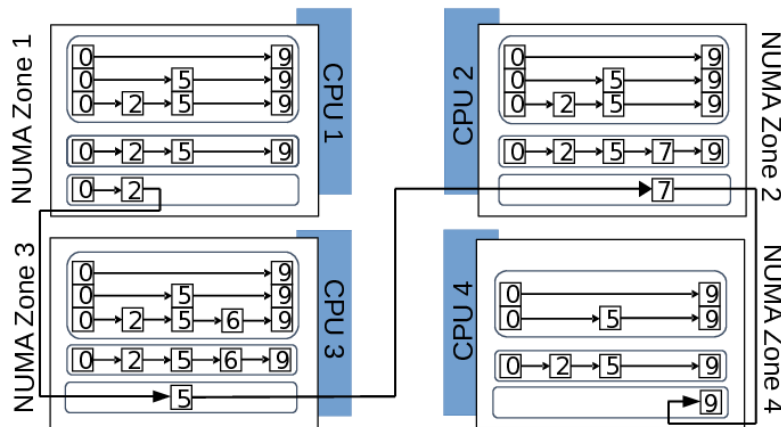
In skip lists, most of the traversed nodes exist in the index layer; therefore, creating as many index layers as the number of NUMA zones allows application threads to perform mostly NUMA-local accesses. Given that the *base skip list* defers updates to the index layer to a helper thread, having multiple independent indexing layers entails the need of deploying the same amount of helper threads (one per NUMA zone) responsible for their management. Consequently, helper threads will also access NUMA-local memory.

#### 4.2 Per-NUMA zone intermediate layers

Decisions on how to update the index layer usually depend upon the current composition of the data layer. That is why the aforementioned per-NUMA zone helper threads, responsible for updating each instance of the index layers, would have to traverse the data layer nodes in order to decide whether to apply certain modifications (e.g., increasing or lowering a level of a certain node in the data layer) or to leave the index layer instance unaltered. Since the traversed data layer nodes are not necessarily NUMA-local, this can produce excessive NUMA-remote accesses and generate significant traffic on the NUMA interconnection, which is the main source of performance degradation in NUMA.

Because in NUMASK we aim at eliminating any NUMA-remote accesses while updating the index layer instances, we create a NUMA-local view of the data layer, which we name the *intermediate* layer. Creating multiple intermediate layers, one per index layer instance, allows helper threads to fully operate on NUMA-local memory. Logically, the intermediate layer is placed in between the index layer and the data layer. With respect to the index layer, the intermediate layer has the same goal as the *base skip list* data layer, meaning it serves as a knowledge base for the helper thread(s) to update the index layer instance(s).

The peculiarity of the intermediate layer is that it need not be an exact replica of the data layer (e.g., it is enough to be eventually synchronized with the data layer). In fact, any inaccuracy in an index layer instance, which could happen due to a temporarily out-dated intermediate layer, affects only the skip list performance and not its correctness. This is the same rationale that led previous skip list designs [8, 12, 17] to lazily update the index layer. Relaxed constraints on the intermediate layer composition enable its NUMA distribution.



■ **Figure 2** NUMASK deployed on a server with four sockets and four NUMA zones. The four instances of the index and intermediate layer are independent, and the data layer is scattered across available memory. The abstract state of the data structure contains the following keys:  $\{0;2;5;7;9\}$ .

In Figure 1b we show a simple example of NUMASK. Here the abstract state of the skip list is the same as Figure 1a; however, the intermediate layer has not been updated with the element with key 7. This is a plausible case in our design, meaning that the *insert(7)* operation result has not yet been propagated to the intermediate layer. We can easily see that the index layer remains the same as the skip list in Figure 1a. The modifications made by *insert(7)* will eventually be propagated to the intermediate layer using a technique (shown below) that does not increase the duration of the actual data structure operation.

### 4.3 Propagation of Data Layer Modifications.

The intermediate layer instances need to be periodically updated to reflect the content of the data layer. A naïve way to do this follows: at the end of each update operation (i.e., insert/delete), necessary information is stored in an intermediary data structure (e.g., a queue), and each per-NUMA helper thread later loads this information and updates its local intermediate layer. However, this naïve approach leads to one major drawback: it requires synchronization and memory allocation overhead on the data structure’s critical path.

To remove this overhead from the application threads, NUMASK assigns a new helper thread the task of updating the intermediate layer instances. This thread operates at predefined intervals and iterates over the data layer. Every time it finds a node that has been modified (i.e., inserted or logically removed), it propagates this modification to all instances.

It is worth noting that this new helper thread does generate traffic on the NUMA-interconnection. However, the impact of this traffic on the data structure performance is minimal given that it does not operate frequently. Also, thanks to our optimizations in the index layers, the number of NUMA-remote accesses is already low (<15% in our experiments). Thus, the NUMA-interconnection is expected not to be saturated; therefore, this helper thread will not cause significant delay.

### 4.4 Example of NUMASK deployment

In Figure 2 we deployed NUMASK on a server with 4 processor sockets and 4 NUMA zones. In the example, the abstract state of the skip list is  $\{0;2;5;7;9\}$ . By looking at the data layer we assume that the elements 0 and 2 have been inserted by an application thread executing



on CPU1, element 5 by a thread on CPU3, and so on. Each NUMA zone has its own intermediate and index layer instance. The composition of the different intermediate layer instances is different because the data layer modifications are not propagated at the same time to all intermediate layer instances. For example, in the figure the element 6 has been removed, but the intermediate layer of NUMA zone 3 still has not applied this modification. Also, in the figure the four index layer instances differ from each other since helper threads work independently and do not proceed synchronously.

#### 4.5 Design Trade-offs

The design of NUMASK presents different trade-offs with respect to the space and time needed to handle its index and intermediate layers, including tuning the configuration associated with the deployed helper threads. These trade-offs are briefly discussed below.

NUMASK introduces space overhead due to the presence of multiple instances of both index layer and intermediate layer. This overhead is proportional to the number of NUMA zones in the system; however it does not increase with the number of application threads. Moreover, as we will detail later, the synchronization overhead to maintain (i.e., traverse and update) this extra space is limited. Finally, it is important to note that, in cases where space utilization is crucial, some optimization can be added to NUMASK to control such utilization. For example, a probabilistic policy can be added to the data layer propagation process. This policy might aim at selecting only some operation made by application threads, rather than all, to be propagated to the different intermediate layer instances.

Another trade off involves the helper threads frequency of operation. Tuning the backoff time after each iteration of the helper threads might affect the overall performance of NUMASK. One viable solution towards a configuration that is effective in multiple scenarios is to use an adaptive technique, similar to the one adopted in [18], in which the application workload is monitored and backoff time is adjusted accordingly.

## 5 NUMASK: Protocol Details

In this section we show the algorithmic details of NUMASK. The pseudo-code describing NUMASK is reported in Algorithms 2 and 3. To clarify the presentation, we abstract a base skip list in Algorithm 1. By leveraging this abstraction, we can avoid listing the details of core operations on the skip list (i.e., traversal, modification to data and index layer, logical and physical removal of elements) and focus on our NUMA-aware modifications. Algorithms 2 and 3 include calls to procedures defined in Algorithm 1. All the low-level details of our implementation are public and available in Synchronbench.

Algorithm 1 abstracts the base skip list as two procedures: **Base-Operation** and **Base-Helper**. **Base-Operation** is the handler for the three different types of data structure operations, namely **insert**, **remove**, and **contains**. Each of these operations is split into **Base-Traversal** and **Base-DoOperation** sub-procedures. The former traverses the index layer and returns a pointer to some data layer node where the operation should act. The latter works entirely on the data layer and applies the invoked operation (e.g., if the operation is an insert, the node is physically inserted in the data layer). **Base-Helper** periodically calls **Base-UpdateIndex** for updating the skip list index layer and performing physical removals.

As mentioned before, in our experiments we selected No Hotspot as the underlying base skip list implementation. The details of how No Hotspot implements **Base-Traversal**, **Base-DoOperation**, and **Base-UpdateIndex** can be found in [8].



---

**Algorithm 1** Abstract Base Skip List.

---

```

1: Global Variable: indexSen ▷ indexSen = sentinel node of index layer
2: procedure BASE-OPERATION(Type t, Element el) ▷ t = Insert/Remove/Contains
3:   Node n = Base-Traversal(indexSen,el.key); ▷ n is the node with the closest key value less
   than or equal to the desired node
4:   boolean res = Base-DoOperation(t,el,n);
5:   return res;
6: end procedure

7: procedure BASE-HELPER(Node s)
8:   while true do
9:     Base-UpdateIndex(s); ▷ This procedure updates the index layer starting from the
   sentinel node s
10:    ▷ In the base skip list, s is the sentinel node of the lowest level of the skip list
11:   end while
12: end procedure

```

---

## 5.1 NUMASK: Data Structure Operations

NUMASK's **Insert**, **Remove**, and **Contains** operations (Algorithm 2) can be summarized in the following steps: *i*) each operation traverses the local index layer instance until it retrieves a pointer to a node in the local intermediate layer; *ii*) this intermediate layer node is used as an indirection to reach a pointer to a data layer node; *iii*) this pointer is then used to perform the actual operation on the data layer. Importantly, the operations terminate right after updating the data layer, since all further updates in both intermediate and index layers are delegated to the helper threads (as detailed in the next two subsections).

The details of Algorithm 2 are as follows. In typical skip lists, index layer traversal starts from a known sentinel node. In NUMASK, each NUMA zone has its own index layer instance and therefore its own sentinel node as well (Algorithm 2:2). When a NUMASK traversal is invoked (Algorithm 2:18), the local thread starts from the sentinel node of the local NUMA zone. From this point, all memory accesses of `NUMASK_Traversal` will be NUMA-local. The traversal operates similar to that of the base skip list: it moves to a node on its right in the same level (using the *next* field) as long as its key is less than or equal to the target key (say  $k$ ), and it moves to the next lower index level (using the *down* field) otherwise. If there is no lower index level to traverse, the traversal exits by returning the pointer to the node in the intermediate layer. Each node in the intermediate layer has a (*down*) pointer to its respective data layer node, from which `Base-DoOperation` can begin.

`Base-DoOperation` operates similar to the base skip list: The data layer is traversed from the pointer reached by the intermediate layer node until either a node with a greater key is found or the list ends. After that, the operation completes based on its type. If it is a **contains** operation, it checks whether the node's key matches  $k$  or not. The **insert** and **remove** operations use `Compare-And-Swap` for non-blocking updates (details of how No Hotspot, and thus NUMASK, accomplishes that can be found in [8]).

An important task assigned to `NUMASK_DoOperation` is to update the node's *status* field upon a successful write operation. Setting this field to 1 (respectively 2) indicates to helper threads that the node is newly inserted (respectively removed), and this insertion (respectively removal) is not yet propagated to the intermediate and index layers. To simplify the pseudo-code, we exclude this assignment of the status field, replacing it with a comment in Algorithm 2:23.

---

**Algorithm 2** NUMASK: Skip List Operations.

---

```

1: Global Variable:
2: Node indexSents[MaxNumaZones] ▷ Array of index layer sentinel nodes, one per NUMA zone
3: Node interSents[MaxNumaZones] ▷ Array of intermediate layer sentinel nodes, one per NUMA
   zone
4: Node dataSent                                     ▷ data layer sentinel node
5: Queue update-queues[MaxNumaZones]                ▷ Queue utilized for updating the MaxNumaZones
   intermediate layers

6: Node: a struct with fields
7:   next                                             ▷ Pointer to next node in the list
8:   down                                             ▷ Pointer to the node in the level below
9:   status                                           ▷ Up to date = 0, recently added = 1, recently removed = 2
10:  level                                           ▷ The height of the tallest tower in the index layer
11:  deleted                                         ▷ Indicates if node is logically deleted

12: procedure NUMASK_OPERATION(Type t, Element el)
13:   Node intermediate_node = NUMASK_Traversal(getCurrentNUMAZone(), el.key);
14:   Node data_node = intermediate_node.down;
15:   boolean result = NUMASK_DoOperation(t, el, data_node);
16:   return result;
17: end procedure

18: procedure NUMASK_TRAVERSAL(int zone, Key k) ▷ This procedure traverses the index
   layer associated with the local NUMA zone and returns a node in the intermediate layer
19:   Node n = Base-Traversal(indexSents[zone], k);
20:   return n
21: end procedure

22: procedure NUMASK_DOOPERATION(Type t, Element el, Node n)
23:   boolean result = Base-DoOperation(t, el, n); ▷ If successful, DoOperation sets the altered
   node's status
24:   return result;
25: end procedure

```

---

## 5.2 Data-Layer-Helper

In NUMASK, we create a single Data-Layer-Helper thread that periodically traverses the data layer in order to accomplish two objectives: *i*) it is responsible for feeding the different intermediate layer instances with the results of successful update operations on the data layer, and *ii*) it attempts to physically remove any logically-deleted nodes of the data layer.

In order to accomplish *i*), the NUMASK design provides each intermediate layer instance with a single-producer/single-consumer queue (Algorithm 2: 5). As a consequence of this decision, there are as many queues as NUMA zones in NUMASK. The producer for all the queues is the same: the Data-Layer-Helper thread; while each queue has a different consumer: the Per-NUMA-Helper thread running in the queue's NUMA zone (detailed in the next subsection). We implemented these queues similar to the Vyukov SPSC queue [30].

The above queues are used to synchronize the data layer with intermediate layers as follows: when the Data-Layer-Helper thread traverses the data layer, each node's status field is checked to see if it is nonzero (which means it was recently inserted/removed); if so, it is added to the queue of each NUMA zone (Algorithm 3: 6) and its status field is reset to zero (to indicate that it is now up to date).

**Algorithm 3** NUMASK: Updating Metadata.

---

```

1: procedure DATA-LAYER-HELPER      ▷ This procedure propagates recently altered nodes to
   intermediate layers
2:   while true do
3:     Node curr = dataSent.next;
4:     while curr != NULL do
5:       if curr.status != 0 then
6:         Add-Job-To-Queues(curr);
7:         curr.status = 0;
8:       else
9:         if curr.level == 0 && curr.deleted then ▷ If curr is logically deleted and there is
   no tower above it in any index layer
10:          remove(curr);
11:        end if
12:      end if
13:      curr = curr.next;
14:    end while
15:  end while
16: end procedure

17: procedure PER-NUMA-HELPER(int local_zone)
18:   while true do
19:     Update-Intermediate-Layer(local_zone)
20:     Base-UpdateIndex(interSents[local_zone]); ▷ UpdateIndex is assumed to update the level
   field of nodes in the data and intermediate layer, when needed
21:   end while
22: end procedure

23: procedure ADD-JOB-TO-QUEUES(Node node)
24:   for i = 0 to MaxNumaZones do
25:     update-queues[i].push(node);
26:   end for
27: end procedure

28: procedure UPDATE-INTERMEDIATE-LAYER(int z)  ▷ This function updates the intermediate
   layer of zone z
29:   Node sentinel = indexSents[z];
30:   while update-queues[z] is not empty do
31:     Node updatedNode = update-queue[z].pop();
32:     Node intermediate_node = NUMASK_Traversal(sentinel, updatedNode.key);
33:     if updatedNode.status == 1 then
34:       Node local-node = NUMA_alloc(updatedNode); ▷ NUMA-aware memory allocator
35:       NUMASK_Operation(INSERT, local-node, intermediate_node);
36:     else
37:       NUMASK_Operation(REMOVE, updatedNode, intermediate_node);
38:     end if
39:   end while
40: end procedure

```

---

In order to accomplish *ii*), the algorithm checks each node to see if it is logically deleted. If so, then it becomes a candidate to be physically removed. As in No Hotspot (as well as other concurrent skip lists), unlinking a node from the data layer can be done only if no tower above it is present in the index layer. However, since NUMASK deploys multiple index layer instances, the condition for physically removing one node is that no tower above it is present in any index layer instance. Verifying this condition is simple: each node in the data layer has a field named *level*. If the traversed node's *level* equals zero and it is logically deleted (Algorithm 3: 9), then the Data-Layer-Helper will proceed with its physical removal. In the next subsection we discuss how to update this *level* field.

By offloading the above two operations to a dedicated thread, the critical path of the application (`NUMASK_Operation`) is minimized. Note that populating the queues, which is required to update the intermediate layers (and therefore the index layers), entails an additional memory allocation overhead. This memory allocation could have been a dominant cost in the operation's critical path if we did not offload it to a separate helper thread.

A positive side effect of our dedicated Data-Layer-Helper thread is that while the thread traverses the data layer, it reloads the cache of the processor on which it is executing, which increases cache hits for application threads that access the data layer. We exploit this idea further by rotating iterations of the Data-Layer-Helper thread between different NUMA zones. This way, caches in different NUMA zones (especially the L3 caches) are evenly refreshed. This process of refreshing caches is particularly effective when the data structure is not large; otherwise the number of elements evicted from cache might be large.

### 5.3 Per-NUMA-Helper

The role of Per-NUMA-Helper is to keep the index and intermediate layer of one NUMA zone updated. Consequently, NUMASK deploys one Per-NUMA-Helper thread per NUMA zone. Each iteration of the Per-NUMA-Helper thread performs two steps. First, it updates the local intermediate layer using the information contained in the queue of its NUMA zone (Algorithm 3:28). Second, it applies any needed modification to the local index layer.

The `Update-Intermediate-Layer` procedure (Algorithm 3:28) is responsible for achieving the first step. In this procedure, the Per-NUMA-Helper thread fetches jobs from the queue in the local NUMA zone and applies them to the local intermediate layer. To do that, Per-NUMA-Helper calls `NUMASK-Traversal` to reach the interested location of the local intermediate layer in logarithmic time. After that, the intermediate layer instance is updated by simply calling `NUMASK-Operation` using the intermediate node pointer returned by `NUMASK-Traversal`.

A critical low-level operation that happens during the `Update-Intermediate-Layer` procedure is the memory allocation of new nodes to be added to the local intermediate layer (Algorithm 3:34). It is required for all memory allocations by each Per-NUMA-Helper thread to be NUMA-local. Otherwise subsequent invocations of `NUMASK-Traversal` are not guaranteed to access entirely NUMA-local memory. In this regard, we tested multiple thread-local [2, 13] and NUMA-aware [1] allocators, but their overhead slowed performance. To deal with this problem, we developed a simple NUMA-aware memory allocator to serve memory allocation requests from Per-NUMA-Helper (see Section 6 for more details).

Once the local intermediate layer is updated, the procedure `Base-UpdateIndex` is called to update the index layer. In our implementation, inspired by No Hotspot, this procedure handles the raising and lowering of towers based on the composition of the intermediate layer, and it also handles removing any logically deleted nodes. First, the helper thread iterates over the intermediate layer, physically removing any nodes marked for deletion without any towers above (similar to what is done to the data layer nodes in `Data-Layer-Helper`). After that and if necessary, towers are raised or lowered to maintain the logarithmic complexity

of the index layer traversals. When a tower is entirely removed in an index layer instance, the Per-NUMA-Helper thread accesses the linked node to the data layer and decrements its *level* field. Although changing the status field in such cases entails a NUMA-remote access, it is not a frequent operation, and thus it has a negligible impact on performance.

## 5.4 Correctness Arguments

One of the advantages of NUMASK's design is its ability to reuse already-implemented basic operations to manipulate the data (and not metadata) of the data structure. None of our modifications needs to address how to insert or remove a node in the skip list data layer. Even the basic skip list traversal need not be modified.

Such a design makes it possible to integrate the NUMASK approach into other skip list implementations without affecting the overall correctness. This is noticeable by looking at how in Algorithms 2 and 3 we invoke procedures from Algorithm 1. In summary, if the base skip list is correct, then NUMASK will preserve such correctness.

## 6 NUMASK Optimization

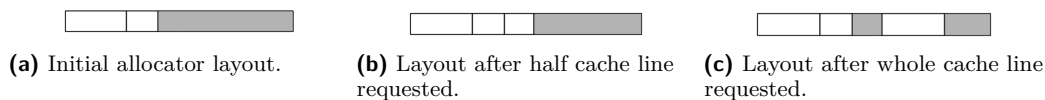
**Custom NUMA-aware Memory Allocator.** NUMASK requires a mechanism to allocate memory in a thread's local NUMA zone. Without this, the proposed architecture would not be beneficial, as application and helper threads would frequently access NUMA-remote memory. Existing NUMA-aware memory allocators (e.g., *libnuma*) repeatedly interact with the operating system in order to retrieve NUMA-local memory. These interactions introduce a noticeable latency. After trying other memory allocators (e.g., [2, 13]), we decided to address our problem by developing a custom linear allocator to support the NUMASK design. To the best of our knowledge, this is the fastest design for memory allocation that fits our software architecture; it is simple yet effective.

Our NUMA allocator is used to serve allocation requests produced by Per-NUMA-Helper, therefore we deploy as many instances of our allocator as the number of Per-NUMA-Helper threads. Importantly, each of these allocator instances serves only one Per-NUMA-Helper thread; therefore, each allocator instance can be sequential (not concurrent).

A linear (or monotonic) allocator consists of a fixed-size memory buffer allocated upon initialization and an internal offset to the beginning of the buffer's free space. Allocation requests increment the buffer offset by the size of the request and return the old value; thus requests are served in constant time without overhead, making the allocator fast.

Our allocator consists of a basic linear allocator plus three additions to fit our needs. The first addition is to allow the allocator to allocate new buffers (linear allocators usually do not reallocate memory). The second addition is to allocate the buffer in a specific NUMA zone, so that all the returned memory addresses reside in the same NUMA zone. With that, intermediate and index layers are formed of NUMA-local memory.

The final addition to our allocator deals with request alignment. Since the allocator is only used to create index and intermediate nodes, and their sizes are less than and greater than a half cache line, respectively, the requests are automatically aligned to either a half or whole cache line. The allocator keeps track of the previous request's alignment internally and aligns the current request based on the previous alignment and the size of the current request. This internal bookkeeping allows the allocator to fit two index nodes in a cache line, which in turn results in faster index traversal, for two nodes in the same cache line will likely be near each other in the index layer, thus reducing necessary memory accesses.



■ **Figure 3** Cache alignment scheme of our allocator. Grey blocks are free space; small white blocks are half cache line; large white blocks are whole cache line.

Figure 3 details how the allocator aligns requests in different scenarios. The example begins in Figure 3a; the previous two requests resulted in a whole cache-line alignment and a half cache-line alignment. Depending on the next request, the allocator could result in two separate layouts. If the next request is an index node (size less than half a cache-line), the allocator can squeeze it in the half cache-line free space. Figure 3b shows the result in this case. However, if an intermediate node is the next memory allocation, the allocator will move the offset to the beginning of the next cache-line to keep the intermediate node from spilling over two cache lines. Figure 3c depicts this. Note that the free space skipped over in Figure 3c will not be used.

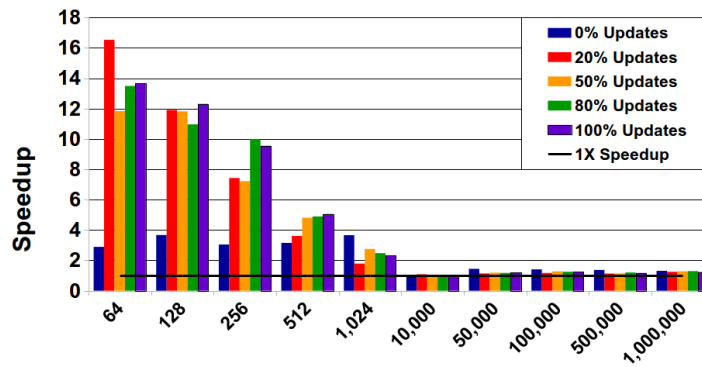
**Avoiding Synchronization When Updating Intermediate Layer.** In Section 5.3, we discussed how each Per-NUMA-Helper thread updates the local intermediate layer. In the pseudo-code we do that by invoking `NUMASK-Operation`, which uses synchronization primitives, since it is the same function used by application threads to operate on the data layer. This task can be changed to let Per-NUMA-Helper modify the intermediate layer without any atomic operations as follows. In order to make updates on an intermediate layer instance synchronization-free, we need to disallow `NUMASK-Operation` from using the intermediate layer to access the data layer (see Algorithm 2:14). To do so, in our implementation we store the pointer to the data layer directly in the index nodes so that application threads never need to access the intermediate layer.

## 7 Evaluation

We implemented NUMASK in C++, and integrated it into Synchronbench [17], a benchmark suite for concurrent data structures. In addition to providing a common software architecture to configure and test different data structure implementations, Synchronbench already implements many state-of-the-art high performance solutions that we used to compare against NUMASK. Specifically, we selected three concurrent skip list implementations: No Hotspot [8], Fraser [15], and Rotating skip list [12]. We also included a sequential skip list implementation [17]. As specified earlier in the paper, NUMASK has been built using No Hotspot as a base skip list implementation for two reasons: it is among the fastest concurrent skip lists of which we are aware, and it alleviates contention by deferring index layer updates.

Our testbed consists of a server with 4 Intel Xeon Platinum 8160 processors (2.1GHz, 24/48 cores/threads per CPU). The machine provides 192 hardware threads. There are 4 sockets hosting the 4 processors, via 4 NUMA zones (one per socket), and 768 GB of memory. In our experiments we ran up to 160 application threads (the actual number of executing threads is higher because of the helper threads used by each competitor) to leave enough resources to the operating system to execute without creating bottlenecks. In our experiments we distribute application threads evenly across NUMA zones.

The workloads we use to test competitors perform insert/remove/contains operations. Note that in order to keep the size of the data structure consistent, during removal the application attempts to pick elements that have previously been inserted successfully. Each



■ **Figure 4** Speedup NUMASK over No Hotspot varying data structure size.

test has a warm-up phase where the skip list is populated and the index is built. This phase is also used to fill out L1/L2/L3 caches. After that, the application runs for 10 seconds while collecting statistics. In the experiments we use a range of key elements that is twice the data structure size; and all elements have integer keys. All results are averages of five test runs.

Before showing the throughput of all competitors, we report two plots that summarize the advantages of NUMASK over the base skip list, which is No Hotspot in our case. Figure 4 demonstrates the speedup of NUMASK over No Hotspot by varying the initial size of the data structure, in the range 64 to 1M elements. To improve clarity, a line is drawn to show when speedup equals 1. We test different percentages of update operations and we record the value for the best performance among all thread ranges. Although for clarity we cannot include the number of threads corresponding to each data point in the plot, it is worth noting that, in our evaluation settings, NUMASK is most effective when the number of threads exceeds 64, as it will be clear analyzing Figure 6. As a result, for all data points in Figure 4, the number of application threads is always in the range of 64 to 160.

NUMASK’s speedup grows significantly when the data structure size decreases. This is mostly due to its capability of exploiting NUMA-local accesses and leveraging cache locality. In fact, with sizes less than 10k elements, most of the data structure will likely fit in processors’ caches, but the presence of updates forces frequent cache refreshing. This refreshing requires loading memory locations from main memory. In No Hotspot, this is likely to be in a remote NUMA zone given that the machine has 4 NUMA zones. However, NUMASK was designed to keep most of the needed memory locations in the local NUMA zone. This is also confirmed by the result using 0% updates; here the speed up is significantly less than in write-intensive workloads because both competitors can benefit from cache locality. Considering 50% updates and 128 elements NUMASK is 11x faster than No Hotspot; and at 100K elements NUMASK is 27% faster. Interestingly, the plots in Figures 6g-6i, meaning when the data structure size is set at 100k, show how NUMASK’s performance does not degrade with respect to competitors. In these cases, the most dominant cost for all is poor cache locality, which brings down performance.

Figure 5 shows the key reason for the performance improvement of NUMASK: its NUMA-local accesses. To collect statistics, we monitored memory accesses performed by application threads and contrasted the application thread’s local NUMA zone with the NUMA zone in which the memory location resides. Here the initial size of the data structure is 100K, and we configured the system to run with 4 and 128 application threads. No Hotspot hovers around 25%, which is the immediate consequence of having uniform distribution of data structure accesses and 4 NUMA zones; NUMASK is around 90% because of its NUMA-aware

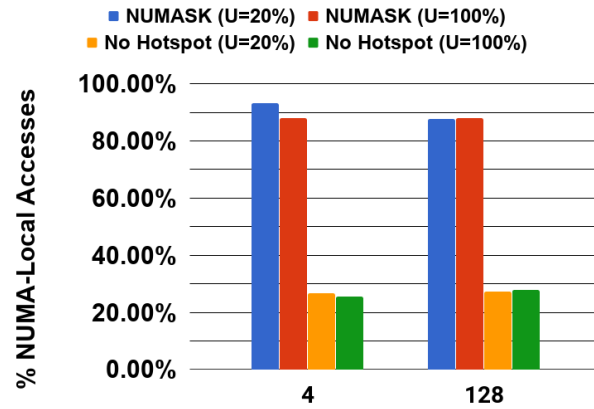


Figure 5 NUMA-local accesses in NUMASK and No Hotspot using {4,128} application threads.

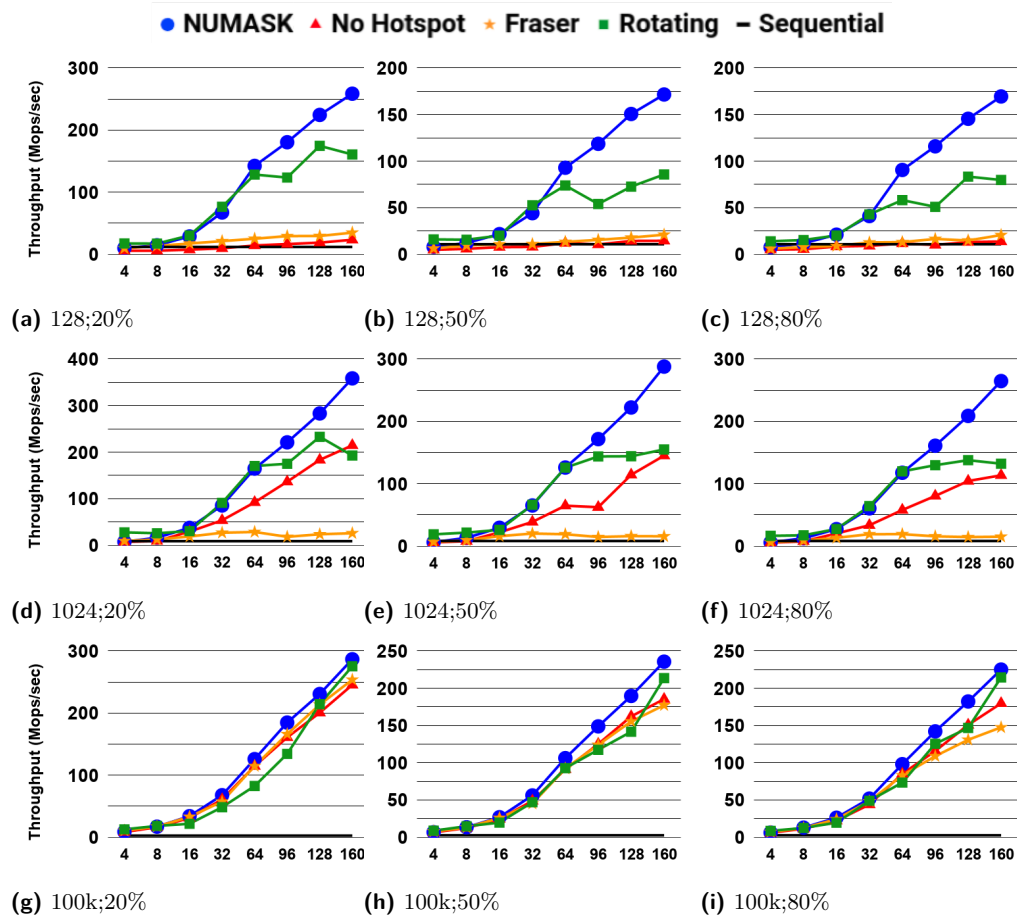


Figure 6 Throughput of NUMASK against other skip list implementations varying data structure size and the percentage of update operations. Throughput is in Millions operations per second.



design. An observation that is not shown in the plots is that the percentage of NUMA-local accesses for the Per-NUMA-Helper threads is consistently slightly lower than 100% (recall that each Per-NUMA-Helper can occasionally access some NUMA-remote location as detailed in Section 5.3).

Figure 6 shows the throughput of NUMASK against the Fraser, Rotating, and No Hotspot skip lists by varying the number of application threads, data structure size, and percentage of update operations. Throughput is measured in millions of operations successfully completed per second. A specially relevant case is the one where the data structure is 1K elements. In the read-intensive scenario, all competitors scale well except for Fraser, with NUMASK demonstrating the highest performance. With 50% and 80% of updates, all competitors stop scaling beyond 64 threads while NUMASK continues scaling, hitting the remarkable performance of 300 million operations per second with 50% updates. In this configuration, at 160 threads NUMASK outperforms rotating skiplist and No Hotspot by 2x.

Reducing the data structure size improves the gap between NUMASK and the other competitors. This is reasonable since our NUMA design avoids synchronization across NUMA zones, which would generate many NUMA-remote accesses.

At 100k element size, the gaps among competitors is reduced. Still, NUMASK is the fastest at 50% updates and 160 threads by gaining 10% over Rotating and 27% over No Hotspot. As mentioned before and confirmed by the analysis of the cache hits/misses, the dominant cost here is repeatedly loading new elements into the cache. This cost obfuscates the effort in improving performance made by NUMASK's design. No Hotspot's performance evaluation also discusses similar findings with large data structure sizes.

## 8 Conclusion

In this paper we presented NUMASK, a high-performance concurrent skip list that uses a combination of distributed design and eventual synchronization to improve performance in NUMA architectures. Our evaluation study shows unquestionably high throughput and remarkable speedups: up to 16x in write-intensive workloads and in the presence of contention.

---

## References

- 1 *numa(3) Linux Programmer's Manual*, second edition, December 2007. URL: <https://linux.die.net/man/3/numa>.
- 2 Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In Larry Rudolph and Anoop Gupta, editors, *ASPLOS-IX Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems, Cambridge, MA, USA, November 12-15, 2000.*, pages 117–128. ACM Press, 2000. Source code available at <https://github.com/emeryberger/Hoard>. doi:10.1145/356989.357000.
- 3 Sergey Blagodurov, Sergey Zhuravlev, Alexandra Fedorova, and Ali Kamali. A case for numa-aware contention management on multicore systems. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 557–558, New York, NY, USA, 2010. ACM. doi:10.1145/1854273.1854350.
- 4 Trevor Brown, Alex Kogan, Yossi Lev, and Victor Luchangco. Investigating the performance of hardware transactions on a multi-socket machine. In Christian Scheideler and Seth Gilbert, editors, *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2016, Asilomar State Beach/Pacific Grove, CA, USA, July 11-13, 2016*, pages 121–132. ACM, 2016. doi:10.1145/2935764.2935796.
- 5 Irina Calciu, Dave Dice, Yossi Lev, Victor Luchangco, Virendra J. Marathe, and Nir Shavit. NUMA-aware Reader-writer Locks. In *PPoPP '13*, 2013.

- 6 Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. Black-box concurrent data structures for NUMA architectures. In Yunji Chen, Olivier Temam, and John Carter, editors, *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*, pages 207–221. ACM, 2017. doi:10.1145/3037697.3037721.
- 7 Tyler Crain, Vincent Gramoli, and Michel Raynal. A speculation-friendly binary search tree. In J. Ramanujam and P. Sadayappan, editors, *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2012, New Orleans, LA, USA, February 25-29, 2012*, pages 161–170. ACM, 2012. doi:10.1145/2145816.2145837.
- 8 Tyler Crain, Vincent Gramoli, and Michel Raynal. No hot spot non-blocking skip list. In *IEEE 33rd International Conference on Distributed Computing Systems, ICDCS 2013, 8-11 July, 2013, Philadelphia, Pennsylvania, USA*, pages 196–205. IEEE Computer Society, 2013. doi:10.1109/ICDCS.2013.42.
- 9 Mohammad Dashti, Alexandra Fedorova, Justin R. Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quéma, and Mark Roth. Traffic management: a holistic approach to memory placement on NUMA systems. In Vivek Sarkar and Rastislav Bodík, editors, *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*, pages 381–394. ACM, 2013. doi:10.1145/2451116.2451157.
- 10 Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized concurrency: The secret to scaling concurrent search data structures. In Özcan Özturk, Kemal Ebcioglu, and Sandhya Dwarkadas, editors, *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*, pages 631–644. ACM, 2015. doi:10.1145/2694344.2694359.
- 11 David Dice, Virendra J. Marathe, and Nir Shavit. Lock Cohorting: A General Technique for Designing NUMA Locks. In *PPoPP '12, 2012*.
- 12 Ian Dick, Alan Fekete, and Vincent Gramoli. A skip list for multicore. *Concurrency and Computation: Practice and Experience*, 29(4), 2017. doi:10.1002/cpe.3876.
- 13 Jason Evans. jemalloc memory allocator. URL: <https://github.com/jemalloc/jemalloc>.
- 14 Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing, PODC '04*, pages 50–59, New York, NY, USA, 2004. ACM. doi:10.1145/1011767.1011776.
- 15 Keir Fraser. *Practical lock-freedom*. PhD thesis, University of Cambridge, September 2003.
- 16 Fabien Gaud, Baptiste Lepers, Justin Funston, Mohammad Dashti, Alexandra Fedorova, Vivien Quéma, Renaud Lachaize, and Mark Roth. Challenges of memory management on modern numa systems. *Commun. ACM*, 58(12):59–66, 2015. doi:10.1145/2814328.
- 17 Vincent Gramoli. More than you ever wanted to know about synchronization: synchrobench, measuring the impact of the synchronization on concurrent algorithms. In Albert Cohen and David Grove, editors, *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2015, San Francisco, CA, USA, February 7-11, 2015*, pages 1–10. ACM, 2015. doi:10.1145/2688500.2688501.
- 18 Ahmed Hassan, Roberto Palmieri, and Binoy Ravindran. Transactional interference-less balanced tree. In *Distributed Computing - 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings*, pages 325–340, 2015.
- 19 Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the Twenty-second Annual ACM*

- Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 355–364, New York, NY, USA, 2010. ACM. doi:10.1145/1810479.1810540.
- 20 M. Herlihy and N. Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.
  - 21 Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. A simple optimistic skiplist algorithm. In *Structural Information and Communication Complexity, 14th International Colloquium, SIROCCO 2007, Castiglione, Italy, June 5-8, 2007, Proceedings*, pages 124–138, 2007.
  - 22 Christoph Lameter. Numa (non-uniform memory access): An overview. *Queue*, 11(7):40:40–40:51, 2013. doi:10.1145/2508834.2513149.
  - 23 Baptiste Lepers, Vivien Quéma, and Alexandra Fedorova. Thread and memory placement on NUMA systems: Asymmetry matters. In Shan Lu and Erik Riedel, editors, *2015 USENIX Annual Technical Conference, USENIX ATC '15, July 8-10, Santa Clara, CA, USA*, pages 277–289. USENIX Association, 2015. URL: <https://www.usenix.org/conference/atc15/technical-session/presentation/lepers>.
  - 24 Zoltan Majo and Thomas R. Gross. Memory management in numa multicore systems: Trapped between cache contention and interconnect overhead. In *Proceedings of the International Symposium on Memory Management, ISMM '11*, pages 11–20, New York, NY, USA, 2011. ACM. doi:10.1145/1993478.1993481.
  - 25 Mohamed Mohamedin, Roberto Palmieri, Sebastiano Peluso, and Binoy Ravindran. On designing numa-aware concurrency control for scalable transactional memory. In Rafael Asenjo and Tim Harris, editors, *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2016, Barcelona, Spain, March 12-16, 2016*, pages 45:1–45:2. ACM, 2016. doi:10.1145/2851141.2851189.
  - 26 Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. Scale-out NUMA. In Rajeev Balasubramonian, Al Davis, and Sarita V. Adve, editors, *Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, Salt Lake City, UT, USA, March 1-5, 2014*, pages 3–18. ACM, 2014. doi:10.1145/2541940.2541965.
  - 27 Iraklis Psaroudakis, Stefan Kaestle, Matthias Grimmer, Daniel Goodman, Jean-Pierre Lozi, and Timothy L. Harris. Analytics with smart arrays: adaptive and efficient language-independent data. In Rui Oliveira, Pascal Felber, and Y. Charlie Hu, editors, *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, pages 17:1–17:15. ACM, 2018. doi:10.1145/3190508.3190514.
  - 28 William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990. doi:10.1145/78973.78977.
  - 29 Nikita Shamgunov. The memsql in-memory database system. In Justin J. Levandoski and Andrew Pavlo, editors, *Proceedings of the 2nd International Workshop on In Memory Data Management and Analytics, IMDM 2014, Hangzhou, China, September 1, 2014.*, 2014.
  - 30 Dmitry Vyukov. Unbounded SPSC Queue, 2018. URL: <http://www.1024cores.net/home/lock-free-algorithms/queues/unbounded-spsc-queue>.



# TuringMobile: A Turing Machine of Oblivious Mobile Robots with Limited Visibility and Its Applications

**Giuseppe A. Di Luna**

Aix-Marseille University and LiS Laboratory, Marseille, France  
giuseppe.diluna@lif.univ-mrs.fr

**Paola Flocchini**

University of Ottawa, Ottawa, Canada  
paola.flocchini@uottawa.ca

**Nicola Santoro**

Carleton University, Ottawa, Canada  
santoro@scs.carleton.ca

**Giovanni Viglietta**<sup>1</sup>

JAIST, Nomi City, Japan  
johnny@jaist.ac.jp

---

## Abstract

In this paper we investigate the computational power of a set of mobile robots with limited visibility. At each iteration, a robot takes a snapshot of its surroundings, uses the snapshot to compute a destination point, and it moves toward its destination. Each robot is punctiform and memoryless, it operates in  $\mathbb{R}^m$ , it has a local reference system independent of the other robots' ones, and is activated asynchronously by an adversarial scheduler. Moreover, the robots are non-rigid, in that they may be stopped by the scheduler at each move before reaching their destination (but are guaranteed to travel at least a fixed unknown distance before being stopped).

We show that despite these strong limitations, it is possible to arrange  $3m + 3k$  of these weak entities in  $\mathbb{R}^m$  to simulate the behavior of a stronger robot that is rigid (i.e., it always reaches its destination) and is endowed with  $k$  registers of persistent memory, each of which can store a real number. We call this arrangement a *TuringMobile*. In its simplest form, a TuringMobile consisting of only three robots can travel in the plane and store and update a single real number. We also prove that this task is impossible with fewer than three robots.

Among the applications of the TuringMobile, we focused on Near-Gathering (all robots have to gather in a small-enough disk) and Pattern Formation (of which Gathering is a special case) with limited visibility. Interestingly, our investigation implies that both problems are solvable in Euclidean spaces of any dimension, even if the visibility graph of the robots is initially disconnected, provided that a small amount of these robots are arranged to form a TuringMobile. In the special case of the plane, a basic TuringMobile of only three robots is sufficient.

**2012 ACM Subject Classification** Computing methodologies → Multi-agent planning

**Keywords and phrases** Mobile Robots, Turing Machine, Real RAM

**Digital Object Identifier** 10.4230/LIPIcs.DISC.2018.19

**Related Version** Full Version on ArXiv: [17], <https://arxiv.org/pdf/1709.08800>.

---

<sup>1</sup> Contact Author. Address: 1-50-D-21 Asahidai, Nomi City, Ishikawa Prefecture 923-1211, Japan. Phone: +81 80-8691-6839.



## 1 Introduction

### 1.1 Framework and Background

The investigations of systems of autonomous mobile robots have long moved outside the boundaries of the engineering, control, and AI communities. Indeed, the computational and complexity issues arising in such systems are important research topics within theoretical computer science, especially in distributed computing. In these theoretical investigations, the robots are usually viewed as punctiform computational entities that live in a metric space, typically  $\mathbb{R}^2$  or  $\mathbb{R}^3$ , in which they can move. Each robot operates in “Look-Compute-Move” (LCM) cycles: it observes its surroundings, it computes a destination within the space based on what it sees, and it moves toward the destination. The only means of interaction between robots are observations and movements: that is, communication is *stigmergic*. The robots, identical and outwardly indistinguishable, are *oblivious*: when starting a new cycle, a robot has no memory of its activities (observations, computations, and moves) from previous cycles (“every time is the first time”).

There have been intensive research efforts on the computational issues arising with such robots, and an extensive literature has been produced in particular in regard to the important class of *Pattern Formation* problems [8, 19, 21, 22, 27, 28] as well as for *Gathering* [1, 2, 4, 7, 9, 10, 12, 11, 14, 20, 24] and *Scattering* [5, 23]; see also [6, 13, 29]. The goal of the research has been to understand the minimal assumptions needed for a team (or swarm) of such robots to solve a given problem, and to identify the impact that specific factors have on feasibility and hence computability.

The most important factor is the power of the adversarial scheduler that decides when each activity of each robot starts and when it ends. The main adversaries (or “environments”) considered in the literature are: *synchronous*, in which the computation cycles of all active robots are synchronized, and at each cycle either all (in the fully synchronous case) or a subset (in the semi-synchronous case) of the robots are activated, and *asynchronous*, where computation cycles are not synchronized, each activity can take a different and unpredictable amount of time, and each robot can be independently activated at each time instant.

An important factor is whether a robot moving toward a computed destination is guaranteed to reach it (i.e., it is a *rigid* robot), or it can be stopped on the way (i.e., it is a *non-rigid* robot) at a point decided by an adversary. In all the above cases, the power of the adversaries is limited by some basic fairness assumption. All the existing investigations have concentrated on the study of (a-)synchrony, several on the impact of rigidity, some on other relevant factors such as agreement on local coordinate systems or on their orientation, etc.; for a review, see [18].

From a computational point of view, there is another crucial factor: the visibility range of the robots, that is, how much of the surrounding space they are able to observe in a Look operation. In this regard, two basic settings are considered: *unlimited visibility*, where the robots can see the entire space (and thus all other robots), and *limited visibility*, when the robots have a fixed visibility radius. While the investigations on (a-)synchrony and rigidity have concentrated on all aspects of those assumptions, this is not the case with respect to visibility. In fact, almost all research has assumed unlimited visibility; few exceptions are the algorithms for Convergence [4], Gathering [15, 16, 20], and Near-Gathering [24] when the visibility range of the robot is limited. The unlimited visibility assumption clearly greatly simplifies the computational universe under investigation; at the same time, it neglects the more general and realistic one, which is still largely unknown.

Let us also stress that, in the existing literature, all results on oblivious robots are for  $\mathbb{R}^1$  and  $\mathbb{R}^2$ ; the only exception is the recent result on plane formation in  $\mathbb{R}^3$  by semi-synchronous rigid robots with unlimited visibility [29]. No results exist for robots in higher dimensions.

## 1.2 Contributions

In this paper we contribute several constructive insights on the computational universe of oblivious robots with limited visibility, especially asynchronous non-rigid ones, in any dimension.

### TuringMobile

The first and main contribution is the design of a “moving Turing Machine” made solely of asynchronous oblivious non-rigid robots in  $\mathbb{R}^m$  with limited visibility, for any  $m \geq 2$ . More precisely, we show how to arrange  $3m + 3k$  identical non-rigid oblivious robots in  $\mathbb{R}^m$  with a visibility radius of  $V + \varepsilon$  (for any  $\varepsilon > 0$ ) and how to program them so that they can collectively behave as a single rigid robot in  $\mathbb{R}^m$  with  $k$  persistent registers and visibility radius  $V$  would. This team of identical robots is informally called a *TuringMobile*. We obtain this result by using as fundamental construction a basic component, which is able to move in  $\mathbb{R}^2$  while storing and updating a single real number. Interestingly, we show that 3 agents are necessary and sufficient to build such a machine. The TuringMobile will then be built by arranging multiple copies of this basic component. Notably, the robots that constitute a TuringMobile need only be able to compute arithmetic operations and square roots.

A TuringMobile is a powerful construct that, once deployed in a swarm of robots, can act as a rigid leader with persistent memory, allowing the swarm to overcome many handicaps imposed by obliviousness, limited visibility, and asynchrony. As examples we present a variety of applications in  $\mathbb{R}^m$ , with  $m \geq 2$ .

There is a limitation to the use of a TuringMobile when deployed in a swarm of robots. Namely, the TuringMobile must be always recognizable (e.g., by its unique shape) so that other robots cannot interfere by moving too close to the machine, disrupting its structure. This limitation can be overcome when the robots of the TuringMobile are visibly distinguishable from the others. However, this requirement is not necessary for all applications, but is only required when we want to perfectly simulate a rigid robot with memory.

We remark that we do not discuss how robots can self-assemble into a TuringMobile. We only focus on how the machine can be designed when we can freely arrange some robots. In the case of robots with unlimited visibility, a TuringMobile can be self-assembled, provided that the initial configuration of the robots is asymmetric. In the case of limited visibility, self-assembling a TuringMobile is a more complex and still open problem.

### Applications

We propose several applications of our TuringMobile. First of all, the TuringMobile can explore and search the space. We then show how it can be employed to solve the long-standing open problem of (Near-)Gathering with limited visibility in spite of an asynchronous non-rigid scheduler and disagreement on the axes, a problem still open without a TuringMobile. Interestingly, the presence of the TuringMobile allows Gathering to be done even if the initial visibility graph is disconnected. Finally we show how the arbitrary Pattern Formation problem can be solved under the same conditions (asynchrony, limited visibility, possibly disconnected visibility graph, etc.).



The paper is organized as follows: In Section 2 we give formal definitions, introducing mobile robots with or without memory as *oracle semi-oblivious real RAMs*. In Section 3 we illustrate our implementation of the TuringMobile. In Section 4 we show how to apply the TuringMobile to solve fundamental problems. Due to space constraints, the proof of correctness of our TuringMobile implementation, several technical parts of the paper, and additional figures can be found in the full paper [17].

## 2 Definitions and Preliminaries

### 2.1 Oracle Semi-Oblivious Real RAMs

**Real random-access machines.** A *real RAM*, as defined by Shamos [25, 26], is a random-access machine [3] that can operate on real numbers. That is, instead of just manipulating and storing integers, it can handle arbitrary real numbers and do infinite-precision operations on them. It has a finite set of internal *registers* and an infinite ordered sequence of *memory cells*; each register and each memory cell can hold a single real number, which the machine can modify by executing its program.<sup>2</sup>

A real RAM’s instruction set contains at least the four arithmetic operations, but it may also contain  $k$ -th roots, trigonometric functions, exponentials, logarithms, and other analytic functions, depending on the application. The machine can also compare two real numbers and branch depending on which one is larger.

The initial contents of the memory cells are the *input* of the machine (we stipulate that only finitely many of them contain non-zero values), and their contents when the machine halts are its *output*. So, each program of a real RAM can be viewed as a partial function mapping tuples of reals into tuples of reals.

**Oracles and semi-obliviousness.** We introduce the *oracle semi-oblivious real RAM*, which is a real RAM with an additional “ASK” instruction. Whenever this instruction is executed, the contents of all the memory cells are replaced with new values, which are a function of the numbers stored in the registers.

In other words, the machine can query an external oracle by putting a question in its  $k$  registers in the form of  $k$  real numbers. The oracle then reads the question and writes the answer in the machine’s memory cells, erasing all pre-existing data. The term “semi-oblivious” comes from the fact that, every time the machine invokes the oracle, it “forgets” everything it knows, except for the contents of the registers, which are preserved.<sup>3</sup>

► Remark. In spite of their semi-obliviousness, these real RAMs with oracles are at least as powerful as Turing Machines with oracles.

### 2.2 Mobile Robots as Real RAMs

**Mobile robots.** Our oracle semi-oblivious real RAM model can be reinterpreted in the realm of *mobile robots*. A mobile robot is a computational entity, modeled as a geometric point, that lives in a metric space, typically  $\mathbb{R}^2$  or  $\mathbb{R}^3$ . It can observe its surroundings and

<sup>2</sup> Nonetheless, the constant operands in a real RAM’s program cannot be arbitrary real numbers, but have to be integers.

<sup>3</sup> Observe that, in general, the machine cannot salvage its memory by encoding its contents in the registers: since its instruction set has only analytic functions, it cannot injectively map a tuple of arbitrary real numbers into a single real number.



move within the space based on what it sees. The same space may be populated by several mobile robots, each with its local coordinate system, and static objects.

To compute its next destination point, a mobile robot executes a real RAM program with input a representation of its local view of the space. After moving, its entire memory is erased, but the content of its  $k$  registers is preserved. Then it makes a new observation; from the observation data and the contents of the registers, it computes another destination point, and so on. If  $k = 0$ , the mobile robot is said to be *oblivious*. Note that robots have no notion of time or absolute positions.

The actual movement of a mobile robot is controlled by an external *scheduler*. The scheduler decides how fast the robot moves toward its destination point, and it may even interrupt its movement before the destination point is reached. If the movement is interrupted midway, the robot makes the next observation from there and computes a new destination point as usual. The robot is not notified that an interruption has occurred, but it may be able to infer it from its next observation and the contents of its registers. For fairness, the scheduler is only allowed to interrupt a robot after it has covered a distance of at least  $\delta$  in the current movement, where  $\delta$  is a positive constant unknown to the robots. This guarantees, for example, that if a robot keeps computing the same destination point, it will reach it in a finite number of iterations. If  $\delta = \infty$ , the robot always reaches its destination, and is said to be *rigid*.

**Mobile robots, revisited.** A mobile robot in  $\mathbb{R}^m$  with  $k$  registers can be modeled as an oracle semi-oblivious real RAM with  $2m + k + 1$  registers, as follows.

- $m$  *position registers* hold the absolute coordinates of the robot in  $\mathbb{R}^m$ .
- $m$  *destination registers* hold the destination point of the robot, expressed in its local coordinate system.
- 1 *timestamp register* contains the time of the robot's last observation.
- $k$  *true registers* correspond to the registers of the robot.

As the RAM's execution starts, it ignores its input, erases all its registers, and executes an "ASK" instruction. The oracle then fills the RAM's memory with the robot's initial position  $p$ , the time  $t$  of its first observation, and a representation of the geometric entities and objects surrounding the robot, as seen from  $p$  at time  $t$ .

The RAM first copies  $p$  and  $t$  in its position registers and timestamp register, respectively. Then it executes the program of the mobile robot, using its true registers as the robot's registers and adding  $m + 1$  to all memory addresses. This effectively makes the RAM ignore the values of  $p$  and  $t$ , which indeed are not supposed to be known to the mobile robot.

When the robot's program terminates, the RAM's memory contains the output, which is the next destination point  $p'$ , expressed in the robot's coordinate system. The RAM copies  $p'$  into its destination registers, and the execution jumps back to the initial "ASK" instruction.

Now the oracle reads  $p$ ,  $p'$ , and  $t$  from the RAM's registers (it ignores the true registers), converts  $p'$  in absolute coordinates (knowing  $p$  and the orientation of the local coordinate system of the robot) and replies with a new position  $p''$ , a timestamp  $t' > t$ , and observation data representing a snapshot taken from  $p''$  at time  $t'$ . To comply with the mobile robot model,  $p''$  must be on the segment  $pp'$ , such that either  $p'' = p'$  or  $\overline{pp''} \geq \delta$ . The execution then proceeds in the same fashion, indefinitely.

Note that in this setting the oracle represents the scheduler. The presence of a timestamp in the query allows the oracle to model dynamic environments in which several independent robots may be moving concurrently (without a timestamp, two observations from the same point of view would always be identical). Also note that in this formulation there are no

actual robots moving through an environment in time, but only RAMs which query an oracle, which in turn provides a “virtual” environment and timeline by writing information in their memory.

**Snapshots and limited visibility.** In the mobile robot model we consider in this paper, an observation is simply an instantaneous *snapshot* of the environment taken from the robot’s position. In turn, each entity and object that the robot can see is modeled as a dimensionless point in  $\mathbb{R}^m$ . A mobile robot has a positive *visibility radius*  $V$ : it can see a point in  $\mathbb{R}^m$  if and only if it is located at distance at most  $V$  from its current position. If  $V = \infty$ , the robot is said to have *unlimited visibility*.

As we hinted at earlier in this section, a mobile robot has its own local reference system in which all the coordinates of the objects in its snapshots are expressed. The origin of a robot’s local coordinate system always coincides with the robot’s position (hence it follows the robot as it moves), and its orientation and handedness are decided by the scheduler (and remain fixed). Different mobile robots may have coordinate systems with a different orientation or handedness. (However, when two robots have the same visibility radius, they also implicitly have the same unit of distance.)

So, a snapshot is just a (finite) list of points, each of which is an  $m$ -tuple of real numbers.

**Simulating memory and rigidity.** The main contribution of this paper, loosely speaking, is a technique to turn non-rigid oblivious robots into rigid robots with persistent memory, under certain conditions. More precisely, if  $3m + 3k$  identical non-rigid oblivious robots in  $\mathbb{R}^m$  with a visibility radius of  $V + \varepsilon$  (for any  $\varepsilon > 0$ ) are arranged in a specific pattern and execute a specific algorithm, they can collectively act in the same way as a single rigid robot in  $\mathbb{R}^m$  with  $k > 0$  persistent registers and visibility radius  $V$  would. This team of identical robots is informally called a *TuringMobile*.

We stress that the robots of a TuringMobile are *asynchronous*, that is, the scheduler makes them move at independent arbitrary speeds, and each robot takes the next snapshot an arbitrary amount of time after terminating each move. The robots are also *anonymous*, in that they are indistinguishable from each other, and they all execute the same program.

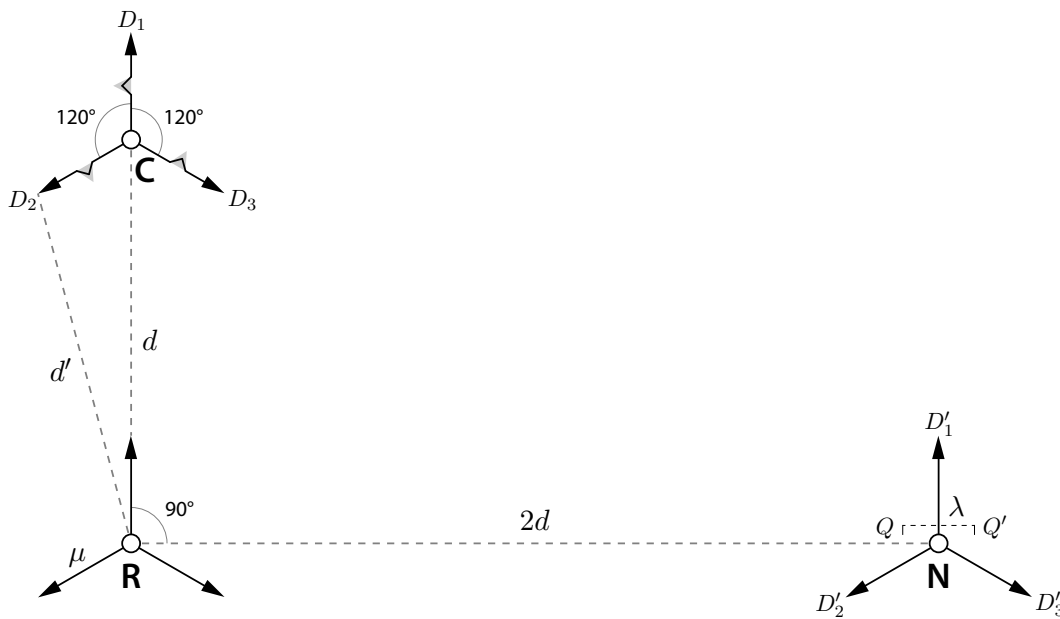
Although our technique is fairly general and has a plethora of concrete applications (some are discussed in Section 4), a “perfect simulation” is achieved only under additional conditions on the scheduler or on the environment (see Section 3.2).

## 3 Implementing the TuringMobile

### 3.1 Basic Implementation

We will first describe how to construct a basic version of the TuringMobile with just three oblivious non-rigid robots in  $\mathbb{R}^2$ . This TuringMobile can remember a single real number and rigidly move in the plane by fixed-length steps: its layout is sketched in Figure 1. In Section 3.2, we will show how to combine several copies of this basic machine to obtain a full-fledged TuringMobile.

**Position at rest.** The elements of the basic TuringMobile are three: a *Commander* robot, a *Number* robot, and a *Reference* robot, located in  $C$ ,  $N$ , and  $R$ , respectively. These robots have the same visibility radius of  $V + \varepsilon$ , where  $\varepsilon \ll V$ , and there is always a disk of radius  $\varepsilon$  containing all three of them. When the machine is “at rest”,  $\angle NRC$  is a right angle, the distance between  $C$  and  $R$  is some fixed value  $d \ll \varepsilon$ , and the distance between  $R$  and  $N$  is approximately  $2d$ . More precisely,  $N$  lies on a segment  $QQ'$  of length  $\lambda$ , where  $\lambda \ll d$  is some fixed value, such that  $Q$  has distance  $2d - \lambda/2$  from  $R$  and  $Q'$  has distance  $2d + \lambda/2$  from  $R$ .



■ **Figure 1** Basic TuringMobile at rest, not drawn to scale ( $\mu$  and  $\lambda$  should be smaller).

**Representing numbers.** The distance between the Reference robot and the Number robot when the TuringMobile is at rest is a representation of the real number  $r$  that the machine is currently storing. One possible technique is to encode the number  $r$  as  $\overline{RN} = 2d + \arctan(r) \cdot \lambda/\pi$  and to decode it as  $r = \tan((\overline{RN} - 2d) \cdot \pi/\lambda)$ . However, there are also more complicated methods that use only arithmetic functions (see the full paper [17]).

**Movement directions.** The Commander's role is to decide in which direction the machine should move next, and to initiate the movement. When the machine is at rest, the Commander may choose among three possible *final destinations*, labeled  $D_1$ ,  $D_2$ , and  $D_3$  in Figure 1. The segments  $CD_1$ ,  $CD_2$ , and  $CD_3$  all have the same length  $\mu$ , with  $\lambda \ll \mu \ll d$ , and form angles of  $120^\circ$  with one another, in such a way that  $D_1$  is collinear with  $R$  and  $C$ .

Around the center of each segment  $CD_i$  there is a *midway triangle*  $\tau_i$ , drawn in gray in Figure 1. This is an isosceles triangle of height  $\lambda$  whose base lies on  $CD_i$  and has length  $\lambda$  as well. When the Commander decides that its final destination is  $D_i$ , it moves along the segment  $CD_i$ , but it takes a detour in the midway triangle  $\tau_i$ , as we will explain shortly.

**Structure of the algorithm.** Algorithm 1 is the program that each element of the basic TuringMobile executes every time it computes its next destination point.

Since the robots are anonymous, they first have to determine their roles, i.e., who is the Commander, etc. (line 1 of the algorithm). We make the assumption that there exists a disk of radius  $\varepsilon$  containing only the TuringMobile (close to its center) and no other robot. Using the fact that the two closest robots must be the Commander and the Reference robot and that the two farthest robots must be the Commander and the Number robot, it is then easy to determine who is who (these properties will be preserved throughout the execution, as proved in the full paper [17]).

Once it has determined its role, each robot executes a different branch of the algorithm (cf. lines 2, 13, and 23). The general idea is that, when the Commander realizes that the machine is in its rest position, it decides where to move next, i.e., it chooses a final destination

**Algorithm 1** Basic TuringMobile in  $\mathbb{R}^2$ .

---

```

1: Identify Commander, Number, Reference (located in  $C$ ,  $N$ ,  $R$ , respectively)
2: if I am Commander then
3:   Compute Virtual Commander  $C'$  (based on  $R$  and  $N$ ) and points  $A_i, S_i, S'_i, B_i, D_i$ 
4:   if I am in  $C'$  then Choose final destination  $D_i$  and move to  $A_i$ 
5:   else if  $\exists i \in \{1, 2, 3\}$  s.t. I am on segment  $C'A_i$  but not in  $A_i$  then Move to  $A_i$ 
6:   else if  $\exists i \in \{1, 2, 3\}$  s.t. I am in  $A_i$  then
7:     Move to point  $P$  on segment  $S_iS'_i$  such that  $\overline{PS_i} = f(\overline{NQ})$ 
8:   else if  $\exists i \in \{1, 2, 3\}$  s.t. I am in triangle  $A_iS_iS'_i$  but not on segment  $S_iS'_i$  then
9:     Move to the intersection of segment  $S_iS'_i$  with the extension of line  $A_iC$ 
10:  else if  $\exists i \in \{1, 2, 3\}$  s.t. I am on  $S_iS'_i$  and  $\overline{NQ} = \overline{CS_i}$  then Move to  $B_i$ 
11:  else if  $\exists i \in \{1, 2, 3\}$  s.t. I am in triangle  $B_iS_iS'_i$  but not in  $B_i$  then Move to  $B_i$ 
12:  else if  $\exists i \in \{1, 2, 3\}$  s.t. I am on segment  $B_iD_i$  but not in  $D_i$  then Move to  $D_i$ 
13: else if I am Number then
14:   if  $\overline{CR} = d + \mu$  or  $\overline{CR} = d'$  then
15:     Compute Virtual Commander  $C'$  (based on  $C$  and  $R$ ) and points  $D'_i$ 
16:     if  $\overline{CR} = d + \mu$  and I am not in  $D'_1$  then Move to  $D'_1$ 
17:     else if  $\overline{CR} = d'$  and  $\angle NRC > 90^\circ$  and I am not in  $D'_2$  then Move to  $D'_2$ 
18:     else if  $\overline{CR} = d'$  and  $\angle NRC < 90^\circ$  and I am not in  $D'_3$  then Move to  $D'_3$ 
19:   else
20:     Compute Virtual Commander  $C'$  (based on  $R$  and  $N$ ) and points  $S_i, S'_i$ 
21:     if  $\exists i \in \{1, 2, 3\}$  s.t.  $C$  is on segment  $S_iS'_i$  then
22:       Move to point  $P$  on segment  $QQ'$  such that  $\overline{PQ} = \overline{CS_i}$ 
23:   else if I am Reference then
24:     if Commander and Number are not tasked with moving (based on the above rules) then
25:        $\gamma$  = circle centered in  $C$  with radius  $d$ 
26:        $\gamma'$  = circle with diameter  $CN$ 
27:       Move to the intersection of  $\gamma$  and  $\gamma'$  closest to  $R$ 

```

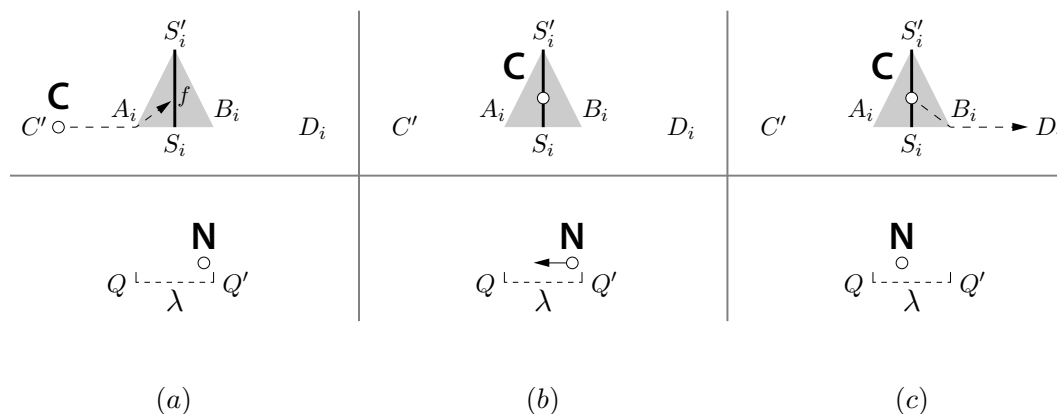
---

$D_i$ . This choice is based on the number  $r$  stored in the machine's "memory" (i.e., the number encoded by  $\overline{RN}$ ), the relative positions of the visible robots external to the machine, and also on the application, i.e., the specific program that the TuringMobile is executing.

When the Commander has decided its final destination  $D_i$ , the entire machine moves by the vector  $\overrightarrow{CD_i}$ , and the Number robot also updates its distance from the Reference robot to represent a different real number  $r'$ . Again, this number is computed based on the number  $r$  the machine was previously representing, the relative positions of the visible robots external to the machine, and the specific program: in general, the new distance between  $N$  and  $Q$  is a function  $f$  of the old distance. When all this is done, the machine is in its rest position again, so the Commander chooses a new destination, and so on, indefinitely.

**Coordinating movements.** Note that it is not possible for all three robots to translate by  $\overrightarrow{CD_i}$  at the same time, because they are non-rigid and asynchronous. If the scheduler stops them at arbitrary points during their movement, after the structure of the machine has been destroyed, they will be incapable of recovering all the information they need to resume their movement (recall that they are oblivious and they have to compute a destination point from scratch every time).

To prevent this, the robots employ various coordination techniques. First the Commander moves to the middle triangle  $\tau_i$ , and precisely to its base vertex  $A_i$ , as shown in Figure 2(a) (cf. line 5 of Algorithm 1). Then it positions itself on the altitude  $S_iS'_i$ , in such a way as to indicate the new number  $r'$  that the machine is supposed to represent. That is, the



■ **Figure 2** Coordinated movement of the Commander and the Number robot, to cope with their asynchronous and non-rigid nature. (a) The Commander stops on  $S_i S'_i$ , recording the number that the machine is going to represent next (which is a function  $f$  of the number currently represented by the Number robot). (b) The Number robot moves within  $QQ'$  to match the Commander's position in  $S_i S'_i$ . (c) Finally, the Commander reaches  $D_i$ .

Commander picks the point on  $S_i S'_i$  at distance  $f(\overline{NQ})$  from  $S_i$  (lines 6 and 7). Even if it is stopped by the scheduler before reaching such a point, it can recover its destination by drawing a ray from  $A_i$  to its current position and intersecting it with  $S_i S'_i$  (lines 8 and 9).

When the Commander has reached  $S_i S'_i$ , it waits to let the Number robot adjust its position on the segment  $QQ'$  to match that of the Commander on  $S_i S'_i$ , as in Figure 2(b) (lines 21 and 22). This effectively makes the Number robot represent the new number  $r'$ . Note that the Number robot can do this even if it is stopped by the scheduler several times during its march, because the Commander keeps reminding it of the correct value of  $r'$ : since  $r'$  depends on the old number  $r$ , the Number robot would be unable to re-compute  $r'$  after it has forgotten  $r$ . Once the Number robot has reached the correct position on  $QQ'$ , the Commander starts moving again (line 10) and finally reaches  $D_i$  while the other robots wait, as in Figure 2(c) (lines 11 and 12).

When the Commander has reached  $D_i$ , the Number robot realizes it and makes the corresponding move (lines 14–18) while the other two robots wait. The destination point of the Number robot is  $D'_i$ , as shown in Figure 1. Finally, when the Number robot is in  $D'_i$ , the Reference robot realizes it and makes the final move to bring the TuringMobile back into a rest position (lines 23–27). Note that the number  $r'$  stored in the machine is not erased after these final movements, because both the Number and Reference robot move by the same vector.

**Computing the Virtual Commander.** After the Commander has left its rest position and is on its way to  $D_i$ , the TuringMobile loses its initial shape, and identifying the  $D_i$ 's and the midway triangles becomes non-trivial. So, the robots try to guess where the Commander's original rest position may have been by computing a point called the *Virtual Commander*  $C'$ .

Assuming that the Reference and Number robots have not moved from their rest positions, the Virtual Commander is easily computed: draw a line  $\ell$  through  $R$  perpendicular to  $RN$ ; then,  $C'$  is the point on  $\ell$  at distance  $d$  from  $R$  that is closest to  $C$ . Once we have  $C'$ , we can construct the points  $D_i$  with respect to  $C'$  (in the same way as we did in Figure 1 with respect to  $C$ ). This technique is used by Algorithm 1 at lines 3 and 20.

In the special case where the Commander has reached its final destination  $D_i$  and the Reference robot has not moved from its rest position (but perhaps the Number robot has moved), the Virtual Commander can also be computed. This situation is recognized because the distance between the Commander and the Reference robot is either maximum (i.e.,  $d + \mu$ ) or minimum (i.e.,  $d' = \sqrt{d^2 + \mu^2 - d\mu}$ ), as Figure 1 shows. If the distance is maximum, then  $C$  must coincide with  $D_1$ ; otherwise,  $C$  coincides with  $D_2$  (if the angle  $\angle NRC$  is obtuse) or  $D_3$  (if the angle  $\angle NRC$  is acute). Since we know the position of  $R$  and one of the  $D_i$ 's, it is then easy to determine the other  $D_i$ 's. This technique is used at line 15.

**The Reference robot's behavior.** To know when it has to start moving, the Reference robot executes Algorithm 1 from the perspective of the Commander and the Number robot: if neither of them is supposed to move, then the Reference robot starts moving (line 24).

We have seen that the Number robot can determine its destination  $D_i$  solely by looking at the positions of  $C$  and  $R$ , which remain fixed as it moves. For the Reference robot the destination point is not as easy to determine, because the distance between  $C$  and  $N$  varies depending on what number is stored in the TuringMobile.

However, the Reference robot knows that its move must put the TuringMobile in a rest position. The condition for this to happen is that its destination point be at distance  $d$  from  $C$  (line 25) and form a right angle with  $C$  and  $N$  (line 26). There are exactly two such points in the plane, but one of them has distance much greater than  $\mu$  from  $R$ , and hence the Reference robot will pick the other (line 27).

As the Reference robot moves toward such a point, all the above conditions must be preserved, due to the asynchronous and non-rigid nature of the robots. This is not a trivial requirement, and a proof that it is indeed fulfilled is in the full paper [17].

### 3.2 Complete Implementation

We have shown how to implement a basic component of the TuringMobile in  $\mathbb{R}^2$  consisting of three robots: a Commander, a Number, and a Reference. The basic component is able to rigidly move by a fixed distance  $\mu$  in three fixed directions,  $120^\circ$  apart from one another. It can also store and update a single real number.

**Planar layout.** We can obtain a full-fledged TuringMobile in  $\mathbb{R}^2$  by putting several tiny copies of the basic component side by side. For the machine to work, we stipulate that there exists a disk of radius  $\sigma$  that contains all the robots constituting the TuringMobile and no extraneous robot, where  $\sigma \ll \varepsilon$ . The distance between two consecutive basic components of the TuringMobile is roughly  $s$ , where  $d \ll s \ll \sigma$ . This makes it easy for the robots to tell the basic components apart and determine the role of each robot within its basic component.

Since a basic component of the TuringMobile is a scalene triangle, which is chiral, all its members implicitly agree on a clockwise direction even if they have different handedness. Similarly, all robots in the TuringMobile agree on a “leftmost” basic component, whose Commander is said to be the *Leader* of the whole machine.

**Coordinated movements.** All the basic components of the TuringMobile are always supposed to agree on their next move and proceed in a roughly synchronous way. To achieve this, when all the basic components are in a rest position, the Leader decides the next direction among the three possible, and executes line 4 of Algorithm 1. Then all the other Commanders see where the Leader is going, and copy its movement.

When all the Commanders are in their respective  $A_i$ 's, they execute line 7 of the algorithm, and so on. At any time, each robot executes a line of the algorithm only if all its homologous robots in the other basic components of the TuringMobile are ready to execute that line or have already executed it; otherwise, it waits. When the last Reference robot has completed its movement, the machine is in a rest position again, and the coordinated execution repeats with the Leader choosing another direction, etc.

**Simulating a non-oblivious rigid robot.** Let a program for a rigid robot  $\mathcal{R}$  in  $\mathbb{R}^2$  with  $k$  persistent registers and visibility radius  $V$  be given. We want the TuringMobile described above to act as  $\mathcal{R}$ , even though its constituting robots are non-rigid and oblivious.

Our TuringMobile consists of  $2 + k$  basic components, each dedicated to memorizing and updating one real number. These  $2 + k$  numbers are the  $x$  coordinate and the  $y$  coordinate of the destination point of  $\mathcal{R}$  and the contents of the  $k$  registers of  $\mathcal{R}$ . We will call the first two numbers the  $x$  variable and the  $y$  variable, respectively.

When the TuringMobile is in a rest position, its  $x$  and  $y$  variables represent the coordinates of the destination point of  $\mathcal{R}$  relative to the Leader of the machine. Whenever the TuringMobile moves by  $\mu$  in some direction, these values are updated by subtracting the components of an appropriate vector of length  $\mu$  from them. Of course, this update is computed by the Commanders of the first two basic components of the machine, which communicate it to their respective Number robots, as explained in Section 3.1.

Let  $P$  be the destination point of  $\mathcal{R}$ . Since the TuringMobile can only move by vectors of length  $\mu$  in three possible directions, it may be unable to reach  $P$  exactly. So, the Leader always plans the next move trying to reduce its distance from  $P$  until this distance is at most  $2\sigma$  (this is possible because  $\mu \ll d \ll \sigma$ ).

When the Leader is close enough to  $P$ , it “pretends” to be in  $P$ , and the TuringMobile executes the program of  $\mathcal{R}$  to compute the next destination point. Recall that the visibility radius of  $\mathcal{R}$  is  $V$ , and that of the robots of the TuringMobile is  $V + \varepsilon$ . Since  $\sigma \ll \varepsilon$ , each member of the TuringMobile can therefore see everything that would be visible to  $\mathcal{R}$  if it were in  $P$ , and compute the output of the program of  $\mathcal{R}$  independently of the other members. The only thing it should do when it executes the program of  $\mathcal{R}$  is subtract the values of the  $x$  and  $y$  variables to everything it sees in its snapshot, discard whatever has distance greater than  $V$  from the center of the snapshot, and of course discard the robots of the TuringMobile and replace them with a single robot in the center of the snapshot (representing the robot itself). Then, the robots that are responsible for updating the  $x$  and  $y$  variables add the relative coordinates of the new destination point of  $\mathcal{R}$  to these variables. Similarly, the robots responsible for updating the  $k$  registers of  $\mathcal{R}$  do so.

**Restrictions.** The above TuringMobile correctly simulates  $\mathcal{R}$  under certain conditions. The first one is that, if all robots are indistinguishable, then no robot extraneous to the TuringMobile may get too close to it (say, within a distance of  $\sigma$  of any of its members). This kind of restriction cannot be dispensed with: whatever strategy a team of oblivious robots employs to simulate a single non-oblivious robot’s behavior is bound to fail if extraneous robots join the team creating ambiguities between its members. Nevertheless, the restriction can be removed if the members of a TuringMobile are distinguishable from all other robots.

Another difficulty comes from the fact that, if the TuringMobile is made of more than one basic component and its Commanders are all in their respective  $A_i$ 's and ready to update the values represented by the machine, they may get their snapshots at different times, due to

asynchrony. If the environment moves in the meantime, the snapshots they get are different, and this may cause the machine to compute an incorrect destination point or put inconsistent values in its simulated registers.

There are several possible solutions to this problem: we will only mention two trivial ones. We could assume the Commanders to be *synchronous*, that is, make the scheduler activate them in such a way that all of them take their snapshots at the same time. This way, all Commanders get compatible snapshots and compute consistent outputs. Another possible solution is to make the TuringMobile operate in an environment where everything else is static, i.e., no moving entities are present other than the TuringMobile's members.

We stress that these restrictions make sense if a perfect simulation of  $\mathcal{R}$  is sought. As we will see in Section 4, there are several other applications of the TuringMobile technique where no such restrictions are required.

**Higher dimensions.** Let us now generalize the above construction of a planar TuringMobile to  $\mathbb{R}^m$ , for any  $m \geq 2$ . We start with the same TuringMobile  $\mathcal{M}$  with  $2 + k$  basic components laid out on a plane  $\gamma \subset \mathbb{R}^m$ . Since  $\mathcal{M}$  has only two basic components for the  $x$  and  $y$  variables, we will add  $m - 2$  basic components to it, positioned as follows.

Let vectors  $v_1$  and  $v_2$  be two orthonormal generators of  $\gamma$ , and let us complete  $\{v_1, v_2\}$  to an orthonormal basis  $\{v_1, v_2, \dots, v_m\}$  of  $\mathbb{R}^m$ . Now, for all  $i \in \{3, 4, \dots, m\}$ , we make a copy of the basic component of  $\mathcal{M}$  containing the Leader, we translate it by  $s \cdot v_i$ , and we add it to the TuringMobile ( $s$  is the same value used in the construction of the planar TuringMobile at the beginning of Section 3.2). Note that the Leader of this new TuringMobile  $\mathcal{M}'$  is still easy to identify, as well as the plane  $\gamma$  when  $\mathcal{M}'$  is at rest.

Clearly,  $m$  basic components allow the machine to record a destination point in  $\mathbb{R}^m$ , as opposed to  $\mathbb{R}^2$ . Additionally, the positions of the basic components with respect to  $\gamma$  give the machine an  $m$ -dimensional sense of direction (see the full paper [17] for further details).

► **Theorem 1.** *Under the aforementioned restrictions, a rigid robot in  $\mathbb{R}^m$  with  $k$  persistent registers and visibility radius  $V$  can be simulated by a team of  $3m + 3k$  non-rigid oblivious robots in  $\mathbb{R}^m$  with visibility radius  $V + \varepsilon$ .*

## 4 Applications

In this section we discuss some applications of the TuringMobile. We also prove that the basic TuringMobile constructed in Section 3.1 is minimal, in the sense that no smaller team of oblivious robots can accomplish the same tasks.

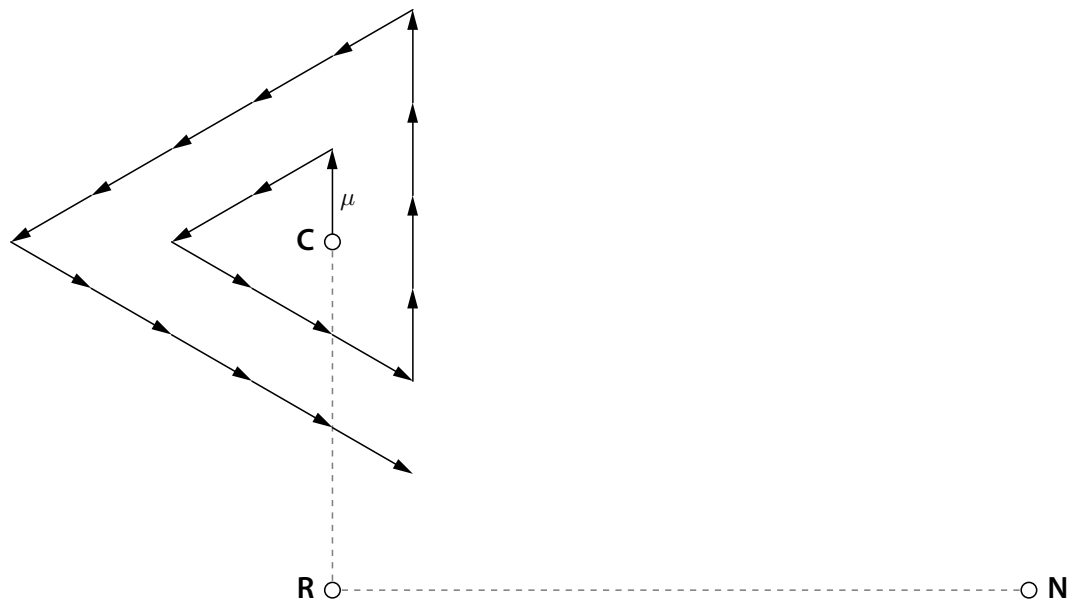
### 4.1 Exploring the Plane

The first elementary task a basic TuringMobile in  $\mathbb{R}^2$  can fulfill is that of *exploring* the plane. The task consists in making all the robots in the TuringMobile see every point in the plane in the course of an infinite execution. We first assume that the three members of the TuringMobile are the only robots in the plane. Later in this section, we will extend our technique to other types of scenarios and more complex tasks.

► **Theorem 2.** *A basic TuringMobile consisting of three robots in  $\mathbb{R}^2$  can explore the plane.*

**Proof.** Recall that a basic TuringMobile can store a single real number  $r$  and update it at every move as a result of executing a real RAM program with input  $r$ . In particular, the TuringMobile can count how many times it has moved by simply starting its execution with  $r = 0$  and computing  $r := r + 1$  at each move.





■ **Figure 3** Exploration of the plane by a basic TuringMobile.

Moreover, the Commander chooses the direction of the next move (in the form of a point  $D_i$ , see Figure 1) by executing another real RAM program with input  $r$ . If  $r$  is an integer, the Commander can therefore compute any Turing-computable function on  $r$ , and so it can decide to move to  $D_1$  the first time, then to  $D_2$  twice, then to  $D_3$  three times, to  $D_1$  four times, and so on. This pattern of moves is illustrated in Figure 3, and of course it results in the exploration of the plane, because the visibility radius of the robots is much greater than the step  $\mu$ . ◀

## 4.2 Minimality of the Basic TuringMobile

We can use the previous result to prove indirectly that our basic TuringMobile design is minimal, because no team of fewer than three oblivious robots in  $\mathbb{R}^2$  can explore the plane.

► **Theorem 3.** *If only one or two oblivious identical robots with limited visibility are present in  $\mathbb{R}^2$ , they cannot explore the plane, even if the scheduler lets them move synchronously and rigidly.*

**Proof.** Assume that a single oblivious robot is given in  $\mathbb{R}^2$  (hence no other entities or obstacles are present). Since the robot always gets the same snapshot, it always computes the same destination point in its local coordinate system, and so it always translates by the same vector. As a consequence, it just moves along a straight ray, and therefore it cannot explore the plane.

Let two oblivious robots be given, and suppose that their local coordinate systems are oriented symmetrically. Whether the robots see each other or not, if they take their snapshots simultaneously, they get identical views, and so they compute destination points that are symmetric with respect to  $O$ . If they keep moving synchronously and rigidly,  $O$  remains their midpoint. So, if the robots have visibility radius  $V$ , they see each other if and only if they are in the circle  $\gamma$  of radius  $V/2$  centered in  $O$ .

Let  $O$  be the midpoint of the robots' locations, and consider a Cartesian coordinate system with origin  $O$ . Without loss of generality, when the robots do not see each other, they move by vectors  $(1, 0)$  and  $(-1, 0)$ , respectively. Let  $\xi$  be the half-plane  $y \geq V$ , and observe that  $\xi$  lies completely outside  $\gamma$ .

It is obvious that the robots cannot explore the entire plane if neither of them ever stops in  $\xi$ . The first time one of them stops in  $\xi$ , it takes a snapshot from there, and starts moving parallel to the  $x$  axis, thus never seeing the other robot again, and never leaving  $\xi$ . Of course, following a straight line through  $\xi$  is not enough to explore all of it. ◀

### 4.3 Near-Gathering with Limited Visibility

The exploration technique can be applied to several more complex problems. The first we describe is the *Near-Gathering* problem, in which all robots in the plane must get in the same disk of a given radius  $\varepsilon$  (without colliding) and remain there forever. It does not matter if the robots keep moving, as long as there is a disk of radius  $\varepsilon$  that contains them all.

It is clear that solving this problem from every initial configuration is not possible, and hence some restrictive assumptions have to be made. The usual assumption is that the initial visibility graph of the robots be connected [20, 24]. Here we make a different assumption: there are three robots that form a basic TuringMobile somewhere in the plane, and each robot not in the TuringMobile has distance at least  $\varepsilon$  from all other robots. (Actually we could weaken this assumption much more, but this simple example is good enough to showcase our technique.) Also, in the existing literature on the Near-Gathering problem it is always assumed that the robots agree on at least one coordinate axis, but here we do not need this assumption.

Say that all robots in the plane have a visibility radius of  $V \gg \varepsilon$ , and that the TuringMobile moves by  $\mu \ll \varepsilon$  at each step. The TuringMobile starts exploring the plane as above, and it stops in a rest position as soon as it finds a robot whose distance from the Commander is smaller than  $V/2$  and greater than  $\varepsilon$ . On the other hand, if a robot is not part of the TuringMobile, it waits until it sees a TuringMobile in a rest position at distance smaller than  $V/2$ . When it does, it moves to a designated area  $\mathcal{A}$  in the proximity of the Commander. Such an area has distance at least  $3d$  from the Commander, so no confusion can arise in the identification of the members of the TuringMobile. If several robots are eligible to move to  $\mathcal{A}$ , only one at a time does so: note that the layout of the TuringMobile itself gives an implicit total order to the robots around it. Observe that the robots cannot form a second TuringMobile while they move to  $\mathcal{A}$ : in order to do so, two of them would have to move to  $\mathcal{A}$  at the same time and get close enough to a third robot. Once they enter  $\mathcal{A}$ , the robots position themselves on a segment much shorter than  $d$ , so they cannot possibly be mistaken for a TuringMobile.

Once the eligible robots have positioned themselves in  $\mathcal{A}$ , the TuringMobile resumes its exploration of the plane, and the robots in  $\mathcal{A}$  copy all its movements. Of course, at each step the TuringMobile waits for all the robots in  $\mathcal{A}$  to catch up before carrying on with the exploration. Now, if the total number of robots in the plane is known, the TuringMobile can stop as soon as all of them have joined it. Otherwise, the machine simply keeps exploring the plane forever, eventually collecting all robots. In both cases, the Near-Gathering problem is solved.

#### 4.4 Pattern Formation with Limited Visibility

Suppose  $n$  robots are tasked with forming a given *pattern* consisting of a multiset of  $n$  points: this is the *Pattern Formation* problem, which becomes the *Gathering* problem in the special case in which the points are all coincident. For this problem, it does not matter where the pattern is formed, nor does its orientation or scale.

Again, the Pattern Formation problem is unsolvable from some initial configurations, so we make the same assumptions as with the Near-Gathering problem. The algorithm starts by solving the Near-Gathering problem as before. The only difference is that now there is a second tiny area  $\mathcal{B}$ , attached to  $\mathcal{A}$  (and still far enough from the TuringMobile), which the robots avoid when they join  $\mathcal{A}$ . This is because this second area will later be used to form the pattern.

Since  $n$  is known, the TuringMobile knows when it has to interrupt the exploration of the plane because all robots have already been found. At this point, the robots switch algorithms: one by one, they move to  $\mathcal{B}$  and form the pattern. This task is made possible by the presence of the TuringMobile, which gives an implicit order to all robots, and also unambiguously defines an embedding of the pattern in  $\mathcal{B}$ . So, each robot is implicitly assigned one point in  $\mathcal{B}$ , and it moves there when its turn comes.

If  $n = 3$  or  $n = 4$ , there are uninteresting ad-hoc algorithms to do this: so, let us assume that  $n \geq 5$ . The first to move are the robots in  $\mathcal{A}$ : this part is easy, because they all lie on a small segment, which already gives them a total order, and allows them to move one by one. The robots only have to be careful enough not to collide with other robots before reaching their final positions. Again, this is trivial, because only one robot is allowed to move at a time.

When this part is done, there are at least two robots in  $\mathcal{B}$ , all of which have distance much smaller than  $d$  from each other. Then the members of the TuringMobile join  $\mathcal{B}$  as well, in order from the closest to the farthest. Each of them chooses a position in  $\mathcal{B}$  based on the robots already there and the remnants of the TuringMobile. Moreover, the members of the TuringMobile that have not started moving to  $\mathcal{B}$  yet cannot be mistaken for robots in  $\mathcal{B}$ , because they are at a greater distance from all others (and vice versa).

Note that, when the last robot leaves the TuringMobile and joins  $\mathcal{B}$ , it is able to find its final location because there are already at least four robots there, which provide a reference frame for the pattern to be formed. When this last robot has taken position in  $\mathcal{B}$ , the pattern is formed.

#### 4.5 Higher Dimensions

Everything we said in this section pertained to robots in the plane. However, we can generalize all our results to robots in  $\mathbb{R}^m$ , for  $m \geq 2$ . Recall that, at the end of Section 3.2, we have described a TuringMobile for robots in  $\mathbb{R}^m$ , which can move within a specific plane  $\gamma$  exactly as a bidimensional TuringMobile, but can also move back and forth by  $\mu$  in all other directions orthogonal to  $\gamma$ .

Now, extending our results to  $\mathbb{R}^m$  actually boils down to exploring the space with a TuringMobile: once we can do this, we can easily adapt our techniques for the Near-Gathering and the Pattern Formation problem, with negligible changes.

There are several ways a TuringMobile can explore  $\mathbb{R}^m$ : we will only give an example. Consider the exploration of the plane described at the beginning of this section, and let  $P_i$  be the point reached by the Commander after its  $i$ th move along the spiral-like path depicted in Figure 3 ( $P_0$  is the initial position of the Commander).

Our  $m$ -dimensional TuringMobile starts exploring  $\gamma$  as if it were  $\mathbb{R}^2$ . Whenever it visits a  $P_i$  for the first time, it goes back to  $P_0$ . From  $P_0$ , it keeps making moves orthogonal to  $\gamma$  until it has seen all points in  $\mathbb{R}^m$  whose projection on  $\gamma$  is  $P_0$  and whose distance from  $P_0$  is at most  $i$ . Then it goes back to  $P_0$ , moves to  $P_1$ , and repeats the same pattern of moves in the section of  $\mathbb{R}^m$  whose projection on  $\gamma$  is  $P_1$ . It then does the same thing with  $P_2$ , etc. When it reaches  $P_{i+1}$  (for the first time), it goes back to  $P_0$ , and proceeds in the same fashion. By doing so, it explores the entire space  $\mathbb{R}^m$ .

Note that this algorithm only requires the TuringMobile to count how many moves it has made since the beginning of the execution: thus, the machine only has to memorize a single integer. The direction of the next move according to the above pattern is then obviously Turing-computable given the move counter.

## 5 Conclusions

We have introduced the TuringMobile as a special configuration of oblivious non-rigid robots that can simulate a rigid robot with memory. We have also applied the TuringMobile to some typical robot problems in the context of limited visibility, showing that the assumption of connectedness of the initial visibility graph can be dropped if a unique TuringMobile is present in the system. Our results hold not only in the plane, but also in Euclidean spaces of higher dimensions.

The simplest version of the TuringMobile (Section 3.1) consists of only three robots, and is the smallest possible configuration with these characteristics (Theorems 2 and 3). Our generalized TuringMobile (Section 3.2), which works in  $\mathbb{R}^m$  and simulates  $k$  registers of memory, consists of  $3m + 3k$  robots (Theorem 1). We believe we can decrease this number to  $m + k + 3$  by putting all the Number robots in the same basic component and adopting a more complicated technique to move them. However, minimizing the number of robots in a general TuringMobile is left as an open problem.

Our basic TuringMobile design works if the robots have the same radius of visibility, because that allows them to implicitly agree on a unit of distance. We could remove this assumption and let each of them have a different visibility radius, but we would have to add a fourth robot to the TuringMobile for it to work (as well as keep the TuringMobile small compared to *all* these radii).

In order to encode and decode arbitrary real numbers we used a function and its inverse, which in turn are computed using the arctan and the tan functions. However, using transcendental functions is not essential: we could achieve a similar result by using only comparisons and arithmetic operations. The only downside would be that such a real RAM program would not run in a constant number of machine steps, but in a number of steps proportional to the value of the number to encode or decode. With this technique, we would be able to dispense with the trigonometric functions altogether, and have our robots use only arithmetic operations and square roots to compute their destination points.

---

## References

- 1 C. Agathangelou, C. Georgiou, and M. Mavronicolas. A distributed algorithm for gathering many fat mobile robots in the plane. In *32nd ACM Symposium on Principles of Distributed Computing (PODC)*, pages 250–259, 2013.
- 2 N. Agmon and D. Peleg. Fault-tolerant gathering algorithms for autonomous mobile robots. *SIAM Journal on Computing*, 36(1):56–82, 2006.


- 3 A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Massachusetts, 1974.
- 4 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.
- 5 Q. Bramas and S. Tixeuil. The random bit complexity of mobile robots scattering. *International Journal of Foundations of Computer Science*, 28(2):111–134, 2017.
- 6 D. Canepa, X. Défago, T. Izumi, and M. Potop-Butucaru. Flocking with oblivious robots. In *18th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 94–108, 2016.
- 7 S. Cicerone, G. Di Stefano, and A. Navarra. Minimum-traveled-distance gathering of oblivious robots over given meeting points. In *10th International Symposium on Algorithms and Experiments for Sensor Systems (Algosensors)*, pages 57–72, 2014.
- 8 S. Cicerone, G. Di Stefano, and A. Navarra. Asynchronous pattern formation: The effects of a rigorous approach. In *arXiv:1706.02474*, 2017.
- 9 M. Cieliebak, P. Flocchini, G. Prencipe, and N. Santoro. Distributed computing by mobile robots: Gathering. *SIAM Journal on Computing*, 41(2):829–879, 2012.
- 10 R. Cohen and D. Peleg. Convergence properties of the gravitational algorithm in asynchronous robot systems. *SIAM Journal on Computing*, 36(6):1516–1528, 2005.
- 11 P. Courtieu, L. Rieg, S. Tixeuil, and X. Urbain. Impossibility of gathering, a certification. *Information Processing Letters*, 115(3):447–452, 2015.
- 12 P. Courtieu, L. Rieg, S. Tixeuil, and X. Urbain. Certified universal gathering in  $\mathbb{R}^2$  for oblivious mobile robots. In *30th International Symposium on Distributed Computing (DISC)*, pages 187–200, 2016.
- 13 S. Das, P. Flocchini, N. Santoro, and M. Yamashita. Forming sequences of geometric patterns with oblivious mobile robots. *Information Processing Letters*, 28(2):131–145, 2015.
- 14 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 (DISC)*, pages 46–60, 2006.
- 15 B. Degener, B. Kempkes, P. Kling, and F. Meyer auf der Heide. Linear and competitive strategies for continuous robot formation problems. *ACM Transactions on Parallel Computing*, 2(1):2:1–2:8, 2015.
- 16 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 (SPAA)*, pages 139–148, 2011.
- 17 G. Di Luna, P. Flocchini, N. Santoro, and G. Viglietta. Turingmobile: A turing machine of oblivious mobile robots with limited visibility and its applications. In *arXiv:1709.08800*, 2017.
- 18 P. Flocchini, G. Prencipe, and N. Santoro. *Distributed Computing by Oblivious Mobile Robots*. Morgan & Claypool, 2012.
- 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 P. Flocchini, G. Prencipe, N. Santoro, and P. Widmayer. Arbitrary pattern formation by asynchronous, anonymous, oblivious robots. *Theoretical Computer Science*, 407(1–3):412–447, 2008.
- 22 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.

- 23 T. Izumi, M. Gradinariu, and S. Tixeuil. Connectivity-preserving scattering of mobile robots with limited visibility. In *12th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 319–331, 2010.
- 24 P. Linda, G. Prencipe, and G. Viglietta. Getting close without touching: Near-gathering for autonomous mobile robots. *Distributed Computing*, 28(5):333–349, 2015.
- 25 F.P. Preparata and M.I. Shamos. *Computational Geometry*. Springer-Verlag, Berlin and New York, 1985.
- 26 M.I. Shamos. *Computational Geometry*. Ph.D. thesis, Department of Computer Science, Yale University, 1978.
- 27 I. Suzuki and M. Yamashita. Distributed anonymous mobile robots: Formation of geometric patterns. *SIAM Journal on Computing*, 28(4):1347–1363, 1999.
- 28 M. Yamashita and I. Suzuki. Characterizing geometric patterns formable by oblivious anonymous mobile robots. *Theoretical Computer Science*, 411(26–28):2433–2453, 2010.
- 29 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.

# Beeping a Deterministic Time-Optimal Leader Election

**Fabien Dufoulon**

LRI, Université Paris-Sud, CNRS, Université Paris-Saclay, Orsay, France  
dufoulon@lri.fr

 <https://orcid.org/0000-0003-2977-4109>

**Janna Burman**

LRI, Université Paris-Sud, CNRS, Université Paris-Saclay, Orsay, France  
burman@lri.fr

**Joffroy Beauquier**

LRI, Université Paris-Sud, CNRS, Université Paris-Saclay, Orsay, France  
beauquier@lri.fr

---

## Abstract

The *beeping model* is an extremely restrictive broadcast communication model that relies only on carrier sensing. In this model, we solve the *leader election* problem with an asymptotically optimal round complexity of  $O(D + \log n)$ , for a network of unknown size  $n$  and unknown diameter  $D$  (but with unique identifiers). Contrary to the best previously known algorithms in the same setting, the proposed one is deterministic. The techniques we introduce give a new insight as to how local constraints on the exchangeable messages can result in efficient algorithms, when dealing with the beeping model.

Using this deterministic leader election algorithm, we obtain a randomized leader election algorithm for anonymous networks with an asymptotically optimal round complexity of  $O(D + \log n)$  w.h.p. In previous works this complexity was obtained in expectation only.

Moreover, using deterministic leader election, we obtain efficient algorithms for symmetry-breaking and communication procedures:  $O(\log n)$  time MIS and *5-coloring* for tree networks (which is time-optimal), as well as  $k$ -source *multi-broadcast* for general graphs in  $O(\min\{k, \log n\} \cdot D + k \log \frac{nM}{k})$  rounds (for messages in  $\{1, \dots, M\}$ ). This latter result improves on previous solutions when the number of sources  $k$  is sublogarithmic ( $k = o(\log n)$ ).

**2012 ACM Subject Classification** Theory of computation → Distributed computing models, Theory of computation → Distributed algorithms, Theory of computation → Design and analysis of algorithms

**Keywords and phrases** distributed algorithms, leader election, beeping model, time complexity, deterministic algorithms, wireless networks

**Digital Object Identifier** 10.4230/LIPIcs.DISC.2018.20

**Related Version** A full version is available at <https://hal.archives-ouvertes.fr/hal-01794711>.

## 1 Introduction

The leader election (LE) problem, where a single (leader) node is given a distinguished role in the network, is a fundamental building block in algorithm design. This is because a leader can initiate and coordinate behaviors in the network, often making leader election a



© Fabien Dufoulon, Janna Burman, and Joffroy Beauquier;  
licensed under Creative Commons License CC-BY

32nd International Symposium on Distributed Computing (DISC 2018).

Editors: Ulrich Schmid and Josef Widder; Article No. 20; pp. 20:1–20:17

Leibniz International Proceedings in Informatics



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

crucial first step in applications requiring communication and agreement on a global scale. For example, more advanced communication primitives such as broadcast, gossiping and multi-broadcast, rely on a leader to coordinate transmissions [10] (see also Sect. 4.3).

Wireless networks with severe restrictions on communication capabilities are an increasingly prevalent subject of study, e.g., [6, 20, 7, 13, 14, 8]. In order to model these networks, Cornejo and Kuhn [7] introduced a convenient formal framework: the *discrete beeping model* ( $\mathcal{BEEP}$ ). In this model, time is divided into synchronous rounds, and in each round, a node can either listen or transmit a unary signal (beep) to all its neighbors. The possibility to directly transmit a beep to a node is defined by a *static communication graph*, and nodes have no knowledge of this graph. As a beep is merely a detectable burst of energy, a listening node does not receive the *identifiers* (ids) of its beeping neighbors. Even more critically, a beeping node receives no feedback, while a silent (listening) one can only detect that either at least one of its neighbors beeped or that all of them were silent. Although algorithms can take advantage of the synchronous nature of the rounds to transmit information using beeps, doing so impacts the time complexity in a quantifiable manner. This work studies how this impact can be minimized.

The beeping model has also been justified by its possible applications to biological networks [19], which are reliant on primitive communications. Fireflies communicate through flashes of light [2, 15] and cells through the diffusion of specific chemical markers [1, 22].

Beeps are an extremely limited form of communication, making it difficult to coordinate nodes. Being a fundamental coordination problem, leader election has received a lot of attention (see Sect. 1.1). Probabilistic and deterministic solutions were proposed for general graphs, and a time complexity lower bound of  $\Omega(D + \log n)$  was established ( $D$  is the diameter of the network, and  $n$  its size). A prime concern is the design of time-efficient *uniform* solutions, that is, not requiring any knowledge on the graph topology or on the parameters  $n$  and  $D$  (or even on their upper bounds). Indeed, it is unrealistic to assume that upper bounds on these parameters are always available, especially when considering dynamic networks. Amongst existing works on LE in  $\mathcal{BEEP}$ , the more difficult (for design) deterministic case has received less attention. However, this case is useful whenever random behavior is inappropriate or deterministic guarantees are required. We show in this work that an asymptotically time-optimal deterministic algorithm can be designed. This algorithm gives rise to an anonymous (not using ids) randomized algorithm that also matches the lower bound.

## 1.1 Related Work

Leader election (LE), being a fundamental problem in distributed computing, has been studied in various models. In each newly introduced model, an *efficient* leader election algorithm is a foremost concern, since it is frequently used as a building block in more complex algorithms. In particular, recent models designed for wireless networks assume that simultaneous communications interfere with each other. Consequently, leader coordination is even more important in these models, though LE is harder to solve efficiently.

Even though computational complexities (in particular time complexity) for LE are key aspects in the algorithmic design, additional properties are also of concern: for example, one might want nodes to detect termination, or to ensure that there is never more than one leader node during any execution (*safety property*).

Ghaffari and Haeupler [13] present the first LE algorithm for  $\mathcal{BEEP}$ , which elects a leader in  $O(D + \log n) \cdot O(\log^2 \log n)$  rounds with high probability (w.h.p.: with probability  $1 - n^{-\theta(1)}$ ). [13] also gives a lower bound of  $\Omega(D + \log n)$  rounds for LE, applicable both to



■ **Table 1** LE algorithms in the beeping model.

Reference	Round complexity	Safety	Knowledge
[13]	$O(D + \log n \log \log n) \cdot \min\{\log \log n, \log \frac{n}{D}\}$ w.h.p.	w.h.p.	$N = n^c$
[12]	$O(D \cdot \log n)$ deterministic time	Deterministic	None
[9]	$O(D + \log n)$ expected time	w.h.p.	$N = n^c$
Here	$O(D + \log n)$ deterministic time	Deterministic	None
Here	$O(D + \log n)$ w.h.p.	w.h.p.	$N = n^c$

deterministic and randomized (w.h.p. time) algorithms. This bound can be compared to the  $\Omega(D)$  lower bound in the *ECONGEST* model [16]. *ECONGEST* differs from *BEEP* in that any given node can send (different) messages of  $O(\log n)$  bits to each of its neighbors during a round. When nodes receive messages, there are no collisions and they can distinguish from which edge they received a particular message. Intuitively, since a beep can convey at most one bit, additional  $\Omega(\log n)$  rounds are necessary [18, 5, 11]. Following the result from [13], Czumaj and Davies [8, 9] presented a randomized LE algorithm with  $O(D + \log n)$  expected time in *BEEP*. In both randomized algorithms, the safety property is guaranteed w.h.p., but some upper bound  $N$  on the number of nodes  $n$  is required. As for deterministic LE, Förster et al. [12] give the first algorithm in *BEEP*, with an  $O(D \cdot \log n)$  round complexity. This algorithm is uniform in both  $n$  and  $D$ . The round complexities of different LE algorithms, including those presented in this work, are compared below (see Table 1).

It is mentioned in [13, 9] that upper bounds in *BEEP* apply to the well-known radio networks with collision detection (*RN-CD*). In *RN-CD*, nodes can send messages of  $O(\log n)$  bits (instead of beeps) and listening nodes receive a “collision” message if more than two neighbors communicate at the same time. For both models, previous results are not tight, especially for deterministic leader election.

[13, 8, 9, 12] concentrate on improving the time complexity of LE in general graphs, in *BEEP*. A different focus is presented in [14], where the goal is to minimize the size of the state machine representation of an algorithm solving randomized LE in single-hop networks.

Amongst the extensive leader election literature in other models, Casteigts et al. [5] is particularly relevant to this work. [5] proposes an  $O(D + \log n)$  time deterministic LE algorithm in the constant-size *ECONGEST* model, where the algorithm is uniform in both the number of nodes  $n$  and the diameter  $D$ . This model is much stronger than *BEEP*, in that a node can easily learn its local topology and has direct links to communicate with its neighbors, whereas the absence of such links in the beeping model causes interference and makes directed messages (with known sender and receiver) unachievable or plainly inefficient. Notice that by using a 2-hop coloring and by separating in time the transmission of messages, according to the colors of both the sender and receiver, the constant-size *ECONGEST* model can be simulated, but with a prohibitive multiplicative factor of  $O(\Delta^4)$  [3] (where  $\Delta$  is the maximum degree).

Nevertheless, one of the main contributions of [5] is a rooted (in the maximum id node) spanning tree construction and an *information diffusion* algorithm, designed to spread the maximum identifier efficiently, in a pipeline-like manner (rather than performing consecutive local comparisons on complete identifiers). This latter shift is crucial to the time-optimality of their algorithm, and is used here to improve on the  $O(D \cdot \log n)$  result from [12].

## 1.2 Contributions

We propose a deterministic and completely uniform (in  $n$  and  $D$ ) leader election algorithm with an  $O(D + \log n)$  asymptotically optimal round complexity. By independently sampling  $\theta(\log n)$  bits to create unique identifiers w.h.p. and using this algorithm, we obtain a uniform (in  $D$  only) randomized leader election algorithm which takes  $O(D + \log n)$  rounds w.h.p. and works in anonymous networks. Both solutions are the first to achieve time-optimality for these guarantees in both  $\mathcal{BEEP}$  and  $\mathcal{RN-CD}$ , outperforming all previous deterministic and randomized results. This work closes the gap between upper and lower bounds for LE.

Furthermore, using the proposed deterministic LE algorithm, we propose the first asymptotically time-optimal (in  $O(\log n)$  rounds) Maximal Independent Set (MIS) and 5-coloring algorithms for trees in  $\mathcal{BEEP}$  (leveraging the fact that given a leader in a tree network, it is simple to compute a 2-coloring). The MIS and coloring algorithms can be considered as essential symmetry-breaking procedures, and designing optimal-time solutions (even limited to tree networks) might be crucial for other applications in  $\mathcal{BEEP}$ .

Then, we give an  $O(\min\{k, \log n\} \cdot D + k \log \frac{nM}{k})$  time  $k$ -source multi-broadcast (with provenance) algorithm (for messages in  $\{1, \dots, M\}$ ). This latter algorithm improves a previous result by Czumaj and Davies [8], when the number of sources  $k$  is sublogarithmic ( $k = o(\log n)$ ), by executing  $k$  consecutive leader elections. Communication primitives are especially important in  $\mathcal{BEEP}$ , as they allow to deal with the interferences caused by simultaneous communications, and thus to design complex algorithms.

## 2 Model and Definitions

### 2.1 Preliminaries

The *communication network* is represented by a simple connected undirected graph  $G = (V, E)$ , where  $V$  is the node set and  $E$  the edge set. The *network size*  $|V|$  is also denoted by  $n$ , and the *diameter* by  $D$ . Nodes have unique identifiers (ids). This property is essential in order to break symmetry in deterministic algorithms. The *identifier* of a node  $u \in V$ ,  $id(u)$ , is an integer from  $\{1, \dots, U\}$  where  $U$  is some upper bound unknown to nodes. Then, the *maximum length* over all identifiers in  $G$  is  $O(\log U)$  (also unknown). For simplicity, we make the common assumption that identifiers have logarithmic (in  $n$ ) length, i.e., the id space is  $\{1, \dots, N\}$  where  $N = n^c$  for some unknown constant  $c > 1$ . In Sect. 3.3, we explain how the results of the paper apply to an arbitrary id space setting.

Now, we give definitions pertaining to (binary) words. The empty word is denoted by  $\epsilon$ . The operator  $\parallel$  is for the *word concatenation*. The *length* of a word  $x$  is denoted by  $|x|$ . For any word  $x$  and integer  $j \in \{1, \dots, |x|\}$ ,  $x[j]$  denotes the  $j^{\text{th}}$  *most significant bit* of  $x$ . Let  $x$  and  $y$  be two words (of possibly different lengths),  $x$  is said to be the *prefix* (resp., *proper prefix*) of  $y$  if there exists a word (resp., non empty word)  $z$  such that  $x \parallel z = y$ . Moreover,  $x$  is said to be *higher* than  $y$ , denoted by  $x \succ y$ , if  $y$  is a proper prefix of  $x$ , or if  $x[j] > y[j]$  for the first differing bit  $j$  (even if  $|x| < |y|$ ).

The  $\alpha$ -encoding [5] of an integer  $i \in \mathbb{N}^{>0}$  is a word obtained from the binary representation *bin* of  $i$ . By definition,  $\alpha(i) = 1^{|bin|} \parallel 0 \parallel bin$ . In the proposed LE algorithm (Sect. 3), instead of ids, nodes compare their  $\alpha$ -encodings ( $\alpha$ -ids). Finding the highest  $\alpha$ -id is equivalent to finding the maximum id, and can be performed uniformly (without padding the binary representations of ids) using bit-wise comparisons. A word  $x$  is *well-formed* if there exists an integer  $i$  such that  $x = \alpha(i)$ . It is simple to prove that for every word  $x$ , there is at most one such integer  $i$ . Thus the  $\alpha^{-1}$  function ( $\alpha$ 's “inverse”) is defined on well-formed words.

## 2.2 Model Definitions

In the *beeping model* ( $\mathcal{BEEP}$ ), an execution proceeds in synchronous rounds, i.e., there are synchronized local clocks and all nodes start at the same time, in a *synchronous start*. In each round, nodes synchronously execute the following steps. First, each node beeps (instruction *BEEP* in algorithms) or listens (*LISTEN* in algorithms). Beeps are transmitted to all neighbors of the beeping node. Then, if a node beeped (in the previous step of the same round), it learns no information from its neighbors. Otherwise, it knows whether or not at least one of its neighbors beeped (during the previous step of the same round). Finally, each node performs local computations. The synchronous start assumption can be replaced by a slightly weaker variant called *wake-on-beep* [1], for a constant multiplicative overhead (and an additive factor of  $O(D)$  rounds). In this variant, a node starts spontaneously either at an arbitrary time, or at one of its neighbors' beep, whichever happens first. Upon waking up, a node beeps immediately, to wake up its neighbors. As a result, the local clocks of two neighboring nodes differ by at most 1. Therefore, nodes can use phases of 3 rounds [1] (in which the node can beep or listen in the central round, and listens in both other rounds) to simulate rounds of a synchronous start execution.

We adopt the usual definitions for the system/algorithm: *state* of a node (values of its variables), *configuration* (a vector of all the nodes' states), *execution* (a sequence of configurations at consecutive rounds' ends), *terminal configuration* (a configuration repeated indefinitely), *termination* (when a terminal configuration has been reached), *round complexity* (number of rounds needed until a terminal configuration satisfying the problem conditions is reached, in the worst case). A variable *var* of a node  $v$  is explicitly associated to  $v$  using a subscript  $var_v$ . An algorithm is said to be *uniform*<sup>1</sup> in a parameter  $p$  if the algorithm is not given  $p$  (and is unable to infer it from the information it receives). For example, in a uniform (in  $n$ ) algorithm, nodes do not know the size  $n$  of the network, neither can they deduce it from their identifier. Notice that the variable size of the identifiers gives an unusable upper bound on the network size, as it leads to an excessive and unrealistic time complexity.

## 2.3 Leader Election

In the *leader election* (LE) problem, each node has a boolean variable, indicating a *leader* or a *non-leader* state. During an execution, there is never more than one leader (*safety* property). Initially, all nodes are non-leaders. Every execution terminates, and at the termination there is exactly one leader.

Now we give auxiliary definitions. First, we define *eventual leader election*, where the algorithm terminates but no node can detect this. Then, we define *terminating leader election*, where the algorithm terminates and all nodes detect when there remains a single candidate node (the leader). We solve *explicit leader election* (when nodes have unique identifiers): a terminating leader election in which all nodes know the elected leader's identifier at the termination.

---

<sup>1</sup> It is known that termination detection is easy in a synchronous setting whenever particular parameters related to the size of the communication graph are known, i.e., non-uniform terminating algorithms are easier to construct than the uniform ones.

**Algorithm 1** Uniform Eventual Leader Election Algorithm.

---

```

1: IN:  $id$ : identifier ; OUT:  $leader$ : boolean,  $leaderId$ : identifier
2:  $candidate := true$ ,  $prefix := \epsilon$ ,  $suspicious := false$  ▷  $\epsilon$  is the empty word
3:  $leaderId := 0$ ,  $leader := false$  ▷  $id$  and  $leaderId$  are ids, from  $\{1, \dots, N\}$ 
4: for diffusion phase  $p := 1$  ;  $p++$  do
5:   // First, a communication phase with  $c$  rounds.
6:   Communicate  $(prefix, suspicious)$  to all neighboring nodes.
7:   // Then, apply predicates of rules 1 to 5 on received  $(prefix, suspicious)$  pairs.
8:   Use received  $(prefix, suspicious)$  pairs to update  $prefix$ ,  $candidate$  and  $suspicious$ 
9:   if not  $candidate$  then  $leader := false$ 
10:  else if  $prefix = \alpha(id)$  then  $leader := true$ 
11:  if  $prefix$  is well-formed then  $leaderId := \alpha^{-1}(prefix)$ 

```

---

**3 Leader Election Algorithms**

Classical approaches used to solve leader election in *CONGEST* models do not directly apply to *BEEP*. Although they can be adapted using a transformer, doing so is too costly in most communication graph topologies (see discussion in the related work section: Sect. 1.1).

To solve the strongest version of LE, the explicit leader election, we proceed in two main steps. First, we design a uniform algorithm for eventual leader election, in Sect. 3.1. Then, in Sect. 3.2, we combine this algorithm with a specially designed uniform termination detection component to obtain a uniform explicit leader election algorithm. Finally, in Sect. 3.3, we discuss how the presented algorithm can be applied to other settings (e.g., arbitrary id range).

**3.1 Uniform Eventual Leader Election**

The algorithm (Algorithm 1) is described first (Sect. 3.1.1). Then, in Sect. 3.1.2,  $k$ -balanced messages are presented. They are used to allow constant-size *communication phases* composed of rounds and dedicated to the communication of (large) messages respecting local constraints. Using the  $k$ -balanced message technique, a detailed description of the communication phases (appearing in Algorithm 1) is given in Sect. 3.1.3. Finally, in Sect. 3.1.4, we relate the presented techniques to existing works in *CONGEST* models.

**3.1.1 Description**

All nodes aim to spread their  $\alpha$ -identifiers ( $\alpha(id)$  in Alg. 1) to the whole network (*information diffusion algorithm*). They execute loosely synchronized bit-wise comparisons and propagate the bits of the highest detected prefix (of  $\alpha$ -id). All nodes start out as *candidates*, with two variables:  $prefix$  and  $suspicious$ . The binary word  $prefix$  is initialized to the empty word  $\epsilon$  and represents the prefix of an  $\alpha$ -id. Most of the time, it represents the highest prefix of which the node is aware. Each node adapts its  $prefix$  by adding or removing the less significant bits, depending on the information gathered. The boolean  $suspicious$  is initialized to *false* and indicates whether the node removed bits from  $prefix$  in the last phase.

Nodes execute *diffusion phases* (of  $c$  rounds each) synchronously. A diffusion phase consists of one *communication phase* of  $c = O(1)$  rounds (line 6), used to send  $prefix$  and  $suspicious$  to all neighbors, followed by a (limited) modification of  $prefix$ .

The communication phase is described in detail in Sect. 3.1.3. In the same phase, each node receives  $(prefix, suspicious)$  pairs from its neighbors, but does not know which node sent which message, nor how many nodes sent any of these messages (*multiplicity*).

After the communication phase, any node  $v$  checks if  $prefix_v$  is a *locally higher prefix*, using the received pairs (see details below) and the previously gathered information. If this is the case, it appends a bit from its  $\alpha$ -id to  $prefix_v$  (if  $prefix_v$  is a proper prefix of  $\alpha(id_v)$ ), or does nothing (if  $prefix_v = \alpha(id_v)$ ). Otherwise, it modifies  $prefix_v$  depending on the highest detected  $prefix$  value, and becomes a *follower*. It can no longer become a leader. If that modification removes bits from  $prefix_v$ , node  $v$  is said to be *suspicious* for the following phase, and  $suspicious_v$  is assigned to *true* for one phase.

The five rules below associate conditions (predicates) to actions. A predicate evaluated to *true* triggers the associated action. In line 8, these predicates are evaluated (by some node  $v$ ) on the set of the received  $(prefix, suspicious)$  pairs, in the given order of priority, and the first triggered action is performed.

1. If there exists a suspicious neighbor  $u$ , such that  $prefix_u$  is a proper prefix of  $prefix_v$ , remove  $\min\{|prefix_v| - |prefix_u|, 3\}$  letters from the end of  $prefix_v$ .
2. If  $prefix_v = (z \parallel 0 \parallel w)$  with  $w \neq \epsilon$  and there exists a neighbor  $u$  with  $prefix_u = (z \parallel 1 \parallel y)$ , delete  $|w|$  letters from the end of  $prefix_v$ .
3. If  $prefix_v = (z \parallel 0)$  and there exists a neighbor  $u$  with  $prefix_u = (z \parallel 1 \parallel y)$ , then change  $prefix_v$  to  $(z \parallel 1)$ .
4. If there exists a neighbor  $u$  with  $prefix_u = (prefix_v \parallel 1 \parallel w)$  then append 1 to  $prefix_v$ .
5. If there exists a neighbor  $u$  with  $prefix_u = (prefix_v \parallel 0 \parallel w)$  then append 0 to  $prefix_v$ .

If any of the predicates (of the rules 1-5) is true,  $prefix_v$  is not a locally higher prefix. Indeed, if a neighbor  $u$  (of  $v$ ) is suspicious and  $prefix_u$  is a proper prefix of  $prefix_v$ , then a neighbor of  $u$  has a higher prefix than  $prefix_v$ , or is changing its  $prefix$  according to rule 1 above. By deleting the last bits of  $prefix_v$ , node  $v$  is matching  $prefix_v$  to an unknown but higher  $prefix$ . In all 4 other cases,  $prefix_u$  is clearly a higher prefix than  $prefix_v$ , therefore  $prefix_v$  modifies (a limited amount of) its last bits to more closely match  $prefix_u$ .

Additional local computations in lines 9-11 conclude a diffusion phase. Once a candidate's  $prefix$  variable is well-formed (i.e., once  $id_v = \alpha^{-1}(prefix_v)$ ), this node becomes a leader. If in later rounds it becomes a follower, then it withdraws from the leader role. Although this process violates the safety property, it is necessary in order to elect a leader, as the last remaining candidate cannot detect that it is the last, due to the lack of termination detection in this preliminary eventual LE version.

The 5 rules described above are an idea adopted from [5]. Thus the described information diffusion process satisfies Lemma 1 and Theorem 2 below, adopted from the results of [5] and adapted here to our beeping algorithm (see Sect. 3.1.4 for more details).

► **Lemma 1** (Beeping version of Lemma 8 in [5]). *Let  $u$  and  $v$  be two neighboring nodes. Then,  $prefix_u$  and  $prefix_v$  are identical, except in at most 6 (least significant) bits: without loss of generality, from the  $|prefix_u|^{th}$  bit (possibly included) to the  $|prefix_v|^{th}$  bit. Note that if the  $|prefix_u|^{th}$  bit differs in  $prefix_u$  and  $prefix_v$ , then  $||prefix_u| - |prefix_v|| < 6$*

► **Theorem 2** (Beeping version of Theorem 10 in [5]). *Let  $X$  be the maximum identifier. After  $|\alpha(X)| + 6r$  phases of the information diffusion algorithm, all nodes within distance  $r$  (for any  $r \geq 0$ ) from the node with id  $X$  have  $prefix = \alpha(X)$ . Thus, after at most  $|\alpha(X)| + 6D$  phases, for each node  $v$ ,  $prefix_v = \alpha(X)$ , and there is a unique candidate node.*

**Proof.** Let  $l$  be the maximum id node. We prove the theorem by induction on  $r$ . Node  $l$  has the maximum identifier  $X$ , thus it appends a bit from  $\alpha(X)$  in each diffusion phase. After  $|\alpha(X)|$  phases,  $prefix_l = \alpha(X)$ . This concludes the case when  $r = 0$ . For the induction step ( $r > 0$ ), consider any given node  $u$  at distance  $r + 1$  of node  $l$ , and one of its neighbors  $v$  at distance  $r$  from  $l$ . By Lemma 1,  $prefix_u$  and  $prefix_v$  differ in less than 6 bits. After  $|\alpha(X)| + 6r$  phases, since  $prefix_v = \alpha(X)$  (induction hypothesis), node  $v$  does not modify  $prefix_v$  and node  $u$  necessarily corrects (removes, changes or adds) at least one of  $prefix_u$ 's bits in each of the 6 following phases, until  $prefix_u = \alpha(X)$ . ◀

Recall that a communication phase is composed of  $c = O(1)$  rounds ( $c$  is defined in Sect. 3.1.3). This implies the following theorem.

► **Theorem 3.** *Uniform Eventual Leader Election is solved by Algorithm 1 in  $O(D + \log n)$  rounds (in the beeping model).*

**Proof.** Let  $v$  be any given node and  $X$  the maximum identifier in the network. From Theorem 2,  $prefix_v = \alpha(X)$  after  $O(D + \log n)$  phases. Nodes have the leader's identifier by applying the  $\alpha^{-1}$  function. Moreover, the maximum id node is well-formed after  $|\alpha(X)| = O(\log n)$  phases, thus after  $O(\log n)$  rounds. As a result, the maximum id node is, and remains, a leader henceforth. ◀

### 3.1.2 Balanced messages

A basic design technique called multi-slot design pattern [4], allows to communicate constant-size messages without the sender's id nor multiplicity, given a synchronous start. It works in communication phases of  $M$  rounds, if at most  $M$  different messages (in  $\{1, \dots, M\}$ ) are allowed. Beeping in the  $j^{th}$  round of a phase is equivalent to sending the message  $j$ . However, receivers cannot detect which (and how many) nodes sent that message. Thus, due to the beeping model's restrictions, if a node sends a message  $m$ , it receives no information about whether any of its neighbors also did.

Clearly, this design technique cannot be used to directly send *prefix* values, as these values are in  $\{1, \dots, N\}$ , and communication phases would be  $O(N)$  rounds long. But this technique can be adapted to send the values of a locally constrained ( $k$ -balanced) variable. A variable  $var$  is said to be  $k$ -balanced if it satisfies the *k-balancing property*, that is, if the difference between neighboring  $var$  values is at most  $k$  (for every node  $v$  and neighboring node  $u$ ,  $|var_u - var_v| \leq k$ ).

If one wishes to communicate  $k$ -balanced messages, then it is enough to transmit, for a message  $m$ , the *remainder*  $r = m \bmod(1 + 2k)$ , using the previous technique, with phases of  $M = 1 + 2k$  rounds (where  $k$  is known a priori to all nodes). Then, the receiver knowing at the same time its own remainder, the sender's remainder and the fact that the messages are  $k$ -balanced, can deduce the originally sent message (but does not know if multiple nodes have sent this message). Specifically, let  $v$  be the receiver and  $u$  the sender. Node  $v$  deduces the original message  $m_u$  from the received remainder message  $r_u$ :  $m_u = m_v + r_u - r_v - \lfloor \frac{r_u - r_v}{k+1} \rfloor M$ .

Consider the example depicted in Table 2 for  $k = 4$ . For a given node  $v$ , any message  $m_u$  sent by a neighboring node  $u$  is in  $\{m_v - k, \dots, m_v + k\}$ . By transmitting the remainder  $r_u = m_u \bmod(1 + 2k)$ , node  $u$  indicates whether its message  $m_u$  is in the next 4 values or in the previous 4, respectively to  $m_v$ , and the exact position amongst the 4 possibilities (more precisely, through  $r_u - r_v$ ). The remaining  $-\lfloor \frac{r_u - r_v}{k+1} \rfloor M$  factor deals with the fact that some lower (than  $m_v$ ) messages  $m_u$  result in a high remainder  $r_u$ , and some higher messages  $m_u$  in a low remainder  $r_u$ , due to the modulo operation. Node  $v$  can deduce the message  $m_u$  by using all of this information, along with its own message  $m_v$ .

■ **Table 2** Communication of  $k$ -balanced messages, where  $k = 4$  and  $M = 9$ . The executing node  $v$ , and its message value  $m_v$ , are highlighted. If  $v$  receives a message  $r_u = 3$ , it is able to deduce that the corresponding message  $m_u$  is 21.

Received remainder $r_u =$	'0'	'1'	'2'	'3'	'4'	'5'	'6'	'7'	'8'
$m_u - m_v =$ $(r_u - r_v) - \lfloor \frac{r_u - r_v}{k+1} \rfloor M$	-1	$v$	+1	+2	+3	+4	-4	-3	-2
Decoded message $m_u =$	18	19	20	21	22	23	15	16	17

The  $k$ -balanced message technique is of independent interest, and allows efficient algorithm design when nodes communicate locally-similar values.

### 3.1.3 Designing constant-size communication phases

In this section, we show that by applying the balanced messages approach, using only  $O(1)$  beeping rounds, a node can deduce its neighbors' *prefix* values (and whether some of them are suspicious), even though there are  $O(N)$  different possible values of *prefix*.

From Lemma 1, we know that  $|prefix|$  is a 6-balanced variable. Moreover, two neighboring nodes have similar *prefix* values, which differ only in (at most 6 of) the last bits. Therefore, if a node can learn the last 6 bits of a neighboring *prefix*, and their exact positions, then it can fill up the empty bits (in more significant positions) using the bits from its own *prefix*. To learn that, one could use two consecutive communication subphases: the first communicates the *position* of the last bit (which is  $|prefix|$ , a 6-balanced variable) in a subphase with 13 rounds, and the second communicates an *ending* message with the last 6 bits (using a message from  $\{1, \dots, 2^6\}$ , encoding all possible 6 letters combinations), in a subphase with 64 rounds. However, this does not work in  $\mathcal{BEEP}$  because one needs to know, for every different ending message sent by neighbors, the corresponding position of the last bit (thus the corresponding position message). Although this is trivial in  $\mathcal{ECONGEST}$ , because messages from different neighbors are received on different edge ports, it is too costly to simulate this functionality in  $\mathcal{BEEP}$  (see Sect. 1.1). Fortunately, as the message space is constant-size in both of these communication subphases, the Cartesian product of both message spaces is also constant-size. This allows to associate position and ending messages, using  $O(1)$  rounds, even in  $\mathcal{BEEP}$ . Consequently, communication phases with 832 rounds (for messages in  $\{1, \dots, 13\} \times \{1, \dots, 2^6\}$ ) are needed to communicate enough information for a node to deduce all neighboring *prefix* values.

On top of that, the nodes also need to communicate the boolean *suspicious*. For this reason, the message space is adapted to  $\{1, \dots, 13\} \times \{1, \dots, 2^6\} \times \{false, true\}$ . This results in communication phases (introduced in Algorithm 1, Sect. 3.1.1) of length  $c = 1664$  rounds, which although large, is still  $O(1)$  size.

### 3.1.4 Remarks on the eventual leader election algorithm

As mentioned in the related work (Sect. 1.1), [5] is particularly relevant to our work. In this section, we discuss this in detail.

The structure of the information diffusion algorithm is essentially the same. The algorithm progresses in diffusion phases, consisting of a communication phase (corresponding to a single round in the considered  $\mathcal{ECONGEST}$  model) where nodes send their (*prefix*, *suspicious*)



values, after which nodes change their *prefix* variable depending on the  $(prefix, suspicious)$  pairs received. Recall the 5 rules presented in Sect. 3.1.1: the set of the different possible changes for the *prefix* variable is of a constant size, and these changes are meant to affect at most a constant number of (the last) bits of *prefix*. An important point in [5] is the proof that this set of changes allows the maximum identifier to spread over the network, in an optimal  $O(D + \log n)$  number of phases. We use the same constant-size set of changes (for *prefix*). That is why Lemma 1 also applies to our algorithm.

However, the other core element of their information diffusion algorithm, the communication phase, cannot be used in  $\mathcal{BEEP}$ . In [5], nodes maintain up-to-date copies of the *prefix* variables of their neighbors to circumvent the limited message size and can keep these copies up-to-date in a single communication phase of  $O(1)$  rounds. In such a phase, nodes communicate what change was carried out (and which neighbor sent which message): sending the type of change is equivalent to sending the complete *prefix* value in this situation. In  $\mathcal{BEEP}$ , nodes are unable to know which neighbor sent which message. Although this capability can be simulated, it is unlikely that it can be done without increasing the time complexity of [5]. Current methods result in a  $O(\Delta^4)$  multiplicative factor (see discussion in Sect. 1.1).

One of the main contributions of this work is the introduction of the *k-balanced message* method to leverage the local constraints between (unbounded) values, which allows to communicate in  $O(1)$  rounds. With the *k-balanced message* technique, a node can transmit a value of *prefix* to its neighbors in  $O(1)$  rounds (of  $\mathcal{BEEP}$ ) only. This communication process differs greatly from that of [5].

## 3.2 Uniform Terminating Leader Election (Explicit LE)

Being often used as a primitive, the LE algorithm must be uniform and detect termination (e.g., so that it can be composed with other algorithms). Since classical approaches are not suited to  $\mathcal{BEEP}$ , we propose an explicit leader election algorithm using a different termination detection approach. Notice that, as mentioned previously, it is simple (in a synchronous setting) to transform the uniform eventual leader election algorithm, Algorithm 1, into a *non-uniform* one using knowledge of  $D$  and  $N$ , and thus of the time complexity expressed in terms of these parameters. Then, candidates can wait until the algorithm terminates, by counting rounds corresponding to the evaluated time complexity. However, this technique cannot be used here.

Instead, we use a primitive called *overlay networks*. We briefly describe it in Sect. 3.2.1. Then, in Sect. 3.2.2, an adapted version of this primitive is used to create a uniform termination detection component. This component is combined with the previously presented eventual leader election algorithm to obtain uniform explicit leader election.

### 3.2.1 Overlay network

The overlay network approach, in the context of leader election, was first used for  $\mathcal{BEEP}$  in [12]. Such an overlay has a designated root, and consists of layers centered around the root. Nodes at a distance  $d$  from that root (*level d*), have *up links* (resp. *down links*) towards all neighboring nodes (of the overlay) at distance  $d - 1$  (resp. at distance  $d + 1$ ) from the root. Using these (virtual) links, the root can gather information about the network and disseminate it. The default behavior for non-root overlay nodes is to relay any beep received over an up (resp. down) link in some phase  $p$ , to all down (resp. up) links in phase  $p + 1$ .



**Algorithm 2** Uniform Terminating Leader Election Algorithm.

---

```

1: IN: id: identifier ; OUT: leader: boolean, leaderId: identifier
2: candidate := true, prefix :=  $\epsilon$ , suspicious := false ▷  $\epsilon$  is the empty word
3: leaderId := 0, leader := false
4: for diffusion phase  $p := 1$  ;  $p++$  do
5:   // First, a communication phase with  $c = O(1)$  rounds.
6:   Communicate (prefix, suspicious) to all neighboring nodes.
7:   // Then, apply predicates of rules 1 to 5 on received (prefix, suspicious) pairs.
8:   Use received (prefix, suspicious) values to update prefix, candidate and suspicious.
9:   // Finally, termination detection phase with  $s = O(1)$  rounds.
10:  Execute a termination detection phase.
11:  if candidate and prefix =  $\alpha(id)$  then
12:    | If no beep is heard in down links, exit the loop.
13:  else
14:    | If a beep is heard in up links, exit the loop.
15:  leaderId :=  $\alpha^{-1}(prefix)$ 
16: if candidate then leader := true ▷ Last candidate becomes the leader

```

---

In more detail, overlays work in the following way. Time is divided into *overlay phases* of 9 rounds, where each phase consists of 3 subphases of 3 rounds each. The first 3 rounds are called *control* rounds, the next 3 - *up* rounds and the last 3 - *down* rounds. Each set of 3 rounds is numbered from 0 to 2.

When nodes join the overlay, they initialize a *depth* variable (in  $\{0, 1, 2\}$ ), through which they know some information about their layer (and thus how to communicate with the other layers). The root node *joins* the overlay at a given time, and assigns itself *depth* := 0. The other nodes willing to join the overlay listen in all control rounds. Since overlay nodes beep in the control round *depth* (in all overlay phases), the joining nodes assign themselves *depth* = *beepHeard* + 1 (mod 3), where *beepHeard* is the smallest control round in which a beep was heard.

It is important that the *depth* variable satisfies some local constraints, to be guaranteed by the joining process. More specifically, for any distance  $d$  and for any given (overlay) node  $v$  in level  $d$ , all neighboring (overlay) nodes  $u$  in level  $d - 1$  (resp. in level  $d + 1$ ) must have  $depth_u = depth_v - 1 \pmod{3}$  (resp.  $depth_u = depth_v + 1 \pmod{3}$ ), where  $-1 \pmod{3} = 2$ . With this property, nodes can listen over an up link (resp. down link) by listening in up (resp. down) round  $depth - 1 \pmod{3}$  (resp.  $depth + 1 \pmod{3}$ ). Moreover, nodes beep over an up link (resp. down link) by beeping in up (resp. down) round  $depth \pmod{3}$ . In other words, communication through up and down links is the same as sending, or listening for, a *depth* message (using the multi-slot design pattern from [4], described in Sect. 3.1.2) using the corresponding subphase (a message from  $M_{depth} = \{0, 1, 2\}$ ).

### 3.2.2 Termination detection component for explicit leader election

We describe the proposed *termination detection component* and its interactions with the eventual leader election algorithm (Algorithm 1). The termination detection component is meant to gather information from the whole network, on whether there are any candidates with a higher  $\alpha$ -id. If there are none, the last candidate terminates and becomes leader. The combined final algorithm structure is given in Algorithm 2.

As in Algorithm 1, time is divided into diffusion phases, but these phases now include an additional *termination detection phase*. A termination detection phase consists of a border detection phase followed by an adapted overlay phase. The *border detection phase* is a communication phase for messages in  $M_{prefix} = \{1, \dots, 13\} \times \{1, \dots, 2^6\}$ , where nodes can detect if any of their neighbors has a different *prefix* value (similar to the communication in Sect. 3.1.3). If that is the case for an overlay node (even the root) which has been part of the overlay for more than 6 phases, this node becomes a *border node* (i.e., there exists a neighbor with a different *prefix* value). The *adapted overlay phase* is a communication phase with 3 subphases, each for messages in  $M_{depth} \times M_{prefix}$ . Each adapted overlay network is associated to a specific *prefix* (i.e., that of the overlay's root, necessarily a candidate node). This prefix is used (communicated) so that nodes can detect whether the other endpoint of a down link or up link, is part of the same overlay (i.e., has the same *prefix*). Consequently, different overlay networks do not interfere with each other. A border detection phase has  $|M_{prefix}|$  rounds and an adapted overlay phase has  $9|M_{prefix}|$  rounds, thus a termination detection phase has  $s = 10|M_{prefix}|$  rounds.

Upon having a well-formed *prefix*, each candidate designates itself as root and starts constructing an overlay network by using the termination detection phase. Nodes which have just joined the overlay and border nodes beep in their up links (relayed all the way back to the root) using the adapted overlay phase. As a result, the root hears beeps in its down links in each (termination detection) phase, until the overlay network covers the whole graph (Lemma 5). Moreover, the only overlay that can cover the whole graph is the overlay of the highest  $\alpha$ -id node (because this node never changes *prefix* depending on another node's  $\alpha$ -id, and consequently never joins another candidate's overlay). Therefore, when the root hears no beeps in its down links (and is not a border node), it knows that its overlay covers the whole graph, and that it is the highest  $\alpha$ -id node (thus the maximum id node). All other roots hear beeps in down links (or become border nodes), until their *prefix* is changed.

In more detail, the *construction* of the adapted overlay networks is done as follows. Once a candidate node has a well-formed *prefix* (after exactly  $|\alpha(id)|$  diffusion phases), it sets itself up as an overlay's root (in phase  $p = |\alpha(id)|$ ), but it stays silent for 6 termination detection phases (from phase  $p$  to phase  $p + 5$ ) before beeping in the control rounds of phase  $p + 6$  (and only in this phase). On the other hand, follower nodes with a well-formed *prefix* attempt to join the overlay corresponding to *prefix* right away. Once a follower node joins an overlay (in phase  $p'$ ), it also stays silent for 6 termination detection phases before beeping in the control rounds of phase  $p' + 6$ . For any given node  $v$  that joins an overlay in termination detection phase  $p'$ , its neighbors know if they join  $v$ 's overlay or not, by phase  $p' + 6$  at the latest (by Theorem 2). By staying silent for 6 termination detection phases upon joining,  $v$  ensures that all of its neighbors join the overlay at the same time (if they choose to join). Consequently, two nodes  $u$  and  $v$ , at the same distance  $d$  from an overlay's root  $r$ , never join  $r$ 's overlay in different termination detection phases, and  $depth_u = depth_v$ . Otherwise, we could have  $depth_u \neq depth_v$ , which means a common neighbor of  $u$  and  $v$  at distance  $d - 1$  from  $r$  would not have properly defined down links.

► **Lemma 4.** *Let  $r$  be the root of an overlay network. This overlay is properly constructed, that is, nodes at level  $d$  have the same depth value.*

**Proof.** Let us prove by induction that if a node at distance  $d$  from  $r$  joins  $r$ 's overlay, then it is in phase  $|\alpha(id_r)| + 6d$ . Let us first consider a node  $v$  at distance 1 from  $r$ . For node  $v$  to join  $r$ 's overlay, another overlay node must beep in the control rounds and  $prefix_v$  must be equal to  $\alpha(id_r)$ , in the same phase. Notice that for any given two neighbors  $u$  and  $v$ , which are in different overlays, both nodes beep in different control rounds, because  $prefix_u \neq prefix_v$ .

In phase  $|\alpha(id_r)| + 6$ ,  $r$  beeps in the control rounds, and thus  $v$  can join in that phase (if  $prefix_v = \alpha(id_r)$ ). In addition, if  $prefix_v \neq \alpha(id_r)$  in phase  $|\alpha(id_r)| + 6$ , then by Theorem 2, node  $v$  does not consider  $\alpha(id_r)$  as the highest  $prefix$  value it has encountered. As a result, it is impossible that  $prefix_v = \alpha(id_r)$  after phase  $|\alpha(id_r)| + 6$ , and that  $v$  joins  $r$ 's overlay after phase  $|\alpha(id_r)| + 6$ . The induction step ( $d > 1$ ) is similar, starting from a node  $v$  at distance  $d$  from  $r$ . ◀

Because the adapted overlay networks are properly constructed, we can prove that as long as an overlay has not covered the whole network, follower nodes beep in their up links, stopping the root from becoming a leader. In more detail, after a candidate node beeps in the control rounds, it listens to its down links in every termination detection phase. As long as it hears a beep in these links, or is a border node, it does not become leader. Once no beep is heard, it becomes leader, sends a beep in its down links and terminates. On the other hand, after a follower node joins the overlay (in phase  $p$ ), it beeps in its up links in the first 7 termination detection phases (from phase  $p$  to phase  $p + 6$ ). It also beeps in the up links if it is a border node (and relays any beep heard through a down link). Finally, when a follower node hears a beep in its up links, it terminates. Consequently, before an overlay network covers the whole network, the root receives beeps in every (termination detection) phase.

► **Lemma 5.** *Let  $r$  be the root of an overlay network. Then from diffusion phase  $|\alpha(id_r)| + 6$  onwards, node  $r$  hears beeps in its down links every phase, until it becomes a border node itself, or until its overlay covers the whole network.*

**Proof.** Let  $r$  be the root of an overlay network. From Lem. 4,  $r$ 's overlay network is properly constructed, therefore the virtual links can be used. We define a (overlay) *downwards path* from node  $v$  to node  $u$ , as a sequence of down links starting in  $v$  and ending in  $u$ . A node  $u$  is *downwards reachable* from node  $v$  if there is an overlay downwards path from  $v$  to  $u$ .

Consider a follower node  $v$ , having just joined  $r$ 's overlay (in phase  $p$ ). Node  $v$  beeps in its up links for 7 termination detection phases after it joins (from phase  $p$  to phase  $p + 6$ ). For each additional level in the overlay with nodes downwards reachable from  $v$ ,  $v$  beeps in its up links during 7 additional termination detection phases (by relaying beeps heard in its down links, to its up links). Although the next layer (node  $u$ ) is one further hop away from the root, and starts beeping in phase  $p + 6$ ,  $v$  beeps during phase  $p + 6$  (7<sup>th</sup> termination detection phase after it joins) and relays  $u$ 's first beep in phase  $p + 7$ . Consequently, there is no interruption in beeps sent through the up links. If an overlay node becomes a border node (some of its neighbors do not join in phase  $p + 6$ ), then it beeps in up links in all phases  $p' > p + 6$ . If it exits the overlay, then its neighbors closer to the root become border nodes and beep in their up links. Therefore, the root keeps hearing beeps in its down links while levels are added to its overlay, but also if one of its overlay nodes becomes a border node. In that latter case, the root does not have the maximum id, and hears beeps in its down links until it becomes a border node itself. ◀

► **Theorem 6.** *Explicit Leader Election is solved (uniformly) in  $O(D + \log n)$  rounds in the beeping model.*

**Proof.** The maximum identifier node  $v$  starts to construct its overlay network in phase  $|\alpha(id_v)| + 6$ , which is  $O(\log n)$ . For any given node  $u \neq v$ ,  $prefix_v$  never modifies its bits to match  $prefix_u$ . Consequently,  $v$  never joins  $u$ 's overlay and  $v$ 's overlay is the only one to grow until it covers the whole network, at a rate of adding a level every 6 diffusion phases. Thus,  $v$ 's overlay covers the whole network after an additional  $O(D)$  diffusion phases. Node  $v$  hears beeps in its down links for another additional  $O(D)$  phases, since beeps from the

last nodes to join the overlay take  $O(D)$  rounds to reach  $v$ . After that, node  $v$  no longer hears beeps in its down links (Lemma 5) and is the only node in the network to terminate as leader. Then, it beeps in its down links, so that all nodes can terminate. ◀

### 3.3 Discussion and Perspectives

The deterministic LE algorithm presented in this section works without any change with an arbitrary (unbounded) id space  $\{1, \dots, U\}$ . In this case, its time complexity is  $O(D + \log U)$ . For an unbounded id space, a known result from distributed bit complexity [11] gives a lower bound of  $\Omega(\log U)$  for a network with two nodes. This implies a lower bound of  $\Omega(D + \log U)$  for multi-hop networks. Consequently, the presented algorithm is asymptotically time-optimal even with an unbounded id space.

Furthermore, the algorithm can be modified to work if it starts with only a subset of nodes as candidates, or if the ids are not unique (as long as the highest id-encoding is still unique). Since a set of (non-unique) ids with a unique maximum can be generated without knowing  $n$  or  $N$  [17], this last variant can then be applied to obtain a randomized uniform (in both  $n$  and  $D$ ) leader election algorithm.

## 4 Additional Results

LE is an important and often-used primitive when designing distributed algorithms. Thus, it makes sense that improving the time complexity of LE results in improved time complexities for other tasks. We propose improved algorithms for leader election in anonymous networks, MIS and coloring (in trees), and multi-broadcast.

### 4.1 Randomized Leader Election

When dealing with communication-restrictive models such as  $\mathcal{BEE}\mathcal{P}$ , anonymity is especially important from an application viewpoint. Indeed, when considering large scale dynamic wireless networks, it might not be economically feasible to equip all nodes with unique identifiers. Additionally, nodes might be prevented from revealing their unique ids (explicitly or through their actions), due to privacy or security concerns [23]. For this case, a deterministic algorithm assuming unique identifiers can be adapted into a randomized one (w.h.p. time and safety guarantees) for anonymous networks, as stated in [13]. Indeed, one can generate a unique id w.h.p. by independently sampling  $\theta(\log n)$  bits. But in return the knowledge of the network size  $n$  or at least some polynomial upper bound  $N = O(n^c)$ , is required.

### 4.2 MIS and 5-coloring for Trees

Symmetry breaking procedures such as maximal independent set (MIS) and coloring are important building blocks, especially in the communication-restrictive beeping model. Specifically, the MIS problem consists of choosing a set of nodes (*local leaders*) so that there are no two neighbors in the set (*independence*), and such that no other node of the network can be added to the set without causing the loss of the independence property. On the other hand, the  $c$ -coloring problem consists of assigning colors in  $\{1, \dots, c\}$  to the nodes of the network, such that neighboring nodes have different colors.

It is well-known that given a leader in tree networks (elected using  $O(D + \log n)$  rounds), it is simple to 2-color the tree in  $O(D)$  supplementary rounds. However, MIS and coloring have an  $\Omega(\log n)$  lower bound (even in tree networks, as the bound from [21] holds for a graph

of disconnected pair of nodes), so this  $O(D + \log n)$  2-coloring algorithm is non optimal for most communication graphs. Still, using the proposed uniform leader election algorithm, we design uniform, asymptotically time-optimal  $O(\log n)$  MIS and 5-coloring algorithms in  $\mathcal{BEEP}$ , for tree networks.

We first give the algorithmic description of the 5-coloring algorithm. Roughly, low degree nodes are colored first using 3 colors, and the remaining nodes form a subgraph where the connected components have at most a logarithmic diameter. Using the LE algorithm, these connected components can be 2-colored in a logarithmic number of rounds. Now, we give more details as to how these steps are achieved. First, the *LimitedDegreeColoring* algorithm from [3] is used to 3-color all nodes  $v$  with  $\deg(v) \leq 2$ , in  $O(\log n)$  rounds. Then, since all remaining nodes have a degree of at least 3, every remaining (non-colored) connected component (a tree) has a diameter of at most  $\log n$ . Thus, electing a leader for each such connected component requires  $O(\log n)$  rounds. It is well-known that, in trees, coloring nodes according to their distance to the root can be done using 2 colors. This distance can be learnt by all nodes in  $O(\log n)$  rounds. Specifically, nodes are synchronized after the leader election, and the leader broadcasts a beep, using a beep wave [13, 10] or reusing the overlay network from the leader election. The phase in which a node receives the broadcasted beep indicates its distance to the leader. Therefore the remaining non-colored nodes can be colored with another 2 colors, resulting in a 5-coloring for the communication graph.

From this 5-coloring, it is simple to compute an MIS in 5 additional rounds. Nodes with the same color form an independent set. Adding iteratively (at each round) nodes from each such set to a common independent set results in an MIS. Consequently, an MIS on the communication graph can also be computed in  $O(\log n)$  rounds.

Notice that since all parts of the uniform 5-coloring algorithm are themselves uniform, it is a bit tricky to force nodes to resynchronize during the sequential execution. For this purpose, we use the EBET technique [3], to provide synchronization points in a uniform fashion - that is possible because, for every component of the proposed algorithm, the terminal state at a node can be detected locally - and thus solve the issue.

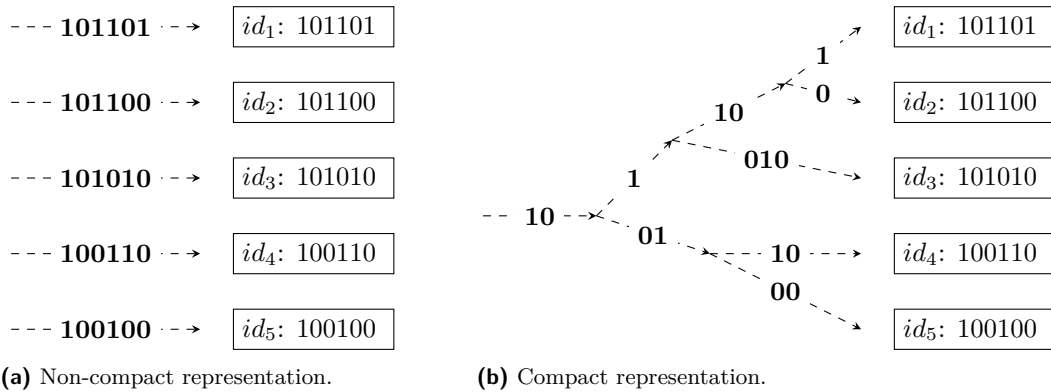
### 4.3 Multi-Broadcast with Provenance

Efficient communication primitives are fundamental building blocks in distributed computing, both for obtaining efficient algorithms and providing convenient abstractions of the actual communication mechanisms. These primitives are of even greater importance in  $\mathcal{BEEP}$ . When compared to other message-passing models, it is far more difficult to communicate messages throughout the network with beeps.

Now, consider the multi-broadcast problem. Multiple sources ( $k$  sources) have each a message they have to broadcast to all other nodes in the network. All messages are in  $\{1, \dots, M\}$ . In multi-broadcast with provenance, the  $k$  sources need to communicate their message, associated with their id, to all nodes in the network. In [10], an  $O(D \cdot \log n + k \log \frac{nM}{k})$  round algorithm is given and the authors conjecture that the  $D \cdot \log n$  term might be a lower bound. By presenting an  $O(D + \log n)$  deterministic LE algorithm, this work shows that leader election is not a bottleneck for the multi-broadcast problem (whereas the previous deterministic LE algorithm required  $O(D \cdot \log n)$  rounds). This suggests that  $D \cdot \log n$  might be reducible to  $D$  in both the deterministic and randomized cases.

The multi-broadcast with provenance algorithm in [10] can be divided into three core components: leader election, communicating the ids of all  $k$  sources and finally using the order of these ids to communicate all messages properly to the leader (which then broadcasts the information to the network). In [10], the second component relies on the leader and

## 20:16 Beeping a Deterministic Time-optimal Leader Election



■ **Figure 1** Difference between non-compact and compact representations of  $k$  different values (ids), indicated by the number of bits used as labels.

performs  $k$  simultaneous binary searches, in  $O(D \cdot \log n + k \log \frac{n}{k})$  rounds. Our contribution for this problem lies in improving the time complexities of the first and second components. The previous section (Sect. 3) improves the first component ([10] uses the leader election from [12]). More precisely, we use the leader election variant mentioned in Sect. 3.3, where the candidates can be a subset of all nodes. Here, only sources are candidates and the elected leader is the source with the maximum id. As for the second component, it is improved (whenever the number of sources  $k$  is sublogarithmic) by executing  $k - 1$  consecutive (variant) leader elections, where each leader election selects the source with the maximum id amongst the remaining non-elected sources. Notice that if nodes use their complete id for all  $k - 1$  consecutive leader elections, the time complexity is  $O(k \cdot (D + \log n))$  rounds. By being more efficient and leveraging the information communicated through the previous leader elections,  $k - 1$  consecutive leader elections are executed in  $O(k \cdot D + k \log \frac{n}{k})$  rounds only.

This result hinges on a compact manner of representing  $k$  unique values, which compresses the  $k \log n$  bits required to communicate  $k$  identifiers consecutively, into  $k \log \frac{n}{k}$  bits. As shown in Figure 1, after communicating  $id_1$  (6 bits), communicating  $id_2$  only takes one bit, and after that communicating  $id_3$  takes an additional 3 bits. Thus, with this compact representation, after the first leader is elected (amongst sources), subsequent leader elections are more efficient as candidates for subsequent leader elections (non-elected sources) are not required to communicate their whole id. For this reason, we introduce the  $\beta$ -encoding:  $\beta(i) = 0^{|bin|} \parallel 1 \parallel bin$  for an integer  $i$  and its binary representation  $bin$ . Contrary to  $\alpha$ -encodings, the highest  $\beta$ -encoding is produced by integers with the shortest but highest binary representations.

Assume that all candidates for leader election (sources which have not yet been elected) are given an identifier  $greaterID$ , greater than their own. They compute a reduced identifier  $reducedID$ , consisting of all bits from the first difference with  $greaterID$  onwards. Communicating  $reducedID$  to other nodes is, in this setting, the same as communicating  $id$  since these other nodes have knowledge of  $greaterID$  and thus deduce  $id$  from  $reducedID$ . Now, if candidates use the proposed deterministic LE algorithm with  $\beta(reducedID)$ , then the algorithm elects the node with the next maximum  $id$  value. Using this, the ids of all  $k$  sources are communicated to all nodes in  $O(k \cdot D + k \log \frac{n}{k})$  rounds.

Thus, executing both  $k - 1$  consecutive leader elections and  $k$  binary searches in parallel, the  $k$  ids (of the sources) are communicated to all nodes in  $O(\min\{k, \log n\} \cdot D + k \log \frac{n}{k})$  rounds. Then, the messages are gathered and broadcast using the leader, in a further  $O(D + k \log M)$  rounds, resulting in a  $O(\min\{k, \log n\} \cdot D + k \log \frac{nM}{k})$  algorithm for multi-broadcast with provenance.

## References

- 1 Y. Afek, N. Alon, Z. Bar-Joseph, A. Cornejo, B. Haeupler, and F. Kuhn. Beeping a maximal independent set. *Distributed Computing*, 26(4):195–208, Aug 2013.
- 2 D. Alistarh, A. Cornejo, M. Ghaffari, and N. Lynch. Firefly synchronization with asynchronous wake-up. In *Workshop on Biological Distributed Algorithms*, 2014.
- 3 J. Beauquier, J. Burman, F. Dufoulon, and S. Kutten. Fast Beeping Protocols for Deterministic MIS and  $(\Delta+1)$ -Coloring in Sparse Graphs. In *IEEE INFOCOM*, 2018, to appear.
- 4 A. Casteigts, Y. Métivier, J. M. Robson, and A. Zemmari. Design Patterns in Beeping Algorithms. In *OPODIS*, pages 15:1–15:16, 2016.
- 5 A. Casteigts, Y. Métivier, J.M. Robson, and A. Zemmari. Deterministic leader election in  $\mathcal{O}(D + \log n)$  time with messages of size  $\mathcal{O}(1)$ . In *DISC*, pages 16–28, 2016.
- 6 I. Chlamtac and S. Kutten. On broadcasting in radio networks - problem analysis and protocol design. *IEEE Transactions on Communications*, 33(12):1240–1246, 1985.
- 7 A. Cornejo and F. Kuhn. Deploying wireless networks with beeps. In *DISC*, pages 148–162, 2010.
- 8 A. Czumaj and P. Davies. Optimal leader election in multi-hop radio networks. *ArXiv e-prints*, 2015. [arXiv:1505.06149](https://arxiv.org/abs/1505.06149).
- 9 A. Czumaj and P. Davies. Brief announcement: Optimal leader election in multi-hop radio networks. In *PODC*, pages 47–49, 2016.
- 10 A. Czumaj and P. Davies. Communicating with Beeps. In *OPODIS*, pages 1–16, 2016.
- 11 Y. Dinitz and N. Solomon. Two absolute bounds for distributed bit complexity. In *Structural Information and Communication Complexity*, pages 115–126, 2005.
- 12 K.-T. Förster, J. Seidel, and R. Wattenhofer. Deterministic leader election in multi-hop beeping networks. In *DISC*, pages 212–226, 2014.
- 13 M. Ghaffari and B. Haeupler. Near optimal leader election in multi-hop radio networks. In *SODA*, pages 748–766, 2013.
- 14 S. Gilbert and C. Newport. The computational power of beeps. In *DISC*, pages 31–46, 2015.
- 15 R. Guerraoui and A. Maurer. Byzantine fireflies. In *DISC*, pages 47–59, 2015.
- 16 S. Kutten, G. Pandurangan, D. Peleg, P. Robinson, and A. Trehan. On the complexity of universal leader election. In *PODC*, pages 100–109, 2013.
- 17 Y. Métivier, J.M. Robson, and A. Zemmari. Analysis of fully distributed splitting and naming probabilistic procedures and applications. *Theoretical Computer Science*, 584:115–130, 2015. Special Issue on Structural Information and Communication Complexity.
- 18 K. Nakano and S. Olariu. Randomized  $o(\log \log n)$ -round leader election protocols in packet radio networks. In *Algorithms and Computation*, pages 210–219, 1998.
- 19 S. Navlakha and Z. Bar-Joseph. Distributed information processing in biological and computational systems. *Commun. ACM*, 58(1):94–102, 2014.
- 20 D. Peleg. Time-efficient broadcasting in radio networks: A review. In *Distributed Computing and Internet Technology*, pages 1–18, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.
- 21 J. Schneider and R. Wattenhofer. What is the use of collision detection (in wireless networks)? In *DISC*, pages 133–147, 2010.
- 22 A. Scott, P. Jeavons, and L. Xu. Feedback from nature: An optimal distributed algorithm for maximal independent set selection. In *PODC*, pages 147–156, 2013.
- 23 J. Seidel. *Anonymous distributed computing: computability, randomization and checkability*. PhD thesis, ETH Zurich, Zürich, Switzerland, 2015. URL: <http://d-nb.info/1080812695>.





# An Almost Tight RMR Lower Bound for Abortable Test-And-Set

Aryaz Eghbali

Department of Computer Science, University of Calgary, Canada  
aryaz.eghbali@ucalgary.ca

Philipp Woelfel

Department of Computer Science, University of Calgary, Canada  
woelfel@ucalgary.ca

---

## Abstract

---

We prove a lower bound of  $\Omega(\log n / \log \log n)$  for the remote memory reference (RMR) complexity of abortable test-and-set (leader election) in the cache-coherent (CC) and the distributed shared memory (DSM) model. This separates the complexities of abortable and non-abortable test-and-set, as the latter has constant RMR complexity [27].

Golab, Hendler, Hadzilacos and Woelfel [29] showed that compare-and-swap can be implemented from registers and test-and-set objects with constant RMR complexity. We observe that a small modification to that implementation is abortable, provided that the used test-and-set objects are atomic (or abortable). As a consequence, using existing efficient randomized wait-free implementations of test-and-set [23], we obtain randomized abortable compare-and-swap objects with almost constant ( $O(\log^* n)$ ) RMR complexity.

**2012 ACM Subject Classification** Theory of computation → Shared memory algorithms

**Keywords and phrases** Abortability, Test-And-Set, Leader Election, Compare-and-Swap, RMR Complexity, Lower Bound

**Digital Object Identifier** 10.4230/LIPIcs.DISC.2018.21

**Related Version** A full version of the paper is available at [21], <https://arxiv.org/abs/1805.04840>.

**Funding** This research was undertaken, in part, thanks to funding from the Canada Research Chairs program and from the Discovery Grants program of the Natural Sciences and Engineering Research Council of Canada (NSERC).

**Acknowledgements** We thank the anonymous reviewers for their careful reading and detailed feedback.

## 1 Introduction

In this paper, we study the remote memory references (RMR) complexity of abortable test-and-set. Test-and-set (TAS) is a fundamental shared memory primitive that has been widely used as a building block for classical problems such as mutual exclusion and renaming, and for the construction of stronger synchronization primitives [37, 41, 20, 15, 8, 7, 6, 29].

We consider a standard asynchronous shared memory system in which  $n$  processes with unique IDs communicate by reading and writing shared registers. A TAS object stores a bit that is initially 0, and provides two methods, `TAS()`, which sets the bit and returns its previous value, and `read()`, which returns the current value of the bit. TAS is closely



© Aryaz Eghbali and Philipp Woelfel;

licensed under Creative Commons License CC-BY

32nd International Symposium on Distributed Computing (DISC 2018).

Editors: Ulrich Schmid and Josef Widder; Article No. 21; pp. 21:1–21:19

Leibniz International Proceedings in Informatics



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

related to mutual exclusion [18]: a TAS object can be viewed as a one-time mutual exclusion algorithm, where only one process (the one whose `TAS()` returned 0) can enter the critical section [19].

TAS objects have consensus-number two, and therefore they have no wait-free implementations from atomic registers. In particular, in deterministic TAS implementations, processes may have to wait indefinitely, by spinning (repeatedly reading) variables. It is common to predict the performance of such blocking algorithms by bounding remote memory references (RMRs). These are memory accesses that traverse the processor-to-memory interconnect. Local-spin algorithms achieve low RMR complexity by spinning on locally accessible variables. Two models are common: In *distributed shared memory (DSM)* systems, each shared variable is permanently locally accessible to a single processor and remote to all other processors. In *cache-coherent (CC)* systems, each processor keeps local copies of shared variables in its cache; the consistency of copies in different caches is maintained by a *coherence protocol*. Memory accesses that cannot be resolved locally and have to traverse the processor-to-memory interconnect are called *remote memory references (RMRs)*.

Golab, Hendler, and Woelfel [27] implemented deadlock-free TAS objects from registers with  $O(1)$  RMR complexity for the DSM and the CC model, which in turn have been used to construct equally efficient comparison-primitives, such as compare-and-swap (CAS) objects [29]. These constructions are particularly useful in the study of the complexity of the mutual exclusion problem, for which the RMR complexity is the standard performance metric [10, 9, 36, 13, 16, 33, 34, 35, 14, 31, 32, 42, 24, 11, 38, 17, 39, 25].

In the context of mutual exclusion, it has been observed that systems often require locks to support a “timeout” capability that allows a process waiting too long for the lock, to abort its attempt [43]. In database systems, such as Oracle’s Parallel Server and IBM’s DB2, the ability of a thread to abort lock attempts serves the dual purpose of recovering from a transaction deadlock and tolerating preemption of the thread that holds the lock [43]. In real time systems, the abort capability can be used to avoid overshooting a deadline. Solutions to this problem have been proposed in the form of *abortable* mutual exclusion algorithms [43, 33, 42, 39, 17, 26]. In such an algorithm, at any point in the entry section a process may receive an *abort signal* upon which, within a finite number of its own steps, it must either enter the critical section or abort its current attempt to do so, by returning to the remainder section.

The complexity of the mutual exclusion problem in systems providing only registers is not affected by abortability: The abortable algorithms by Danek and Lee [17] and Lee [40] use only atomic registers and achieve  $O(\log n)$  RMR complexity, which asymptotically matches the known lower bound for non-abortable mutual exclusion [13]. But abortable mutual exclusion algorithms seem to be much more difficult to obtain than non-abortable ones, and it is not surprising that all such algorithms preceding [17, 40] used stronger synchronization primitives (e.g., LL/SC objects in [33]). Moreover, no RMR efficient randomized abortable mutual exclusion algorithms are known, unless stronger primitives are used [42, 26]; on the other hand, several non-abortable randomized implementations use only registers [30, 31, 25, 14].

As mentioned earlier, CAS objects with  $O(1)$  RMR complexity can be obtained from registers [29], but they cannot be used in an abortable mutual exclusion algorithm without sacrificing its abortability: if a process receives the abort signal while being blocked in an operation on a CAS object, it has no option to finish that operation in a wait-free manner, and thus may not be able to abort its attempt to enter the critical section. In general, implemented blocking strong objects, cannot be used to obtain abortable mutual exclusion objects.

One way of dealing with this impasse can be to make implementations of strong primitives also abortable, and to devise mutual exclusion algorithms in such a way that they accommodate operation aborts. Similarly, other algorithms and data structures that may require timeout capabilities, can potentially be implemented from abortable objects, but not from non-abortable ones.

We define abortability in the following, natural way: In a concurrent execution, a process executing an operation on the object may receive an abort signal at any point in time. When that happens, the process must finish its method call within a finite number of its own steps, and as a result the method call may fail to take effect, or it may succeed. The resulting execution must satisfy the safety conditions of the object (e.g., linearizability), if all failed operations are removed. Moreover, a process must be able to find out, by looking at the return value, whether its aborted operation succeeded, and if it did, then the return value must be consistent with a successful operation.

It may be tempting to define a weaker forms of abortability, e.g., where a return value of an aborted operation does not indicate whether the operation succeeded or not. As discussed in Section 1.1, such a weaker notion of abortability has indeed been suggested [4], but in a different context, where a process can choose for itself to abort its own pending operation (e.g., if it detects contention). In our scenario, where aborts are determined in an adversarial manner, the usefulness of such a weaker notion is not clear. For example, abortable TAS objects (according to our definition) can easily be used to implement an abortable mutual exclusion algorithm (TAS-lock): One can store a pointer to a “current” TAS object in a single register  $R$ . To get the lock, a process calls  $\text{TAS}()$  on the TAS object that  $R$  points to, and if the return value is 0, then the process has the lock, and otherwise it keeps reading  $R$  until its value changes. To release the lock, the process simply swings the pointer  $R$  so that it points to a new, fresh TAS object (this technique was proposed in [5], and [1, 2] showed how to bound the number of involved TAS objects). This also works in the case of aborts, because a process knows whether its operation took effect, and thus whether it is allowed to swing the pointer (and in fact must, to avoid dead-locks).

For the weaker definition of abortability mentioned above, a process whose  $\text{TAS}()$  aborted may not be able to find out whether it has the lock or not, and then it can also not swing the pointer to a new TAS object, even though its  $\text{TAS}()$  may have set the bit from 0 to 1. In fact, suppose that two processes call  $\text{TAS}()$ , and both  $\text{TAS}()$  calls abort without receiving the information whether the aborted operation took effect. Then the TAS bit may be set, but none of the processes has received any information regarding who was successful, and reading the TAS object also provides no information.

Even though our notion of abortability may seem strong, any abortable mutual exclusion algorithm can be used to obtain any abortable object from its corresponding sequential implementation, by simply protecting the sequential code in the critical section. An interesting question is therefore, whether abortable objects can be obtained at a lower RMR cost than mutual exclusion.

We observe that this is true for implementations of abortable CAS objects from abortable TAS objects on the CC model: a straightforward modification of the constant RMR implementation of non-abortable CAS from TAS objects and registers [29], immediately yields an abortable CAS object, provided that the used TAS objects are atomic or also abortable.

► **Observation 1.** *There is a deadlock-free implementation of abortable CAS from atomic registers and deadlock-free abortable TAS objects, which has  $T + O(1)$  RMR complexity on the CC model, provided that  $\text{TAS}()$  operations have RMR complexity  $T$ .*

## 21:4 RMR Lower Bound for Abortable TAS

This theorem immediately implies that we could use atomic TAS objects (which are trivially abortable) to obtain abortable CAS objects with constant RMR complexity. But obviously, it does not help constructing deterministic abortable CAS objects from registers. However, we can use the fact that there are known randomized constructions of TAS objects, which are not only RMR efficient, but even efficient with respect to step complexity. More precisely, Giakkoupis and Woelfel [23] presented a randomized TAS implementation from registers, where the maximum number of steps any process takes in a  $\text{TAS}()$  operation has expectation  $O(\log^* n)$  against an oblivious adversary. (This means, the order in which processes take steps must be completely independent of their random decisions.) This construction is also randomized wait-free, meaning that, for any schedule all  $\text{TAS}()$  calls terminate with probability 1. Therefore,  $\text{TAS}()$  calls are abortable (in a randomized sense), as for any schedule each method call terminates with probability 1 (whether the process receives an abort signal or not). In the construction of CAS above, we can therefore use such randomized TAS implementation in place of abortable TAS.

► **Corollary 2.** *There is a deadlock-free randomized implementation of abortable CAS from atomic registers, such that on the CC model against an oblivious adversary each abort is randomized wait-free, and each operation on the object incurs at most  $O(\log^* n)$  RMRs.*

Recall that there is also a deterministic constant RMR implementation of TAS from registers [27]. Hence, making this implementation abortable and applying Observation 1 would immediately yield deterministic constant RMR abortable implementations of CAS from registers. Unfortunately, it turns out that a deterministic constant RMR implementation of abortable TAS from registers does not exist. In particular, we define the abortable leader election (LE) problem, which is not harder than abortable TAS (with respect to RMR complexity). Our main technical result is an RMR lower bound of  $\Omega(\log n / \log \log n)$  for that object.

In a (non-abortable) LE protocol, every process decides for itself whether it becomes the leader (it returns *win*) or whether it loses (it returns *lose*). At most one process can become the leader, and not all participating processes can lose. I.e., if all participating processes finish the protocol, then exactly one of them returns *win* and all others return *lose*. Note that then in an abortable LE protocol all participating processes are allowed to return *lose*, provided that all of them received the abort signal.

An abortable TAS object immediately yields an abortable LE protocol: Each process executes a single  $\text{TAS}()$  operation and returns *win* if the  $\text{TAS}()$  call returns 0, and otherwise *lose* (i.e., it returns *lose* also when the  $\text{TAS}()$  return value indicates a failed abort). Similarly, it is easy to implement abortable TAS from abortable LE and a single register, preserving the asymptotic RMR complexity (but care must be taken to obtain linearizability).

Our main result is the following:

► **Theorem 3.** *For both, the DSM and the CC model, any deadlock-free abortable leader election (and thus any abortable TAS) implemented from registers has an execution in which at least one process incurs  $\Omega(\log n / \log \log n)$  RMRs.*

This lower bound is asymptotically tight up to a  $\log \log n$  factor, because one can trivially obtain a TAS object with  $O(\log n)$  RMR complexity by protecting a straight forward sequential  $\text{TAS}()$  implementation with the abortable mutual exclusions algorithms by Danek and Lee [17] and Lee [40].

Leader election is one of the seemingly simplest synchronization primitives that have no wait-free implementation. In particular, as argued above, the lower bound in Theorem 3 immediately also applies to abortable TAS. This is in stark contrast to the  $O(1)$  RMRs

upper bound for non-abortable TAS and even CAS implementations [27, 29]. It shows that adding abortability to synchronization primitives is almost as difficult as solving abortable mutual exclusion, which has an RMR complexity of  $\Theta(\log n)$  [17, 40].

In our lower bound proof we identify the crucial reason for why abortable LE is harder than its non-abortable variant: According to standard bi-valency arguments, for any deadlock-free LE algorithm, there is an execution in which some process takes an infinite number of steps. But it is not hard to see that one can design an (asymmetric) 2-process LE protocol in which one fixed process is wait-free, because the other one waits for the first one to make a decision if it detects contention. It turns out that this is not the case for abortable LE: Here, for any process, there is an execution in which that process takes an infinite number of steps.

## 1.1 Other Related Work.

Attiya, Guerraoui, Hendler, and Kuznetsov [12] consider augmenting non-wait-free implementations with a mechanism that allows processes to abort an ongoing operation and returning a special “failed” return value. Contrary to our model, where aborts are chosen in an adversarial manner, in the work of Attiy et. al. processes can decide when to abort in order to achieve termination (e.g., when they detect contention). This makes implemented objects weaker, while our abortable objects are stronger than non-abortable ones. A similar notion of abortable objects was suggested by Aguilera, Frølund, Hadzilacos, Horn, and Toueg [4]. In their work, processes can also decide to abort an ongoing operation, but the caller of an aborted operation may not find out whether its operation took effect or not. Since this uncertainty may not be acceptable, they also introduce query-abortable objects, where a query operation allows a process to determine additional information about its last non-query operation.

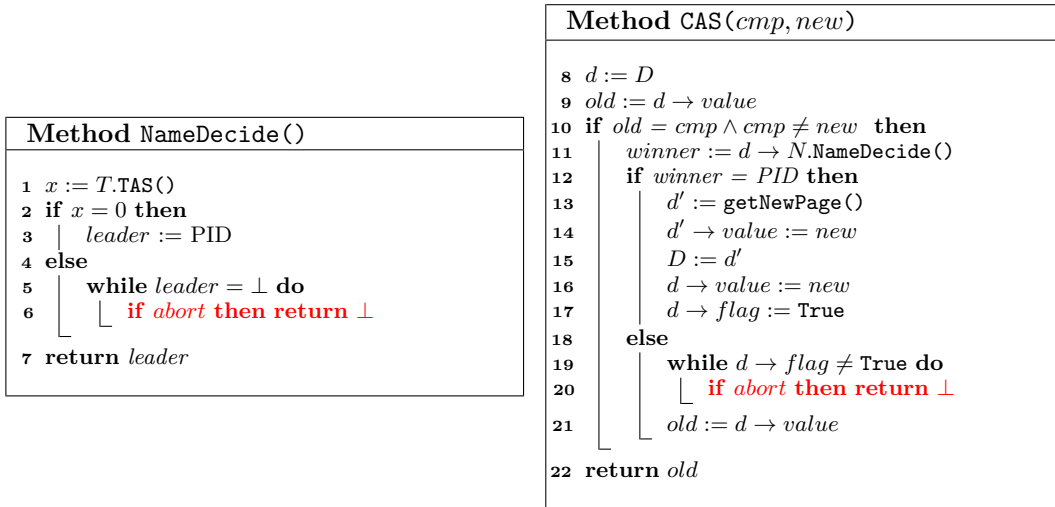
Note that their notion of abortability is quite different from the one used commonly for mutual exclusion and adopted by us, where the system, and not the implementation, dictates when a process needs to abort.

## 2 Abortable Compare-And-Swap in the CC Model

In this section we consider the cache-coherent (CC) model. Each process obtains a cached copy with each read of a register, and the cached copy only gets invalidated if some process later writes to the same register. Writes as well as reads of non-cached registers incur RMRs, while reads of cached registers do not.

A CAS object provides two operations,  $\text{CAS}(cmp, new)$ , and  $\text{read}()$ . Operation  $\text{read}()$  returns the current value of the object. Operation  $\text{CAS}(cmp, new)$  writes  $new$  to the object, if the current value is  $cmp$ , and otherwise does not change the value of the object. In either case it returns the old value of the object.

Golab et. al. [28] gave an implementation of CAS from TAS objects and registers, which has constant RMR complexity in the CC model, i.e., each  $\text{CAS}()$  and each  $\text{read}()$  operation incurs only  $O(1)$  RMRs. In this section we show how to make that implementation abortable, provided that the used TAS objects are either atomic or abortable and deadlock-free. The pseudocode is in Figure 1. The original (non-abortable) version of the code is shown in black and our additional code to make it abortable in red (lines 6 and 20). It is assumed that the abort-signal is sent to a process by means of setting the process’ flag *abort*.



■ **Figure 1** Implementation of (abortable) `NameDecide()` and `CAS()`. Without lines 6 and 20 the algorithms are equivalent to the non-abortable implementations in [28].

## 2.1 From TAS to Name Consensus

The implementation in [28] first constructs a name consensus object from a single TAS object  $T$ . This object supports a method `NameDecide()`, which each process is allowed to call at most once. All `NameDecide()` calls return the same value (agreement), which is the ID of a process calling `NameDecide()` (validity).

The non-abortable implementation in [28] uses a TAS object  $T$  and a register  $leader$  that is initially  $\perp$ . In a `NameDecide()` call, a process  $p$  first calls  $T.TAS()$ . If the `TAS()` returns 0, then  $p$  wins, and writes  $p$  to  $leader$ . Otherwise,  $p$  loses, and so it repeatedly reads  $leader$ , until  $leader \neq \perp$ , upon which  $p$  can return the value of  $leader$ . It is easy to see (and was formally proved in [28]) that this is a correct name consensus algorithm.

We now show how this implementation can be made abortable, assuming the TAS object  $T$  is atomic or abortable. We assume that when a process receives the abort signal, a static process-local variable  $abort$ , which is initially false, changes to `True`.

Recall that abortability requires that the return value of a `TAS()` operation indicates whether it failed or succeeded. We assume a failed `TAS()` simply returns  $\perp$ . In `NameDecide()`, processes are only waiting until  $leader$  changes. If a process is receiving the abort signal while waiting for  $leader$  to change, then it can also simply return  $\perp$ . The rest of the algorithm is the same as the original name consensus algorithm.

Clearly, the new code (line 6) does not affect RMR complexity, and following an abort the code is wait-free. Moreover, correctness (validity and agreement) in case of no failed `NameDecide()` operations follow immediately from correctness of the original algorithm. If a `NameDecide()` operation fails (i.e., returns  $\perp$ ), then it did not change any shared memory object (its `TAS()` must have either failed, or returned 1). Hence, removing an aborted and failed `NameDecide()` operation from the execution does not affect any other processes, and therefore the resulting execution must be correct.

## 2.2 From Name Consensus to Compare-And-Swap

We now show how the abortable name consensus algorithm can be used to obtain abortable CAS. Consider the implementation of `CAS( $cmp, new$ )` on the right hand side in Figure 1. The black code is logically identical to the one in [28]. It uses a register  $D$  that points to

a *page*, which stores two registers, *value* and *flag*, as well as a name consensus object  $N$ . Register *value* at the page pointed to by  $D$  stores the current value of the object. (Thus, a `read()` operation, for which we omit the pseudo code, simply returns  $D \rightarrow \text{value}$ .) The `CAS()` operation assumes a wait-free method `getNewPage()`, which returns an unused page from a pool of pages (for simplicity assume this pool has infinitely many pages, but there are methods for wait-free memory management that allow using a bounded pool [29, 3]).

For a description of how the algorithm `CAS(cmp, new)` works, we refer to [28]. We can prove that the abortable version presented here is correct, provided that the non-abortable version (with line 20 removed) is: First of all, obviously line 20 does not change the RMR complexity. Moreover, if a process receives the abort-signal, then its abortable `NameDecide()` call terminates within a finite number of steps, and the process also does not wait in the while-loop, so its `CAS()` call completes within a finite number of its steps. Finally, notice that a `CAS()` call returns  $\perp$  only if an abort signal was received, and in that case no shared memory objects are affected (the process cannot have won the `NameDecide()` call). Hence, all aborted and failed operations can be removed from the execution without changing anything for the remaining operations. As a result we obtain Observation 1.

### 3 RMR Lower Bound for Abortable Leader Election

In this section, we give an overview of the RMR lower bound proof for abortable leader election (and thus TAS) as stated in Theorem 3. Due to space constraints, the full proof is omitted, but it is made available in the technical report [21].

First, we define some notation, the system model, RMR complexity, and the abortable leader election problem.

#### 3.1 Lower Bound Preliminaries

**System Model and Notation.** For an set  $Q$  and any non-negative integer  $k$ , let  $Q^k$  denote the set of all sequences of length  $k$  that contain only the elements in  $Q$ . Furthermore,  $Q^*$  is the set of all sequences over  $Q$ .

For the lower bound we assume a set  $\mathcal{P}$  of  $n$  processes, and an arbitrary large but finite set  $\mathcal{R}$  of shared registers. In each shared memory step (corresponding to a state transition), a process either reads or writes a register in  $\mathcal{R}$ . At an arbitrary point, a process may also receive an *abort signal* which does not result in a shared memory access, but in a state change of that process, provided the process has not received the abort signal earlier. Once a process has reached a halting state, it remains in that state forever, and does not execute any further shared memory steps.

For each process  $p \in \mathcal{P}$ , we define a special *abort symbol*  $p^\top$ , which is used to indicate that a process receives an abort signal (as defined below). For a set  $P \subseteq \mathcal{P}$  let  $P^\top = \{p^\top \mid p \in P\}$ , and  $P^\Delta = P \cup P^\top$ . A *schedule* is a sequence  $\sigma$  over  $\mathcal{P}^\Delta$ . Thus, any schedule  $\sigma$  is in  $(\mathcal{P}^\Delta)^*$ . The length of a schedule  $\sigma$  is denoted by  $|\sigma|$ . Let  $Proc(\sigma)$  denote the set of processes  $p \in \mathcal{P}$  that occur in  $\sigma$  at least once, not counting symbols in  $\mathcal{P}^\top$ .

A *configuration* is a sequence that describes the state of each process in  $\mathcal{P}$  and each register in  $\mathcal{R}$ . A configuration  $C$  and a schedule  $\sigma \in \mathcal{P}^\Delta$  of length one result in a new configuration  $Conf(C, \sigma)$ , obtained from  $C$  by process  $p$  taking its next step, if  $\sigma = p \in \mathcal{P}$ , or by process  $p$  receiving the abort signal, if  $\sigma = p^\top \in \mathcal{P}^\top$ . If  $\sigma = \sigma_1 \sigma_2 \dots \sigma_k$  is a schedule of length  $k > 1$ , then the new configuration is determined inductively as  $Conf(Conf(C, \sigma_1); \sigma_2 \dots \sigma_k)$ . Configuration  $C$  and schedule  $\sigma = \sigma_1 \dots \sigma_k$  also define an *execution*  $Exec(C, \sigma)$ , which is a sequence  $s_1 s_2 \dots s_k$ , where  $s_i$  is the step executed or the abort signal received in the transition



from  $C_{i-1} = \text{Conf}(C, \sigma_1 \dots \sigma_{i-1})$  to  $C_i = \text{Conf}(C_{i-1}, \sigma_i)$ . To specify that an execution starting in  $C$  and running by schedule  $\sigma$  is running algorithm  $A$ , we use  $\text{Exec}_A(C, \sigma)$ . The length of an execution  $E$  is denoted by  $|E|$ . We call  $s_i$  an *abort step by process  $p$* , if in  $s_i$  process  $p$  receives the abort signal.

The initial configuration is denoted by  $\Gamma$ . A configuration  $C$  is *reachable*, if there exists a schedule  $\sigma$  such that  $\text{Conf}(\Gamma, \sigma) = C$ . Since only reachable configurations are important in our algorithms and proofs, we use *configuration* instead of *reachable configuration* from this point on. For a configuration  $C$  we let  $\sigma_{\rightarrow C}$  denote an arbitrary but unique schedule such that  $\text{Conf}(\Gamma, \sigma_{\rightarrow C}) = C$ , and we define  $E_{\rightarrow C} = \text{Exec}(\Gamma, \sigma_{\rightarrow C})$ .

The projection of a schedule  $\sigma$  to a set  $Q \subseteq \mathcal{P}^\Delta$  is denoted by  $\sigma|Q$ . For an execution  $E$  and a set  $P$  of processes,  $E|P$  denotes the sub-sequence of  $E$  that contains all (abort and shared memory) steps by processes in  $P$ . If  $Q$  or  $P$  contains only one symbol,  $s$ , then we write  $\sigma|s$  instead of  $\sigma|\{s\}$ , or  $E|s$  instead of  $E|\{s\}$ .

Recall that a configuration  $C$  determines the state of each process. I.e., for any two executions  $E$  and  $E'$  resulting in the same configuration  $C$ , each process is in the same state at the end of  $E$  as at the end of  $E'$ , and in particular  $E|p = E'|p$ . Therefore, we associate the state of a process in configuration  $C$  with  $E_{\rightarrow C}|p$ . (But note that if two executions  $E$  and  $E'$  are indistinguishable to each process in  $Q \subseteq \mathcal{P}$ , then this does not in general imply that  $E|Q = E'|Q$ .) The value of register  $r$  in configuration  $C$  is denoted by  $\text{val}_C(r)$ . Configurations  $C$  and  $D$  are *indistinguishable* to some process  $p$ , if  $E_{\rightarrow C}|p = E_{\rightarrow D}|p$  and  $\text{val}_C(r) = \text{val}_D(r)$  for every register  $r \in \mathcal{R}$ . For a set  $Q \subseteq \mathcal{P}$ , we write  $C \sim_Q D$  to denote that configurations  $C$  and  $D$  are indistinguishable to each process in  $Q$ ; for a set consisting of a single process  $p$  we write  $C \sim_p D$  instead of  $C \sim_{\{p\}} D$ .

Finally, for two sequences  $s_1$  and  $s_2$  let  $s_1 s_2$  denotes their concatenation. (We use this for schedules and executions.)

**RMR Complexity.** Our lower bound applies to both, the standard asynchronous distributed shared memory (DSM) model and cache-coherent (CC) model. In fact, we use a model that combines both, caches as well as locally accessible registers for each process.

We assume that the set of registers,  $\mathcal{R}$ , is partitioned into disjoint memory segments  $\mathcal{R}_p$ , for  $p \in \mathcal{P}$ . The registers in  $\mathcal{R}_p$  are *local* to process  $p$  and *remote* to each process  $q \neq p$ . We say that at the end of execution  $E$  a process  $p$  has a valid cached copy of register  $r$ , if in  $E$  process  $p$  reads or writes  $r$  at some point, and no other processes writes  $r$  after that. Note that the configuration obtained at the end of an execution starting in  $\Gamma$  uniquely determines whether  $p$  has a valid cached copy of a register  $r$ . The reason is that the state of  $p$  in configuration  $C$  determines the value that was written to or read from  $r$  when  $p$  accessed  $r$  last, and  $p$  has a valid cached copy of  $r$  if and only if  $\text{val}_C(r)$  equals that value. Let  $\text{Cache}_p(C)$  denote the union of  $\mathcal{R}_p$  and the set of registers of which process  $p$  has a valid cached copy in configuration  $C$  if  $p$  has not terminated in  $C$ , and the empty set if  $p$  is terminated in  $C$ .

A step in an execution  $E$  is either *local* or *remote* (we say it *incurs an RMR* if it is remote). All abort steps are local. A non-abort step by process  $p$  is local, if and only if it is either a read or a write of a register in  $\mathcal{R}_p$ , or it is a read of a register of which  $p$  has a local cached copy. (Our definition corresponds to a write-through cache; in a write-back cache, certain writes may also be local. Even though we believe that our lower bound proof technique can accommodate write-back caches, this is left for future work.)

For an execution  $E$  and a process  $p$ ,  $\text{RMR}_p(E)$  is the number of RMR steps by process  $p$  in execution  $E$ . For  $Q \subseteq \mathcal{P}$  we define  $\text{RMR}_Q(E) = \sum_{q \in Q} \text{RMR}_q(E)$ , which is equal to the total number of RMRs incurred by processes in  $Q$  in  $E$ . For the sake of conciseness, we use  $\text{RMR}(E)$  instead of  $\text{RMR}_{\mathcal{P}}(E)$ .



**Abortable Leader Election.** An algorithm solves *abortable leader election*, if for any schedule  $\sigma$ , in  $Exec(\Gamma, \sigma)$  each process that terminates returns *win* or *lose*, at most one process returns *win*, and if all processes in  $Proc(\sigma)$  return *lose*, then all processes in  $Proc(\sigma)$  receive the abort signal.

We usually assume without explicitly saying so that an abortable leader election satisfies *deadlock-freedom* and *wait-free abort*, defined as follows: Wait-free abort means that after a process received the abort signal it terminates within a finite number of its own steps. An infinite schedule  $\sigma$  is *P-fair* for  $P \subseteq \mathcal{P}$ , if each process in  $P$  appears infinitely many times in  $\sigma$ . An infinite execution  $E$  is *P-fair*, if there exists a configuration  $C$  and a *P-fair* schedule  $\sigma$  such that  $E = Exec(C, \sigma)$ . We use *fair* schedule and *fair* execution, instead of *P-fair*, when  $P = \mathcal{P}$ . A method is *deadlock-free* if for any schedule  $\sigma$  all process' method calls terminate in  $Exec(\Gamma, \sigma)$ , provided this execution is *P-fair*, where  $P$  is the set of processes calling the method.

### 3.2 Properties of Abortable Leader Election

In this section we derive the critical property that distinguishes non-abortable from abortable leader election for the purpose of the lower bound. We consider algorithms in which each process returns either *win* or *lose* upon termination. We call such algorithms *binary*. Note that any (abortable) leader election algorithm is a binary algorithm. (Recall that in abortable leader election aborted and failed operations return *lose*, and not  $\perp$  as in TAS.)

Several results in this section will concern only two arbitrarily selected processes in the  $n$ -process system for  $n \geq 2$ . For ease of notation, we will call these processes  $a$  and  $b$ .

For an execution  $E$  of a binary algorithm in which  $a$  returns  $x$  and  $b$  returns  $y$ , let  $(x, y)$  denote the *outcome vector* of  $E$ . For a binary algorithm  $A$  and a configuration  $C$ , let  $\mathcal{V}_A(C)$  denote the set of all outcome vectors of  $\{a, b\}$ -only executions starting in  $C$ , in which processes  $a$  and  $b$  terminate.

First we observe that the outcome vectors of two indistinguishable configurations are equal.

► **Observation 4.** For any binary algorithm  $A$ , if configurations  $C$  and  $D$  are indistinguishable to processes  $a$  and  $b$ , then  $\mathcal{V}_A(C) = \mathcal{V}_A(D)$ .

**Proof.** Since  $C$  and  $D$  are indistinguishable to processes  $a$  and  $b$ ,  $E_{\rightarrow C}|a = E_{\rightarrow D}|a$ ,  $E_{\rightarrow C}|b = E_{\rightarrow D}|b$ , and for any register  $r$ ,  $val_C(r) = val_D(r)$ . Thus, for any  $x$  in  $\{a, b\}^\Delta$ , we have  $(E_{\rightarrow C} \circ Exec(C, x))|a = (E_{\rightarrow D} \circ Exec(D, x))|a$ ,  $(E_{\rightarrow C} \circ Exec(C, x))|b = (E_{\rightarrow D} \circ Exec(D, x))|b$ , and for any register  $r$ ,  $val_{Conf(C, x)}(r) = val_{Conf(D, x)}(r)$ . So by induction, for any finite  $\{a, b\}$ -only schedule  $\sigma$ ,  $Conf(C, \sigma) \sim_{\{a, b\}} Conf(D, \sigma)$ . Therefore, if in  $Exec(C, \sigma)$  process  $p \in \{a, b\}$  terminates, it also terminates in  $Exec(D, \sigma)$  and it returns the same value in both executions. Hence, the outcome vector  $\mathcal{V}_A(C)$  is equal to  $\mathcal{V}_A(D)$ . ◀

For a binary algorithm  $A$ , configuration  $C$  is *bivalent* if  $\{(win, lose), (lose, win)\} = \mathcal{V}_A(C)$ . This definition of bivalency refers to two fixed but arbitrarily chosen processes,  $a$  and  $b$ . In a system with more than two processes, we may write  $\{a, b\}$ -bivalent to indicate the two processes  $a$  and  $b$  to which this definition applies. A configuration is *strongly bivalent* (or strongly  $\{a, b\}$ -bivalent) if it is bivalent and a solo-run by any process  $p \in \{a, b\}$ , starting in  $C$ , results in  $p$  winning. (Solo-run by  $p$  means an execution in which only process  $p$  takes steps, and none of them is an abort-step.)

A similar argument to the FLP Theorem [22] implies that for any deadlock-free binary algorithm and for any reachable bivalent configuration, there exists an infinite execution, where no process terminates.

► **Lemma 5.** *Let  $A$  be a deadlock-free binary algorithm and  $C$  an  $\{a, b\}$ -bivalent configuration. There exists an infinite schedule  $\sigma \in \{a, b\}^*$ , such that in  $\text{Exec}_A(C, \sigma)$  none of  $a$  and  $b$  terminate.*

To prove this lemma we first prove Claim 6 and use the fact that none of  $a$  and  $b$  can be terminated in an  $\{a, b\}$ -bivalent configuration.

► **Claim 6.** *In any deadlock-free binary algorithm  $A$ , if configuration  $C$  is  $\{a, b\}$ -bivalent, then either one of  $\text{Conf}(C, a)$  and  $\text{Conf}(C, b)$  is  $\{a, b\}$ -bivalent, or there exists an infinite  $\{a, b\}$ -only execution, where none of  $a$  and  $b$  terminates.*

**Proof.** Since configuration  $C$  is  $\{a, b\}$ -bivalent,  $\mathcal{V}_A(C) = \{(win, lose), (lose, win)\}$ . Suppose neither  $\text{Conf}(C, a)$  nor  $\text{Conf}(C, b)$  is  $\{a, b\}$ -bivalent. Then there exist distinct  $x, y \in \{win, lose\}$  such that

$$\begin{aligned} \mathcal{V}_A(\text{Conf}(C, a)) &= \{(x, y)\}, \text{ and} \\ \mathcal{V}_A(\text{Conf}(C, b)) &= \{(y, x)\} \end{aligned} \tag{1}$$

We now distinguish two cases.

**Case 1.** In  $C$ , processes  $a$  and  $b$  are poised to access different registers or poised to read the same register. Thus,

$$\text{Conf}(C, a \circ b) = \text{Conf}(C, b \circ a). \tag{2}$$

By (1),  $(y, x) \notin \mathcal{V}_A(\text{Conf}(C, a))$ . Since  $\mathcal{V}_A(\text{Conf}(C, a \circ b)) \subseteq \mathcal{V}_A(\text{Conf}(C, a))$ , it holds  $(y, x) \notin \mathcal{V}_A(\text{Conf}(C, a \circ b))$ . Thus, by (2),  $(y, x) \notin \mathcal{V}_A(\text{Conf}(C, b \circ a))$ . Since  $\mathcal{V}_A(\text{Conf}(C, b \circ a)) \subseteq \mathcal{V}_A(\text{Conf}(C, b)) = \{(y, x)\}$ , this means that  $\mathcal{V}_A(\text{Conf}(C, b \circ a)) = \emptyset$ . But this contradicts deadlock-freedom, as in a fair schedule starting in  $\text{Conf}(C, b \circ a)$  both processes must terminate and output something.

**Case 2.** In configuration  $C$ , both processes are poised to access the same register  $r$ , and at least one of them is poised to write  $r$ . Without loss of generality, assume that  $a$  is poised to write register  $r$ . If  $a$  takes its write step after  $b$ 's step, then  $a$ 's state and shared register values are no different than if only  $a$  takes its write step and  $b$  does not take its step. So  $\text{Conf}(C, a) \sim_a \text{Conf}(C, b \circ a)$ . If process  $a$  does not terminate in a solo-run starting in  $\text{Conf}(C, a)$ , then the claim is true, because there exists an infinite execution starting in  $C$  that neither  $a$  nor  $b$  terminates. However, if process  $a$  terminates in a solo-run starting in  $\text{Conf}(C, a)$ , by (1), we can conclude that  $(x, y) \in \mathcal{V}_A(\text{Conf}(C, b \circ a))$ . Since  $\mathcal{V}_A(\text{Conf}(C, b \circ a)) \subseteq \mathcal{V}_A(\text{Conf}(C, b))$ , it holds that  $(x, y) \in \mathcal{V}_A(\text{Conf}(C, b))$ . This contradicts  $\mathcal{V}_A(\text{Conf}(C, b)) = \{(y, x)\}$ . ◀

Any deadlock-free (non-abortable) 2-process leader election algorithm has a bivalent initial configuration. But in any fair schedule, both processes terminate. Therefore, the infinite execution that is guaranteed by the above corollary cannot be fair; in particular, it requires one of the two processes to run solo at some point. However, one can construct a deadlock-free (non-abortable) leader election algorithm in which one process never takes an infinite number of steps, no matter what the schedule is. The lemma below shows that this is not true for abortable two-process leader election algorithm.

► **Lemma 7.** *Let  $A$  be a deadlock-free abortable 2-process leader election algorithm with wait-free aborts. For any process  $p$ , there exists an execution starting in the initial configuration, in which  $p$  takes a infinitely many steps.*

**Proof.** Let  $\Gamma$  be the initial configuration of  $A$ . For the purpose of contradiction, assume there is a fixed process,  $a$ , that terminates within a finite number of its own steps in all executions. Let  $b$  be the other process.

By the semantics of abortable leader election, there is no execution in which both processes win, i.e.,

$$(win, win) \notin \mathcal{V}_A(\Gamma). \quad (3)$$

Let algorithm  $A'$  be the same as  $A$  except that during any execution,

- (1) if any of the two processes receive the abort signal, the abort signal is ignored; and
- (2) if in step  $s$  process  $b$  reads  $(a, x)$ , where  $x \neq \perp$ , then  $b$  continues its program, as if it had received the abort signal immediately after step  $s$ .<sup>1</sup>

In any execution of  $A$ , processes  $a$  and  $b$  can only both lose, if they both receive the abort signal. Since in  $A'$  both processes ignore the abort signals (and only  $b$  possibly simulates having received an abort signal), there is no execution of  $A'$  in which  $a$  and  $b$  both lose. Thus, for the initial configuration  $\Gamma'$  of  $A'$ ,

$$(lose, lose) \notin \mathcal{V}_{A'}(\Gamma'). \quad (4)$$

Consider any execution  $E' = Exec(\Gamma', \sigma')$  of algorithm  $A'$  starting in  $\Gamma'$ . We now create an execution  $E = Exec(\Gamma, \sigma)$  of  $A$  starting in  $\Gamma$ , by scheduling the processes in exactly the same order as in  $E'$ , but removing all abort signals. Moreover, when for the first time  $b$  reads a value of  $(a, x)$  in  $E$ , where  $x \neq \perp$  (if that happens), then we send process  $b$  the abort signal. By construction of  $A'$ , processes  $a$  and  $b$  execute exactly the same shared memory steps in execution  $E$  of algorithm  $A$  as in execution  $E'$  of algorithm  $A'$ . Thus, for every schedule  $\sigma'$  there is a schedule  $\sigma$  such that processes  $a$  and  $b$  execute in  $Exec_{A'}(\Gamma', \sigma')$  the same shared memory steps as in  $Exec_A(\Gamma, \sigma)$ . This implies

$$\mathcal{V}_{A'}(\Gamma') \subseteq \mathcal{V}_A(\Gamma). \quad (5)$$

Note that in the construction above, if  $\sigma'$  is fair, then so is  $\sigma$ . Hence, the fact that  $A$  is deadlock-free implies

$$A' \text{ is deadlock-free.} \quad (6)$$

In algorithm  $A$ , in a sufficiently long solo-run by  $a$  starting in  $\Gamma$ , in which  $a$  does not receive the abort-signal, process  $a$  terminates (by deadlock-freedom) and returns *win* (by the semantics of abortable leader election). Hence, in  $A'$  process  $a$  also terminates and returns *win* after a sufficiently long solo-run starting from  $\Gamma$ , because it takes exactly the same steps as in  $A$ . Since  $A'$  is deadlock-free by (6), process  $b$  terminates after a sufficiently long solo-run following  $a$ 's solo-run, and by (3) process  $b$  returns *lose*. With a symmetric argument, for algorithm  $A'$ , in a sufficiently long solo-run by  $b$  starting in  $\Gamma$ , followed by a sufficiently long solo-run of  $a$ , process  $b$  returns *win* and process  $a$  returns *lose*. Hence,  $\{(win, lose), (lose, win)\} \subseteq \mathcal{V}_{A'}(\Gamma')$ . Using (3) and (4) we conclude

$$\mathcal{V}_{A'}(\Gamma') = \{(win, lose), (lose, win)\}. \quad (7)$$

We will now show that  $A'$  is wait-free. This together with (7) contradicts Lemma 5, and thus proves the lemma.

<sup>1</sup> Recall that we assumed that each value that a process  $p$  writes is of the form  $(p, y)$ , where  $y \neq \perp$ .

Recall that in every execution of algorithm  $A$  process  $a$  terminates within a finite number of its own steps. As a result, the same is true for  $A'$ .

Hence, it suffices to show that  $b$  terminates within a finite number of its own steps. Suppose there is an execution  $E^*$  of  $A'$  in which  $b$  executes an infinite number of steps. Then  $b$  never reads a value of  $(a, x)$ , where  $x \neq \perp$ , as otherwise it would simulate having received the abort-signal in  $A$ , and then terminate after a finite number of steps. Since  $b$  never reads a value of  $(a, x)$ , where  $x \neq \perp$ , it cannot distinguish  $E^*$  from a solo-run starting in  $\Gamma'$ . Hence,  $b$  does not terminate in such an infinite solo-run. This contradicts (6).  $\blacktriangleleft$

One of the core properties of the abortable leader election problem that allows us to prove the lower bound is that there are no reachable strongly bivalent configurations in any execution.

► **Lemma 8.** *Let  $A$  be an abortable  $n$ -process leader election algorithm with wait-free aborts for  $n \geq 2$ . Further, let  $C$  be a reachable configuration and  $a, b$  two distinct processes that do not receive the abort-signal in  $E_{\rightarrow C}$ , and which both terminate in any  $\{a, b\}$ -fair execution starting in  $C$ . Then  $C$  is not strongly  $\{a, b\}$ -bivalent.*

**Proof.** Suppose  $C$  is strongly  $\{a, b\}$ -bivalent. Then it is  $\{a, b\}$ -bivalent, so

$$\mathcal{V}_A(C) = \{(lose, win), (win, lose)\}, \quad (8)$$

and if  $a$  or  $b$  runs solo starting in  $C$ , then that process wins. Because  $\sigma \in \mathcal{P}^*$ , neither  $a$  nor  $b$  receives the abort-signal in  $Exec(\Gamma, \sigma)$ . By the assumption that aborts are wait-free, processes  $a$  and  $b$  both terminate in sufficiently long solo runs starting in  $Conf(C, a^\top)$  and  $Conf(C, b^\top)$ , respectively. Let  $x$  and  $y$  be the return values of  $a$  in  $Exec(C, a^\top \circ a^{k_a})$  and of  $b$  in  $Exec(C, b^\top \circ b^{k_b})$ , respectively, for sufficiently large integers  $k_a$  and  $k_b$ .

Since  $Conf(C, a^\top) \sim_a Conf(C, a^\top b^\top)$ ,

$$a \text{ returns } x \text{ in } Exec(C, a^\top b^\top \circ a^{k_a}). \quad (9)$$

Similarly, since  $Conf(C, b^\top) \sim_b Conf(C, a^\top b^\top)$ ,

$$b \text{ returns } y \text{ in } Exec(C, a^\top b^\top \circ b^{k_b}). \quad (10)$$

We distinguish the following cases.

**Case 1:**  $x = y = win$ . In a sufficiently long solo-run by  $b$  following  $Exec(C, a^\top b^\top \circ a^{k_a})$ , process  $b$  must terminate (by deadlock-freedom). Since  $a$  wins in that execution,  $b$  must lose. Thus,

$$(win, lose) \in \mathcal{V}_A(Conf(C, a^\top b^\top)). \quad (11)$$

Applying a symmetric argument to a sufficiently long solo-run by  $a$  following  $Exec(C, b^\top a^\top \circ b^{k_b})$ , we obtain

$$(lose, win) \in \mathcal{V}_A(Conf(C, a^\top b^\top)). \quad (12)$$

Hence, using (8), we get  $\{(win, lose), (lose, win)\} = \mathcal{V}_A(Conf(C, a^\top b^\top))$ . Then by Lemma 5, there exists an infinite execution starting in  $Conf(C, a^\top b^\top)$ , such that  $a$  and  $b$  do not terminate. This contradicts wait-free aborts.

**Case 2:**  $x = y = lose$ . In a sufficiently long solo-run by  $b$  following  $Exec(C, a^\top b^\top \circ a^{k_a})$ , process  $b$  must terminate (by deadlock-freedom). Since  $a$  loses in that execution, by (8), process  $b$  must win. Thus,  $(lose, win) \in \mathcal{V}_A(Conf(C, a^\top b^\top))$ , and with a symmetric argument  $(win, lose) \in \mathcal{V}_A(Conf(C, a^\top b^\top))$ . We get a contradiction for the same reasons as in Case 1.

**Case 3:**  $\{x, y\} = \{win, lose\}$ . Without loss of generality, assume  $x = win$ . Then in  $Exec(C, a^\top a^{k_a})$  process  $a$  wins. On the other hand, since  $C$  is strongly bivalent,  $b$  wins in a sufficiently long solo-run starting in  $C$ . Since  $C \sim_b Conf(C, a^\top)$ , process  $b$  also wins in a long enough solo-run starting in  $Conf(C, a^\top)$ . Hence, we have shown that any of the two processes in  $\{a, b\}$  wins in a solo-run starting in  $Conf(C, a^\top)$ . By deadlock-freedom and (8) the other process loses, if it performs a long enough solo-run afterwards. This shows that  $Conf(C, a^\top)$  is strongly bivalent.

Now let  $A'$  be the 2-process algorithm in which  $a$  and  $b$  act exactly as in algorithm  $A$ , but the initial configuration is  $\Gamma' = Conf(C, a^\top)$ . Then  $A'$  is a deadlock-free abortable 2-process leader election algorithm with wait-free aborts: The wait-free abort property is inherited from  $A$ . Deadlock-freedom follows from the assumption that  $a$  and  $b$  terminate in any fair execution starting in  $C$ . Correctness follows from (8) and the fact that each process wins in a long enough solo-run starting in the initial configuration  $Conf(C, a^\top)$  (because that configuration is strongly bivalent).

Moreover, in  $A'$  process  $a$  always terminates within a finite number of its own steps. This follows from the wait-free abort property of  $A$  and the fact that both processes simulate  $A$  starting in configuration  $Conf(C, a^\top)$ , in which  $a$  has already received the abort-signal. This contradicts Lemma 7.  $\blacktriangleleft$

### 3.3 Constructing an Expensive Execution

We now consider an abortable leader election algorithm. We will construct a schedule such that in an execution starting in the initial configuration at least one process takes  $\Omega(\log n / \log \log n)$  RMR steps, where  $n$  is the number of processes.

#### 3.3.1 Additional Assumptions

We make the following assumptions that do not restrict the generality of our results. Recall that processes are state machines, each using some infinite state space  $\mathcal{Q}$ . We assume that during an execution a process never enters the same state twice. Further, we assume that each register stores a pair in  $\mathcal{P} \times (\mathcal{Q} \cup \{\perp\})$ , where  $\perp \notin \mathcal{Q}$ . The initial value of each register in  $\mathcal{R}_p$  is  $(p, \perp)$ , and when a process  $p$  writes to any register, it writes a pair  $(p, x)$ , where  $x$  is  $p$ 's state before its write operation, and in particular  $x \neq \perp$ . I.e., we are using a full information model, where processes write all information they have observed in the past. As a result, no two writes in an execution write the same value. Each process's first shared memory step is a read outside of its local shared memory segment, that we call *invocation read*, and thus incurs an RMR. Adding such a step to the beginning of each process's program does not affect the asymptotic RMR complexity of the algorithm. We will assume that at the end of its execution, each process  $p$  reads all registers in  $\mathcal{R}_p$  once. Since those reads do not incur any RMRs, this assumption can be made without loss of generality. We call  $p$ 's last read of register  $r \in \mathcal{R}_p$  the *terminating read* of  $r$ , and we assume that after  $p$ 's last terminating read,  $p$  will immediately enter a halting state.

#### 3.3.2 Terminology and Notation

We define some additional terms and notation.

We say process  $p$  is *visible* on register  $r$  in configuration  $C$  if  $val_C(r) = (p, x)$ , for some  $x \in \mathcal{Q}$ . Let  $L(C)$  be the set of processes that have lost in configuration  $C$ .

When we construct our high RMR execution, we need to make sure that whenever a process gains information about some other process that has not yet lost, someone pays for that with an RMR. To keep track of who knows who, we define a set  $K(C)$  that contains

pairs  $(p, q)$  of processes. Informally,  $(p, q)$  is in  $K(C)$  if  $p$  has already gained information about process  $q$  in the execution leading to configuration  $C$ , or  $p$  can gain such information for “free” (i.e., without an RMR being paid for that). Gaining information does not only mean that  $p$  reads a register that  $q$  has written; it means anything that might affect  $p$ 's execution, e.g.,  $p$ 's cache copies being invalidated.  $K(C)$  is the union of three sets  $K_1(C)$ ,  $K_2(C)$ , and  $K_3(C)$ , defined as follows:

- $K_1(C)$  is the set of all pairs  $(p, q)$ ,  $p \neq q$ , such that in  $E_{\rightarrow C}$  process  $p$  reads a register while process  $q$  is visible on that register. I.e.,  $p$  reads a value of  $(q, x)$ , where  $x \in \mathcal{Q}$ .  
*Informally:*  $p$  has learned about  $q$  in  $E_{\rightarrow C}$ .
- $K_2(C)$  is the set of all pairs  $(p, q)$ ,  $p \neq q$ , such that in  $E_{\rightarrow C}$  process  $q$  takes at least one shared memory step and process  $p$  reads a register in  $\mathcal{R}_q$ .  
*Informally:* Process  $p$  may have a valid cached copy of a register  $r \in \mathcal{R}_q$ , and by writing to  $r$  process  $q$  can invalidate that cached copy without incurring an RMR.
- $K_3(C)$  is the set of all pairs  $(p, q)$ ,  $p \neq q$ , such that in  $E_{\rightarrow C}$  process  $p$  takes at least one shared memory step, and  $q$  writes to a register  $r \in \mathcal{R}_p$  before  $p$ 's terminating read of  $r$ .  
*Informally:*  $p$  may learn about  $q$  without incurring an RMR by scanning all its registers in  $\mathcal{R}_p$ .

Let  $K(C) = K_1(C) \cup K_2(C) \cup K_3(C)$ . We say process  $p$  *knows* process  $q$  in configuration  $C$  if  $(p, q) \in K(C)$ .

In our inductive construction of an RMR expensive execution, we will sometimes erase processes from the constructed execution. For that reason, if  $p$  knows about  $q$ , i.e.,  $(p, q) \in K(C)$ , then we will not remove a process  $q$  from the execution  $E_{\rightarrow C}$ . We achieve this by ensuring that whenever  $(p, q) \in K(C)$ ,  $q \in L(C)$ , and as discussed earlier no lost processes will be erased.

However, we have to be careful about cases in which  $p$  does not know directly about  $q$ . For example, suppose process  $q$  writes to register  $r$  in execution  $E$ , and later some process  $z$  overwrites  $r$  and finally  $p$  becomes poised to read  $r$ . In our inductive construction we may want to remove either  $z$  or  $p$  from the execution, because we do not want  $z$  to be discovered by  $p$ . However, removing  $z$  reveals  $q$  on register  $r$ , and so now  $p$  may discover  $q$ . To account for that we introduce the concept of *hidden* processes.

In particular, for a configuration  $C$  and a register  $r$  we define a set  $H_r(C)$  of processes *hidden* on  $r$  as follows:

- (H1) For  $r \notin \mathcal{R}_p$ ,  $p \in H_r(C)$  if and only if either  $p$  does not access  $r$  in  $E_{\rightarrow C}$ , or  $p$  accesses  $r$  in  $E_{\rightarrow C}$  at some point  $t$ , and either no process writes  $r$  after  $t$ , or at least one process that writes  $r$  after  $t$  is in  $L(C)$ ;  
*Idea:* If  $p$ 's write to  $r$  was overwritten by some processes, then at least one of them has lost and thus will not be erased from the execution. Hence, erasing a process does not reveal  $p$ 's write to any other process.
- (H2) For  $r \in \mathcal{R}_p$ ,  $p \in H_r(C)$  if and only if any process other than  $p$  that writes to  $r$  in  $E_{\rightarrow C}$  is in  $L(C)$ .  
*Idea:* If a process  $q$  wrote to a register  $r$  in  $p$ 's local memory segment, then  $q$  has lost. Therefore,  $q$  will not be erased from the execution. This is important because  $p$  can read  $r$  for free and we have to assume that it does so frequently, so erasing  $q$  from the execution might change what  $p$  observes in the execution.

Let  $H(C) = \bigcap_{r \in \mathcal{R}} H_r(C)$ . We say process  $p$  is *hidden* in configuration  $C$ , if  $p \in H(C)$ . We finally define the concept of a safe configuration as follows. Configuration  $C$  is *safe*, if



(S1) for any pair  $(p, q) \in K(C)$ ,  $q \in L(C)$ , and

(S2) if  $p \notin H(C)$ , then either  $p \in L(C)$ , or  $p$  takes no shared memory step in  $E \rightarrow C$ .

The first property ensures that no process  $p$  knows another process  $q$  that has not yet lost, and the second property says that all processes that are not hidden must have lost, or not even started participation. As a result, in an execution leading to a safe configuration, we can erase all processes that do not lose, without affecting any other processes. Formally, we will prove for a schedule  $\sigma$ , a safe configuration  $C = \text{Conf}(\Gamma, \sigma)$  and a set of processes  $P \supseteq L(C)$ ,

- $\text{Exec}(\Gamma, \sigma)|P = \text{Exec}(\Gamma, \sigma|P^\Delta)$ ;
- $\text{RMR}_P(\text{Exec}(\Gamma, \sigma)) = \text{RMR}_P(\text{Exec}(\Gamma, \sigma|P^\Delta))$ ; and
- $\text{Cache}_p(C) = \text{Cache}_p(\text{Conf}(\Gamma, \sigma|P^\Delta))$  for all  $p \in P$ .

Moreover, if  $C$  is safe, then  $\text{Conf}(\Gamma, \sigma|P^\Delta)$  is also safe.

### 3.3.3 Overview of the Construction

Let  $n \geq 4$ ,  $\ell = \lfloor \log n / c \log \log n \rfloor$  for some sufficiently large constant  $c$ . We inductively construct a schedule  $\sigma_i$  and a set of processes  $P_i \subseteq \mathcal{P}$ , for all  $i \in \{0, \dots, \ell\}$ . For the sake of conciseness, let  $E_i = \text{Exec}(\Gamma, \sigma_i)$ ,  $C_i = \text{Conf}(\Gamma, \sigma_i)$ , and  $L_i = L(C_i)$ .

The construction will satisfy the following invariants for  $i \in \{0, \dots, \ell\}$ :

- (I1)  $C_i$  is safe.
- (I2)  $|P_i \setminus L_i| \geq (n-1)/(\log n)^{c^i}$ .
- (I3)  $\text{RMR}_{P_i \setminus L_i}(C_i) \geq i|P_i \setminus L_i| - i$ .
- (I4) For each process  $p \in P_i \setminus L_i$ :  $\text{RMR}_p(C_i) \leq i$ .
- (I5) For each process  $p \in P_i \setminus L_i$ ,  $p^\top$  does not appear in  $\sigma_i$ .

Invariant (I2) for  $i = \ell$  implies  $|P_\ell \setminus L_\ell| \geq 2$ . Hence, by (I3) there are at least two processes that each incur  $\Omega(\ell) = \Omega(\log n / \log \log n)$  RMRs. Theorem 3 follows.

We now sketch how we construct  $\sigma_i$  and  $P_i$  inductively so that the invariants are satisfied. We start with  $P_0 = \mathcal{P}$  and the initial configuration  $C_0$ . We then schedule processes in rounds. In round  $i$ , we choose a subset  $P_{i+1}$  of the processes in  $P_i \setminus L_i$  and remove all processes in  $\mathcal{P} \setminus (P_{i+1} \cup L_i)$  from the execution constructed so far. This does not affect any of the remaining processes, because  $C_i$  is safe. Then we schedule the processes in  $P_{i+1}$  in such a way that each of them incurs an RMR, and only a small fraction of them lose.

To decide which processes to remove and to schedule the remaining processes, we proceed as follows: First we let each process in  $P_i \setminus L_i$  take sufficiently many steps until it is poised to incur an RMR. It is not hard to see that in an execution in which no process incurs an RMR, processes do not learn about each other, so the resulting configuration,  $D_i$ , is again safe. Moreover, in a safe configuration processes only know about lost processes, so they cannot lose.

We then distinguish between a high contention write case, where a majority of processes are poised to write to few registers, and a low contention case, where either many registers are covered by processes poised to write, or a majority of processes are poised to read. Let  $S_i$  be the set of registers processes in  $P_i \setminus L_i$  are poised to access in configuration  $D_i$ . The high contention write case occurs if there are few such registers and a majority of processes are poised to write, i.e.,  $|S_i| = O(|P_i \setminus L_i| / \log n)$ , and otherwise the low contention write case occurs.

In the low contention write case, we choose a set  $Q_i$  of processes, which contains for each register  $r \in S_i$  at most one process poised to write to  $r$  in  $D_i$ . We consider the step  $s_p$  each process  $p \in Q_i$  is poised to take. We then create a directed graph  $G$  with processes

as vertices, and an edge from  $p$  to  $q$  if in the resulting configuration (I) due to  $s_p$  or  $s_q$  process  $p$  knows  $q$ , or (II) due to step  $s_p$  process  $q$  is not hidden. Each application of rule (I) must be paid for by RMRs in the execution, and for each application of (II) a process  $p$  must overwrite some process  $q$ . As a result graph  $G$  is sufficiently sparse, and by Turán's theorem [44] we obtain a large independent set  $J$ . We let each process  $p \in J$  take one step,  $s_p$ , and erase all remaining processes that haven't lost yet from the execution. It is not hard to see that no process loses in any of the steps added, the resulting configuration is safe (this follows from how we added edges to  $G$ ) and, because of the sparsity of the graph, a sufficiently large number of processes survive. From that we obtain Invariants (I1) and (I2). Since each process  $p$  performs an RMR in step  $s_p$  and only local steps before that, we get (I3) and (I4). Moreover, we don't abort any processes, so (I5) is true.

In the high contention write case, we erase all readers from the execution. For each register  $r \in S_i$ , let  $W_r$  denote the set of processes poised to write to  $r$ . Since this is a high contention case,  $|W_r|$  is large for most registers  $r$ . For each register  $r$  with sufficiently large  $|W_r|$ , we choose two distinct processes  $a, b \in W_r$ .

We then argue that, after erasing some  $O(\log n)$  processes, we obtain a configuration  $D'_i$  and an  $\{a, b\}$ -only schedule  $\sigma$  such that in execution  $Exec(D'_i, \sigma)$  processes  $a$  and  $b$  both lose and see no process other than those in  $L_i$ , which have lost already. The argument is based on Lemma 8, but quite involved. We now let, starting from  $D'_i$ , all processes in  $W_r \setminus \{a, b\}$  execute one step, in which they write to  $r$ . After that we schedule  $a$  and  $b$  as prescribed by  $\sigma$ . Then  $a$  and  $b$  will both first write to  $r$ , and thus overwrite the writes by all other processes in  $W_r$ , then continue to take steps and lose without seeing any processes that haven't lost, yet. As a result, all processes in  $W_r \setminus \{a, b\}$  have taken a step but are now hidden, two processes ( $a$  and  $b$ ) have lost, and  $O(\log n)$  processes have been removed. It is not hard to see that the resulting configuration is safe again. We repeat this for all registers  $r$  for which  $|W_r|$  is large enough. Then, we let  $P_{i+1}$  denote the set of all surviving processes and  $C_{i+1}$  the resulting configuration.

Configuration  $C_{i+1}$  is safe, and sufficiently few processes are removed or have lost so that (I1) and (I2) remain true. Moreover, each process that does not lose performs exactly one RMR, so (I3) and (I4) are true. (I5) is true because all processes that received the abort signal lost.

---

## References

- 1 Zahra Aghazadeh, Wojciech Golab, and Philipp Woelfel. Making objects writable. In *Proceedings of the 33rd SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 385–395, 2014. doi:10.1145/2611462.2611483.
- 2 Zahra Aghazadeh and Philipp Woelfel. Space- and time-efficient long-lived test-and-set objects. In *Proceedings of 18th International Conference On Principles Of Distributed Systems (OPODIS)*, pages 404–419, 2014. doi:10.1007/978-3-319-14472-6\_27.
- 3 Zahra Aghazadeh and Philipp Woelfel. Upper bounds for boundless tagging with bounded objects. In *Proceedings of the 30th International Symposium on Distributed Computing (DISC)*, pages 442–457, 2016. doi:10.1007/978-3-662-53426-7\_32.
- 4 Marcos Aguilera, Svend Frølund, Vassos Hadzilacos, Stephanie Lorraine Horn, and Sam Toueg. Abortable and query-abortable objects and their efficient implementation. In *Proceedings of the 26th SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 23–32, 2007.
- 5 Dan Alistarh and James Aspnes. Sub-logarithmic test-and-set against a weak adversary. In *Proceedings of the 25th International Symposium on Distributed Computing (DISC)*, pages 97–109, 2011.



- 6 Dan Alistarh, James Aspnes, Keren Censor-Hillel, Seth Gilbert, and Morteza Zadimoghaddam. Optimal-time adaptive strong renaming, with applications to counting. In *Proceedings of the 30th SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 239–248, 2011.
- 7 Dan Alistarh, James Aspnes, Seth Gilbert, and Rachid Guerraoui. The complexity of renaming. In *Proceedings of the 52nd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 718–727, 2011. doi:10.1109/FOCS.2011.66.
- 8 Dan Alistarh, Hagit Attiya, Seth Gilbert, Andrei Giurgiu, and Rachid Guerraoui. Fast randomized test-and-set and renaming. In *Proceedings of the 24th International Symposium on Distributed Computing (DISC)*, pages 94–108, 2010.
- 9 James H. Anderson and Yong-Jik Kim. Adaptive mutual exclusion with local spinning. In *Proceedings of the 14th International Symposium on Distributed Computing (DISC)*, pages 29–43, 2000.
- 10 James H. Anderson and Yong-Jik Kim. An improved lower bound for the time complexity of mutual exclusion. *Distributed Computing*, 15:221–253, 2002.
- 11 T. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1:6–16, 1990.
- 12 Hagit Attiya, Rachid Guerraoui, Danny Hendler, and Petr Kuznetsov. The complexity of obstruction-free implementations. *Journal of the ACM*, 56(4):24:1–24:33, 2009. doi:10.1145/1538902.1538908.
- 13 Hagit Attiya, Danny Hendler, and Philipp Woelfel. Tight RMR lower bounds for mutual exclusion and other problems. In *Proceedings of the 40th Annual ACM Symposium on Theory of Computing (STOC)*, pages 217–226, 2008.
- 14 Michael Bender and Seth Gilbert. Mutual exclusion with  $O(\log^2 \log n)$  amortized work. In *Proceedings of the 52nd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 728–737, 2011.
- 15 Harry Buhrman, Alessandro Panconesi, Riccardo Silvestri, and Paul Vitányi. On the importance of having an identity or, is consensus really universal? *Distributed Computing*, 18(3):167–176, 2006. doi:10.1007/s00446-005-0121-z.
- 16 Robert Danek and Wojciech Golab. Closing the complexity gap between FCFS mutual exclusion and mutual exclusion. *Distributed Computing*, 23(2):87–111, 2010. doi:10.1007/s00446-010-0096-2.
- 17 Robert Danek and Hyonho Lee. Brief announcement: Local-spin algorithms for abortable mutual exclusion and related problems. In *Proceedings of the 22nd International Symposium on Distributed Computing (DISC)*, pages 512–513, 2008. doi:10.1007/978-3-540-87779-0\_41.
- 18 E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8:569, 1965.
- 19 Cynthia Dwork, Maurice Herlihy, and Orli Waarts. Contention in shared memory algorithms. *Journal of the ACM*, 44(6):779–805, 1997. doi:10.1145/268999.269000.
- 20 Wayne Eberly, Lisa Higham, and Jolanta Warpechowska-Gruca. Long-lived, fast, wait-free renaming with optimal name space and high throughput. In *Proceedings of the 12th International Symposium on Distributed Computing (DISC)*, pages 149–160, 1998.
- 21 Aryaz Eghbali and Philipp Woelfel. An almost tight RMR lower bound for abortable test-and-set. *CoRR*, abs/1805.04840, 2018. arXiv:1805.04840.
- 22 Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985. doi:10.1145/3149.214121.

- 23 George Giakkoupis and Philipp Woelfel. On the time and space complexity of randomized test-and-set. In *Proceedings of the 31st SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 19–28, 2012. doi:10.1145/2332432.2332436.
- 24 George Giakkoupis and Philipp Woelfel. A tight RMR lower bound for randomized mutual exclusion. In *Proceedings of the 44th Annual ACM Symposium on Theory of Computing (STOC)*, pages 983–1002, 2012. doi:10.1145/2213977.2214066.
- 25 George Giakkoupis and Philipp Woelfel. Randomized mutual exclusion with constant amortized RMR complexity on the DSM. In *Proceedings of the 55th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, 2014. To appear.
- 26 George Giakkoupis and Philipp Woelfel. Randomized abortable mutual exclusion with constant amortized RMR complexity on the CC model. In *Proceedings of the 36th SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 221–229, 2017. doi:10.1145/3087801.3087837.
- 27 Wojciech Golab, Danny Hendler, and Philipp Woelfel. An  $O(1)$  RMRs leader election algorithm. *SIAM Journal on Computing*, 39(7):2726–2760, 2010.
- 28 Wojciech M. Golab, Vassos Hadzilacos, Danny Hendler, and Philipp Woelfel. Constant-RMR implementations of cas and other synchronization primitives using read and write operations. In *Proceedings of the 26th SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 3–12, 2007.
- 29 Wojciech M. Golab, Vassos Hadzilacos, Danny Hendler, and Philipp Woelfel. RMR-efficient implementations of comparison primitives using read and write operations. *Distributed Computing*, 25(2):109–162, 2012. doi:10.1007/s00446-011-0150-8.
- 30 Danny Hendler and Philipp Woelfel. Randomized mutual exclusion in  $O(\log N/\log \log N)$  RMRs. In *Proceedings of the 28th SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 26–35, 2009.
- 31 Danny Hendler and Philipp Woelfel. Adaptive randomized mutual exclusion in sub-logarithmic expected time. In *Proceedings of the 29th SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 141–150, 2010.
- 32 Danny Hendler and Philipp Woelfel. Randomized mutual exclusion with sub-logarithmic RMR-complexity. *Distributed Computing*, 24(1):3–19, 2011. doi:10.1007/s00446-011-0128-6.
- 33 Prasad Jayanti. Adaptive and efficient abortable mutual exclusion. In *Proceedings of the 22nd SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 295–304, 2003. doi:10.1145/872035.872079.
- 34 Prasad Jayanti, Srdjan Petrovic, and Neha Narula. Read/write based fast-path transformation for FCFS mutual exclusion. In *31st Conference on Current Trends in Theory and Practice of Informatics (SOFSEM)*, pages 209–218, 2005.
- 35 Y.-J. Kim and J. Anderson. A time complexity bound for adaptive mutual exclusion. In *Proceedings of the 15th International Symposium on Distributed Computing (DISC)*, pages 1–15, 2001.
- 36 Yong-Jik Kim and James H. Anderson. Nonatomic mutual exclusion with local spinning. *Distributed Computing*, 19(1):19–61, 2006. doi:10.1007/s00446-006-0003-z.
- 37 Clyde P. Kruskal, Larry Rudolph, and Marc Snir. Efficient synchronization on multiprocessors with shared memory. *ACM Transactions on Programming Languages and Systems*, 10(4):579–601, 1988. doi:10.1145/48022.48024.
- 38 Hyonho Lee. Transformations of mutual exclusion algorithms from the cache-coherent model to the distributed shared memory model. In *Proceedings of the 25th International Conference on Distributed Computing Systems (ICDCS)*, pages 261–270, 2005. doi:10.1109/ICDCS.2005.83.

- 39 Hyonho Lee. Fast local-spin abortable mutual exclusion with bounded space. In *Proceedings of 14th International Conference On Principles Of Distributed Systems (OPODIS)*, pages 364–379, 2010. doi:10.1007/978-3-642-17653-1\_27.
- 40 Hyonho Lee. *Local-spin Abortable Mutual Exclusion*. PhD thesis, University of Toronto, 2011.
- 41 Alessandro Panconesi, Marina Papatriantafidou, Philippos Tsigas, and Paul M. B. Vitányi. Randomized naming using wait-free shared variables. *Distributed Computing*, 11(3):113–124, 1998.
- 42 Abhijeet Pareek and Philipp Woelfel. RMR-efficient randomized abortable mutual exclusion. In *Proceedings of the 26th International Symposium on Distributed Computing (DISC)*, pages 267–281, 2012. doi:10.1007/978-3-642-33651-5\_19.
- 43 Michael L Scott. Non-blocking timeout in scalable queue-based spin locks. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 31–40. ACM, 2002.
- 44 Paul Turán. Eine extremalaufgabe aus der graphentheorie. *Mat. Fiz. Lapok*, 48(436-452):61, 1941.



# Distributed Set Cover Approximation: Primal-Dual with Optimal Locality

**Guy Even**


Tel-Aviv University, Israel  
guy@eng.tau.ac.il

**Mohsen Ghaffari**

ETH Zurich, Switzerland  
ghaffari@inf.ethz.ch

**Moti Medina**

Ben-Gurion University, Israel  
medinamo@bgu.ac.il

 <https://orcid.org/0000-0002-5572-3754>

---

## Abstract

This paper presents a deterministic distributed algorithm for computing an  $f(1+\varepsilon)$  approximation of the well-studied minimum *set cover* problem, for any constant  $\varepsilon > 0$ , in  $O(\log(f\Delta)/\log\log(f\Delta))$  rounds. Here,  $f$  denotes the maximum element frequency and  $\Delta$  denotes the cardinality of the largest set. This  $f(1+\varepsilon)$  approximation almost matches the  $f$ -approximation guarantee of standard centralized primal-dual algorithms, which is known to be essentially the best possible approximation for polynomial-time computations. The round complexity almost matches the  $\Omega(\log(\Delta)/\log\log(\Delta))$  lower bound of Kuhn, Moscibroda, Wattenhofer [JACM'16], which holds for even  $f = 2$  and for any  $\text{poly}(\log\Delta)$  approximation. Our algorithm also gives an alternative way to reproduce the time-optimal  $2(1+\varepsilon)$ -approximation of vertex cover, with round complexity  $O(\log\Delta/\log\log\Delta)$ , as presented by Bar-Yehuda, Censor-Hillel, and Schwartzman [PODC'17] for weighted vertex cover. Our method is quite different and it can be viewed as a locality-optimal way of performing primal-dual for the more general case of set cover. We note that the vertex cover algorithm of Bar-Yehuda et al. does not extend to set cover (when  $f \geq 3$ ).

**2012 ACM Subject Classification** Theory of computation  $\rightarrow$  Graph algorithms analysis, Mathematics of computing  $\rightarrow$  Graph algorithms, Theory of computation  $\rightarrow$  Distributed algorithms

**Keywords and phrases** Distributed Algorithms, Approximation Algorithms, Set Cover, Vertex Cover

**Digital Object Identifier** 10.4230/LIPIcs.DISC.2018.22

**Acknowledgements** Part of this work was done while the authors were visiting the Max Planck Institute for Informatics.

## 1 Introduction and Related Work

The *set cover* problem is one of the central problems in the study of approximation algorithms. For instance, the first chapter of the textbook of Williamson and Shmoys [27] is dedicated to illustrating “several of the central techniques of the book applied to a single problem, the set cover problem.” In this paper, we present the first time-optimal distributed approximation algorithm for the set cover problem, with an approximation guarantee that essentially matches the best known centralized approximation. Let us elaborate on this by first recalling the problem statement and centralized approximation bounds, as well as the distributed model of computation in the study of this problem.



© Guy Even, Mohsen Ghaffari, and Moti Medina;  
licensed under Creative Commons License CC-BY

32nd International Symposium on Distributed Computing (DISC 2018).

Editors: Ulrich Schmid and Josef Widder; Article No. 22; pp. 22:1–22:14

Leibniz International Proceedings in Informatics



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

## 1.1 Background

**Set Cover.** We are given a ground set of elements  $U$  and some sets  $S_1, S_2, \dots, S_k \subseteq U$ . The objective is to find a minimum-cardinality collection of the sets that covers all the elements, i.e., a collection  $I \subseteq \{1, \dots, k\}$  that minimizes  $|I|$  subject to  $\cup_{i \in I} S_i = U$ .

**Known Centralized Approximations and Inapproximability Bounds.** For each element  $u \in U$ , we use  $f_u = |\{i \mid u \in S_i\}|$  to denote the frequency of this element, i.e., the number of sets that contain  $u$ . We also use  $f$  to denote the maximum frequency among all elements, i.e.  $f = \max_{u \in U} f_u$ . A standard approximation guarantee for the set cover problem is an  $f$ -approximation, see e.g. [4, Theorem 2] or [27, Theorem 1.6] or [26, Theorem 15.2]. Moreover, this approximation is known to be nearly the best possible for polynomial-time central algorithms: For the special case  $f = 2$  (when the problem is better known as the *vertex cover* problem), Dinur and Safra [9] proved NP-hardness of 1.36 approximation, improving on a  $7/6 - \epsilon$  hardness by Hastad [14]. For general  $f$ , the inapproximability has been improved in a sequence of papers: Trevisan gave an  $\Omega(f^{1/19})$  bound [25]; Holmerin gave an  $\Omega(f^{1-\epsilon})$  bound [15]; Dinur, Guruswami, and Khot improved that to  $f - 3 - \epsilon$ ; which was then improved by Dinur, Guruswami, Khot, and Regev to  $f - 1 - \epsilon$  [8]. Furthermore, assuming the Unique Games Conjecture, Khot and Regev proved an inapproximability of  $f - \epsilon$  [16]. We remark that another approximation bound for the set cover is  $\ln |U|$  – see [27, Theorem 1.11]. This bound is of interest when the frequency of appearances of the elements in different sets is large. Moreover, this bound is also known to be the nearly the best possible in the worst-case: A series of works by Lund and Yannakakis [21], Feige [10], and Moshkovitz [22] showed that it is NP-hard to always approximate set cover to within  $(1 - \epsilon) \ln |U|$ , for any constant  $\epsilon > 0$ . We note that although the standard way of formulating the upper bound is  $\ln |U|$ , the actual bound can be written more precisely as  $\ln \Delta$  where  $\Delta$  denotes the cardinality of the largest set.

**Distributed Computation Model.** We consider the CONGEST [23] model, which is the standard synchronous message passing model in distributed computing. In this model, the network is abstracted as a simple graph  $G = (V, E)$  where  $n = |V|$ . There is one processor on each node of the network, which initially does not know the topology of the network. These processors can communicate in synchronous rounds where per round each processor/node can send one  $O(\log n)$  bit message to each of its neighbors.

**Distributed Formulation of Set Cover.** The standard distributed formulation of the set cover problem (see, e.g., [19]) is that we have one processor for each element in the ground set  $U$ , and also one processor for each of the sets  $S_1, S_2, \dots, S_k \subseteq U$ . The network is the natural corresponding (bipartite) graph where each element-processor is connected to the set-processors whose set contains this element. Communications on this network follow the CONGEST model of synchronous message passing, as explained above.

The above is a natural formulation. As prototypical examples, it captures the following settings: cases where we want to select as few as possible of the servers so that they can serve all of the clients, when each element can be served only by certain servers; and cases where we want to select as few as possible of monitoring agents who can control all workers, where each worker can be controlled only by certain monitoring agents<sup>1</sup>.

---

<sup>1</sup> Of course, in the practical version of each of these problems, there might be many more constraints or optimization objectives. However, that goes beyond the objective of our paper, which is to characterize the complexity of a basic and fundamental problem in distributed approximation algorithms.

## 1.2 Our Result

We present a deterministic distributed algorithm that almost matches the  $f$ -approximation mentioned above, up to a  $(1 + \varepsilon)$  factor for any arbitrary small constant  $\varepsilon > 0$ , in a time-complexity that is provably optimal:

► **Theorem 1.** *There is a deterministic distributed algorithm in the CONGEST model that computes an  $f(1 + \varepsilon)$  approximation of minimum set-cover, in  $O\left(\frac{\log(f\Delta)}{\varepsilon \log \log(f\Delta)}\right)$  rounds, in any set-system of frequency  $f$  and maximum set size  $\Delta$ , and for any  $0 < \varepsilon < 1$ . Moreover, the algorithm operates on an anonymous network and uses messages of length  $O(\varepsilon^{-1} \cdot \log(f \cdot \Delta))$ .*

The matching lower bound is due to a celebrated work of Kuhn, Moscibroda, and Wattenhofer [19]: they show that even the simple case of  $f = 2$ , where the set cover problem boils down to vertex cover, has a lower bound of  $\Omega\left(\frac{\log \Delta}{\log \log \Delta}\right)$  rounds, for any approximation up to  $\text{poly}(\log \Delta)$ . Moreover, for all cases of interest for  $f$ -approximation – i.e., when  $f$  is smaller than the other known approximation bound  $\ln \Delta$  –, the  $O\left(\frac{\log(f\Delta)}{\log \log(f\Delta)}\right)$  round complexity of the above algorithm asymptotically matches the  $\Omega\left(\frac{\log \Delta}{\log \log \Delta}\right)$  lower bound.

We note that coming up with a deterministic distributed algorithm that achieves  $\text{poly} \log \Delta$  (or even  $\text{poly} \log n$ ) approximation for set cover, even with unbounded size messages, with  $\text{poly} \log n$  number of rounds, where  $n$  is the number of processors in the network, would be a major breakthrough: as shown recently in [12, Theorem 7.5], it would imply that any randomized distributed algorithm with  $\text{poly} \log n$  number of rounds for any locally checkable problem can be derandomized and solved in  $\text{poly} \log n$  number of rounds deterministically. This includes computing a Maximal Independent Set in  $\text{poly} \log n$  number of rounds, which is an open question by Linial since the 80's [20].

## 1.3 The Main Related Work and Comparison of Techniques

**Sequential Primal-Dual.** A standard centralized approximation algorithm that gives an  $f$ -approximation for set cover is the one based on the primal-dual schema. See, e.g., Bar-Yehuda and Even [4] or Vazirani's textbook [26, Section 15.2] for a comprehensive description. Summarized, this schema works roughly as follows: there is a variable  $y_u \in [0, 1]$  for each element  $u \in U$ ; these are known as dual variables. Until all elements are covered, we iteratively pick an uncovered element, say  $u$ , and we raise its variable  $y_u$  until for one of the sets containing  $u$ , say  $S_i$ , we have  $\sum_{u' \in S_i} y_{u'} = 1$ . We call such a set tight (because its constraint in the primal linear program is tight). Then we add this tight set to the set cover to be outputted at the end, and we consider all of its elements covered. As shown in [4, Theorem 2], and [26, Theorem 15.3], this method gives an  $f$ -approximation.

**Standard Distributed Primal-Dual.** The above method is clearly sequential. However, one can easily adapt it to the distributed setting<sup>2</sup>, when we relax the approximation factor to  $f(1 + \varepsilon)$  for any arbitrarily small constant  $\varepsilon > 0$ . Initially, set  $y_u = 1/\Delta$  for each element  $u \in U$ . Then, in each iteration, we do as follows: (1) for each set  $S_i$  that has  $\sum_{u' \in S_i} y_{u'} \geq 1 - \varepsilon/2$ , add this set  $S_i$  to the output set cover (all at the same time) and consider all of its elements covered. Then, for each uncovered element  $u$ , set  $y_u \leftarrow y_u \cdot \frac{1}{1 - \varepsilon/2}$ . The method terminates in  $O(\log \Delta / \varepsilon)$  rounds and outputs an  $f(1 + \varepsilon)$  approximation.

<sup>2</sup> In fact, this adaptation is so simple and well-known that we are not sure what is the reference for it (or its first appearance). The analysis follows directly from [26, Proposition 15.1].



As a side comment, we add that Kuhn et al. [18,19] give a general algorithm for obtaining a  $(1 + \varepsilon)$  approximation of fractional packing linear programs, which can then be turned into an integral solution for  $f(1 + \varepsilon)$  approximation of set cover via a simple deterministic rounding. However, the resulting algorithm would be slower than the above.

**Sped-up Distributed Solution, via the Local-Ratio Method.** In an elegant recent work, Bar-Yehuda, Censor-Hillel, and Schwartzman [3] presented an improved algorithm for the special case of  $f = 2$  (i.e., vertex cover), based on the local-ratio method [5] which itself is closely related to the primal-dual scheme [6]. Their algorithm improves the round complexity for  $f(1 + \varepsilon) = 2(1 + \varepsilon)$  approximation of vertex cover to the optimal bound of  $O(\log \Delta / (\varepsilon \log \log \Delta))$ . Their algorithm also works for the weighted variant of the vertex cover problem. This round complexity matches the lower bound of Kuhn, Moscibroda, and Wattenhofer [19]. However, the algorithm of [3] seems especially crafted for the case of  $f = 2$  and it does not generalize<sup>3</sup> to even  $f = 3$ . In a very rough sense, the limitation is as follows: the method works by dividing the leftover space in dual constraints (i.e.,  $1 - \sum_{u' \in S_i} y_{u'}$ ) into two parts, a vault and a bank. The vault is used to initiate requests for increases in the dual variables (i.e.,  $y_{u'}$ ) and the bank is used to securely accept these dual variable increases, while making sure that  $\sum_{u' \in S_i} y_{u'} \leq 1$ . When trying to extend this to  $f = 3$  or higher, it is not clear how to make all the sets containing one element agree consistently with the amount of the raise in the dual variable, while respecting their own individual  $\sum_{u' \in S_i} y_{u'} \leq 1$  constraints, and without slowing down the process too much.

**Our Method, in a Nutshell.** We also follow the primal-dual schema. But our method can be viewed as an improved and more general way of performing primal-dual distributedly, with optimal locality (i.e., round complexity) for set cover. In a very rough sense, it is based on a natural dynamic process that, over time, flexibly adjusts the amount of increase per each dual variable, while (1) not violating any of the constraints, (2) maintaining a large step of increase for most variables at most times. We are hopeful that dynamics of the same style may lead to improvements for many other optimization problems.

**A conceptual contribution, in the context of randomized Maximal Independent Set Algorithms [2, 11].** Besides the improvement in the round complexity of the set-cover problem, we think of our solution as shedding some light on some other known prior work [2,11]. The dynamic process that we use for adjusting the increase steps in dual variables is closely related to the randomized maximal independent set algorithm of Ghaffari [11]. We note that a parameter-optimized version of the latter was used by Bar-Yehuda, Censor-Hillel, Ghaffari, and Schwartzman [2] to obtain a  $2(1 + \varepsilon)$  approximation of maximum matching in  $O(\log \Delta / \log \log \Delta)$  rounds. However, the place where we use the general dynamic process appears quite different than those of [2,11]. While in those previous papers the dynamic process was set up to adjust the probability of trying to join the MIS (or the nearly maximal matching), in our current paper, the dynamic process is used in a fully deterministic way and it governs the adjustments in the increase step of dual variables. In hindsight, this suggests (in an informal way) that one can view the probabilities in Ghaffari’s MIS algorithm [11] as fractional solutions to some linear program. The dynamic process tries to adjust these probabilities towards the “sweet spot” where per round many nodes get hit (by either joining

---

<sup>3</sup> We have also double checked this with Gregory Schwartzman, through personal communication.



MIS or having a neighbor join MIS). This is reminiscent of the standard randomized rounding method in design of approximation algorithms, where one first finds a good fractional solution to a suitable linear program formulation, and then performs a randomized rounding to turn these fractional solutions to integral; see e.g., [27, Section 1.7]. The difference is that the algorithm of [11] does not wait for these fractional variables to reach the sweet spot and only then do the rounding (i.e., deciding probabilistically for various elements). It instead performs a certain “iterative rounding” where even the interim fractional values are used for attempts of forming a good integral solution (an independent set that is adjacent to a large set of vertices).

## 1.4 Other related work

In this section we survey other related work. We start with results where  $f = 2$ , i.e., vertex cover. Recently, Ben-Basat, Even, Kawarabayashi, and Schwartzman [7] presented a 2-approximation algorithm for minimum weighted vertex cover in CONGEST with round complexity of  $O\left(\frac{\log n \log \Delta}{(\log \log \Delta)^2}\right)$ . Their approach generalizes the  $(2+\varepsilon)$ -approximation algorithm of [3] and improves the dependency on  $\varepsilon^{-1}$  to logarithmic. For a detailed overview of work on vertex cover we refer the reader to [1, 3].

We now turn to results for general  $f$ : Koufogiannakis and Young [17] presented a distributed algorithm for weighted set cover in the LOCAL model. Their algorithm achieves an approximation ratio of  $f$  in  $O(\log^2 m)$  rounds w.h.p, where  $m$  is the number of elements. Kuhn et al. [18, 19] studied covering and packing linear programs in the LOCAL model and obtained a  $(1 + \varepsilon)$ -approximation algorithm in  $O(\varepsilon^{-1} \log n)$  rounds w.h.p., where  $n$  is the number of primal and dual variables. Ghaffari, Kuhn, and Maus [13] presented a randomized distributed approximation scheme (i.e.,  $(1 + \varepsilon)$ -approximation) for arbitrary distributed covering and packing integer linear programs in the LOCAL model with round complexity  $O(\text{poly log}(n/\varepsilon))$  w.h.p., where  $n$  is the number of primal and dual variables. For more results in the LOCAL model we refer the reader to the survey by [24].

## 2 Problem Definition and Model of Computation

In this section we introduce the problem of *vertex cover in hypergraphs* (VCH). Designing a distributed CONGEST algorithm for VCH directly translates to an algorithm for set cover.

### 2.1 Preliminaries

A hypergraph  $H$  is a pair  $(V, E)$  where  $V$  denotes the set of vertices and  $E \subseteq 2^V$ . Every hyperedge  $e \in E$  is a nonempty subset of vertices. The *maximum degree* of the graph  $G$  is denoted by  $\Delta$ , and defined by  $\Delta \triangleq \max_v |\{e \in E \mid v \in e\}|$ . The *rank* of  $H$  is denoted by  $f$ , and defined by  $f \triangleq \max_{e \in E} |e|$ .

### 2.2 Vertex Cover in Hypergraphs (VCH)

A subset  $C \subset V$  is a *vertex cover* in  $H = (V, E)$  if  $C \cap e \neq \emptyset$ , for every hyperedge  $e \in E$ . The minimum cardinality vertex cover problem in hypergraphs is defined as follows.

- Problem:** Minimum Cardinality Vertex Cover in Hypergraphs (VCH)  
**Instance:** A hypergraph  $H = (V, E)$ .  
**Solution:** A vertex cover  $C$ .  
**Objective:** Minimize the cardinality of the cover  $C$ .

We denote the cardinality of an optimal vertex cover by  $\text{opt}$ .

Note that the VCH problem translates to set cover as follows:

- (i) Each element in the ground set  $U$  is an hyperedge in the VCH formulation, and every set in the set cover problem is a vertex in VCH.
- (ii) Indeed, the maximum rank of a hyperedge in VCH translates to the maximum frequency of an element in set cover and that the maximum degree in VCH translates to the maximum cardinality of a set in the set cover problem.

Also note that VCH is identical to the *hitting set* problem in set systems.

### 2.3 The Network

The network that corresponds to a hypergraph  $H = (V, E)$  is a bipartite graph  $N = (V \cup E, L)$ , where there is a processor for every vertex  $v$  and a processor for every hyperedge  $e$ . The set of links  $L$  consists of all the pairs  $(v, e) \in V \times E$  such that  $v \in e$ . Our algorithm does not require distinct IDs, namely, the network is anonymous. Moreover, the algorithm does not even rely on numbering of ports.

## 3 Algorithm Description

The algorithm is a primal-dual algorithm that updates the primal and dual variables in iterations. Each hyperedge has two variables: an auxiliary variable  $x(e)$  and an edge packing variable  $y(e)$ . We denote the value of the variables in iteration  $t$  by  $x_t(e)$  and  $y_t(e)$ .

- The dual variable  $y(e)$  is a nonnegative edge packing variable. By an edge packing variable we mean that, for every vertex  $v$  and in every iteration  $t$ ,  $\sum_{e \ni v} y_t(e) \leq 1$ . The variable  $y(e)$  is monotone non-decreasing over time.
- The auxiliary variable  $x(e)$  is initialized to  $x_0(t) = 1/K$ . The dynamics of  $x(e)$  allow to either divide or multiply  $x(e)$  by  $K$  in each iteration as long as it is bounded by  $1/K$ . Here,  $K \geq 2$  is a free parameter that is to be fixed later. The role of the auxiliary variables  $x(e)$  is to control the increase of the dual edge packing variables  $y(e)$ .

► **Definition 2.** A vertex  $v$  is  $\varepsilon$ -tight if  $\sum_{e \ni v} y_t(e) \geq 1 - \varepsilon$ .

Following the primal-dual approximation scheme, a vertex  $v$  joins the vertex cover as soon as it becomes  $\varepsilon$ -tight. The algorithm terminates when the set of  $\varepsilon$ -tight vertices covers all the hyperedges.

The following terminology is used in the algorithm and its analysis.

1. The set of edges that contain a vertex  $v$  is denoted by  $E(v)$ .
2. For a subset of edges  $A \subseteq E$ , let  $x_t[A] \triangleq \sum_{e \in A} x_t(e)$ .
3. For a vertex  $v$  let  $y_t[v] \triangleq \sum_{e \ni v} y_t(e)$ .

► **Definition 3.** The *effective degree* of an edge  $e$  is defined by

$$d_t(e) = \sum_{v \in e} x_t[E(v)].$$

Note that  $d_t(e) = \sum_{e': e' \cap e \neq \emptyset} |e \cap e'| \cdot x_t(e')$ . The “natural” definition of effective degree  $d_t(e) = \sum_{e': e' \cap e \neq \emptyset} x_t(e')$  works as well. However, it is not clear how to implement the natural definition in CONGEST.

► **Definition 4.** An edge  $e$  is *light* (in iteration  $t$ ) if  $d_t(e) < K$ . If  $d_t(e) \geq K$ , we say that the edge is *heavy*.

### 3.1 The Algorithm (ALG)

---

**Input:** Hypergraph  $H = (V, E)$  and  $0 < \epsilon < 1$ .

**Output:** A vertex cover  $C \subseteq V$ .

**Initialization:** For every  $e \in E$ ,  $x_0(e) \leftarrow 1/K$ ,  $y_0(e) \leftarrow 0$ ,  $C \leftarrow \emptyset$ ,  $E' \leftarrow E$ .

**Invariants:** (1) The variables  $y(e)$  constitute a feasible edge packing. (2)  $C$  equals the set of  $\epsilon$ -tight vertices.

**ALG:** The algorithm works by iterations until  $E' = \emptyset$ . Iteration  $t$  works as follows:

1. For each light edge  $e \in E'$ , set  $y_{t+1}(e) \leftarrow y_t(e) + x_t(e) \cdot \epsilon/K$ .
2. Add all the new  $\epsilon$ -tight vertices to  $C$ .
3. Remove covered edges:  $E' \leftarrow E' \setminus \{e \in E : e \cap C \neq \emptyset\}$ .
4. Update the auxiliary variables of edge  $e \in E'$ , as follows:

$$x_{t+1}(e) = \begin{cases} x_t(e)/K, & \text{if } d_t(e) \geq K \quad // \text{heavy edge rule} \\ \min\{K \cdot x_t(e), 1/K\}, & \text{if } d_t(e) < K \quad // \text{light edge rule.} \end{cases}$$


---

The following simple observation bounds  $d_t(e)$  for every edge  $e$  and iteration  $t$ .

► **Observation 5.** For all  $e \in E$ , and for all iterations  $t$  it holds that

$$d_t(e) \leq \frac{f\Delta}{K}, \text{ and} \tag{1}$$

$$\frac{d_t(e)}{K} \leq d_{t+1}(e) \leq K \cdot d_t(e). \tag{2}$$

## 4 Analysis

The analysis consists of two parts. In the first part, we prove that if the algorithm terminates, then it finds a vertex cover that is a  $(1 + O(\epsilon)) \cdot f$ -approximation of a minimum cardinality vertex cover. In the second part, we prove an upper bound on the number iterations of the algorithm. Every iteration requires a constant number of communication rounds, and hence the bound on the number of communication rounds follows.

### 4.1 Approximation Ratio

► **Claim 6.** Throughout the algorithm, the variables  $y_t(e)$  constitute a feasible edge packing.

**Proof.** Fix a vertex  $v$ . The proof is by induction on  $t$ . Initially,  $y_0(e) = 0$ , hence,  $y_0$  is clearly a feasible edge packing. Assume that  $\{y_t(e)\}_e$  is an edge packing (i.e.,  $y_t[v] \leq 1$ , for every  $v$ ), we now prove that  $\{y_{t+1}(e)\}_e$  is an edge packing. If  $y_{t+1}[v] > y_t[v]$ , then  $v$  is not  $\epsilon$ -tight in the end of iteration  $t$ , and thus  $y_t[v] < 1 - \epsilon$ .

Let  $e^*$  denote an arbitrary edge such that  $v \in e^*$  and  $y_{t+1}(e^*) > y_t(e^*)$ . In particular, this implies that  $e^*$  is light (see Step 1 of the algorithm), i.e.,  $d_t(e^*) < K$ .

We conclude that

$$\begin{aligned} y_{t+1}[v] - y_t[v] &= \frac{\epsilon}{K} \cdot \sum_{e \ni v, e \text{ light}} x_t(e) \\ &\leq \frac{\epsilon}{K} \cdot d_t(e^*) < \epsilon, \end{aligned}$$

where the second inequality holds because every light edge  $e$  that contains  $v$  contributes at least  $x_t(e)$  to  $d_t(e^*)$ . Since  $y_t[v] < 1 - \epsilon$ , the claim follows. ◀

► **Claim 7.** *At the end of every iteration  $t$  of the algorithm, the cardinality of the set of  $\varepsilon$ -tight vertices is at most  $\frac{f}{1-\varepsilon} \cdot \text{opt}$ .*

**Proof.**

$$\begin{aligned} |\{v \mid y_t[v] \geq 1 - \varepsilon\}| &\leq \sum_{v \mid y_t[v] \geq 1 - \varepsilon} \frac{1}{1 - \varepsilon} \sum_{e \ni v} y_t(e) \\ &\leq \frac{1}{1 - \varepsilon} \sum_{e \in E} \sum_{v \in e} y_t(e) \\ &\leq \frac{f}{1 - \varepsilon} \sum_{e \in E} y_t(e) \leq \frac{f}{1 - \varepsilon} \cdot \text{opt}, \end{aligned}$$

where the first inequality follows from the definition of  $\varepsilon$ -tight vertices, the third inequality follows from the fact that  $|e| \leq f$ , and the fourth inequality follows from weak. ◀

Note that throughout the algorithm,  $C$  is the set of  $\varepsilon$ -tight vertices. Upon termination,  $E' = \emptyset$ , and thus  $C$  is a vertex cover. Hence, by Claim 7, it follows that when the algorithm terminates, the set  $C$  is vertex cover and its cardinality is  $(1 + O(\varepsilon)) \cdot f \cdot \text{opt}$ .

## 4.2 Bounding the Number of Rounds

In this section we prove the following theorem. Recall that the algorithm terminates when  $E' = \emptyset$ .

► **Theorem 8.** *Let  $K \geq 2$ , the algorithm terminates after  $O\left(\frac{\log(f\Delta)}{\log K} + \frac{K^3}{\varepsilon}\right)$  iterations.*

### 4.2.1 Golden Iterations

Let  $Light_t \triangleq \{e \in E \mid d_t(e) < K\}$ , and  $Heavy_t \triangleq \{e \in E \mid d_t(e) \geq K\}$ .

► **Definition 9.** An iteration  $t$  is a Type-1 iteration with respect to hyperedge  $e$  if it satisfies:

$$d_t(e) < K \quad \text{and} \quad x_t(e) = 1/K.$$

► **Definition 10.** An iteration  $t$  is a Type-2 iteration with respect to hyperedge  $e$  if it satisfies:

$$d_t(e) \geq 1 \quad \text{and} \quad \sum_{v \in e} x_t[E(v) \cap Light_t] \geq \frac{1}{2K^2} \cdot d_t(e).$$

An iteration  $t$  is a *golden* iteration with respect to  $e$  if it is a Type-1 or Type-2 iteration with respect to  $e$ .

Our goal is to bound the number of iterations until termination. Throughout the analysis, fix a hyperedge  $e$ , and assume that it is not covered after  $T$  iterations (i.e.,  $e \cap C = \emptyset$ ).

► **Definition 11.** For a fixed hyperedge  $e$  not covered after  $T$  iterations, define the following

subsets of iterations.

$$\begin{aligned}
G_1 &\triangleq \left\{ t \in [T] \mid d_t(e) < K \text{ and } x_t(e) = \frac{1}{K} \right\} && \text{Type-1} \\
G_2 &\triangleq \left\{ t \in [T] \mid d_t(e) \geq 1 \text{ and } \sum_{v \in e} x_t[E(v) \cap \text{Light}_t] \geq \frac{1}{2K^2} \cdot d_t(e) \right\} && \text{Type-2} \\
H &\triangleq \{ t \in [T] \mid d_t(e) \geq K \} && \text{Heavy} \\
L &\triangleq \{ t \in [T] \mid d_t(e) < K \} && \text{Light} \\
U &\triangleq \{ t \in [T] \mid x_{t+1}(e) = K \cdot x_t(e) \} && \text{Up} \\
S &\triangleq \left\{ t \in [T] \mid x_t(e) = \frac{1}{K} \right\} && \text{Saturated}
\end{aligned}$$

### 4.2.2 Useful Claims

We denote the cardinalities of these subsets using lower case letters, e.g.,  $g_1 = |G_1|$ ,  $h = |H|$ , etc.

► **Claim 12.**  $H = \{ t \in [T] \mid x_{t+1}(e) = x_t(e)/K \}$  and  $u \leq h$ .

**Proof.** The first part follows from Line 4 of the algorithm. The variable  $x_0(e)$  is initialized to  $1/K$ , never exceeds  $1/K$ , is divided by  $K$  in iterations in  $H$ , and multiplied by  $K$  in iterations in  $U$ . Hence,  $1/K \geq x_T(e) = x_0(e) \cdot K^{u-h}$ , and  $u \leq h$ , as required. ◀

► **Claim 13.**  $T \leq 3h + g_1$ .

**Proof.** Note that  $\ell \leq u + s$ . Indeed, If  $t \in L$ , then either  $t \in U$  or  $x_t(e)$  could not be multiplied by  $K$ , hence  $t \in S$ . Since  $T = h + \ell$ , by Claim 12 we conclude that  $T \leq h + u + s \leq 2h + s$ .

To conclude the proof, we show that  $s \leq g_1 + h$ . This holds simply because,  $S \setminus G_1 \subseteq H$ . ◀

► **Claim 14.**  $\max\{g_1, g_2\} \leq \frac{2K^3}{\varepsilon}$ .

**Proof.** For each Type-1 iteration  $t \in [T]$ , the update of  $y_t(e)$  due to Steps 1 and 4 is

$$y_{t+1}(e) = y_t(e) + x_t(e) \cdot \frac{\varepsilon}{K} = y_t(e) + \frac{\varepsilon}{K} \cdot \frac{1}{K}.$$

Hence  $y_{T+1}(e) \geq g_1 \cdot \frac{\varepsilon}{K^2}$ . Claim 6 implies that the  $y_{T+1}(e')$  variables constitute a feasible edge packing, i.e.,  $y_{T+1}[v] \leq 1$  for every  $v$ , then  $y_{T+1}(e) \leq 1$ , and hence  $g_1 \leq K^2/\varepsilon$ , as required.

We bound  $g_2$  as follows. Consider a Type-2 iteration  $t \in [T]$ . Then,

$$\begin{aligned}
\sum_{v \in e} y_{t+1}[v] - y_t[v] &= \sum_{v \in e} \left( \sum_{e' \ni v, e' \in \text{Light}_t} \frac{\varepsilon}{K} \cdot x_t(e') \right) \\
&= \frac{\varepsilon}{K} \cdot \sum_{v \in e} x_t[E(v) \cap \text{Light}_t] \\
&\geq \frac{\varepsilon}{K} \cdot \frac{1}{2K^2} \cdot d_t(e) \geq \frac{\varepsilon}{2K^3},
\end{aligned}$$

where the last two inequalities follow from the definition of a Type-2 golden round (see Definition 10). This implies that  $g_2(e) \leq 2K^3/\varepsilon$ , as required. ◀

## 22:10 Distributed Set Cover Approximation: Primal-Dual with Optimal Locality

► **Claim 15.** *If  $d_t(e) \geq 1$ , and  $t \notin G_2$ , then*

$$d_{t+1}(e) < \frac{3}{2K} \cdot d_t(e). \quad (3)$$

**Proof.** If  $d_t(e) \geq 1$  and  $t$  is not a Type-2 iteration with respect to  $e$ , then  $\sum_{v \in e} x_t[E(v) \cap \text{Light}_t] < \frac{1}{2K^2} \cdot d_t(e)$ . Since  $x_{t+1}(e) \leq K \cdot x_t(e)$  if  $e \in \text{Light}_t$ , and  $x_{t+1}(e) = x_t(e)/K$  if  $e \in \text{Heavy}_t$ , we conclude that

$$\begin{aligned} d_{t+1}(e) &\leq \frac{1}{K} \cdot \sum_{v \in e} x_t[E(v) \cap \text{Heavy}_t] + K \cdot \sum_{v \in e} x_t[E(v) \cap \text{Light}_t] \\ &\leq \frac{1}{K} \cdot d_t(e) + K \cdot \frac{1}{2K^2} \cdot d_t(e). \end{aligned}$$

The claim follows. ◀

► **Claim 16.**  $h \leq \frac{\log(f\Delta/k^2)}{\log(\frac{2K}{3})} + 4g_2$ .

**Proof.** Partition  $H$  into maximally contiguous (disjoint) intervals  $H = H_1 \cup \dots \cup H_z$ . Denote the endpoints of  $H_i$  by  $[t_i, b_i]$ . Define

$$a_i \triangleq \begin{cases} t_1 & \text{if } i = 1 \\ \min\{t < t_i \mid \forall r \in [t, t_i - 1] : 1 \leq d_r(e) < K\} & \text{if } z \geq i > 1. \end{cases}$$

Note that, if  $i > 1$ , then the set  $\{t < t_i \mid \forall r \in [t, t_i - 1] : 1 \leq d_r(e) < K\}$  is not empty. Indeed,  $t_i - 1$  belongs to this set as  $d_{t_i}(e) \geq K$  and  $1 \leq d_{t_i-1}(e) < K$ .

Let  $I_i \triangleq [a_i, b_i]$ . Note that the intervals  $\{I_i\}_{i=1}^z$  are pairwise disjoint.

Since  $x_0(e) = \frac{1}{K}$  for every  $e \in E$  and since  $\sum_{v \in e} |E(v)| \leq f \cdot \Delta$  we get that  $d_{a_i}(e) \leq f\Delta/K$ . Hence, by the definition of  $a_i$ , we have

$$d_{a_i}(e) \leq \begin{cases} f\Delta/K & \text{if } i = 1 \\ K & \text{if } i > 1 \end{cases}$$

By the definition of  $b_i$  we have

$$d_{b_i}(e) \geq K.$$

Now,

$$\begin{aligned} d_{b_i}(e) &\leq d_{a_i}(e) \cdot \left(\frac{3}{2K}\right)^{|I_i \cap \bar{G}_2|} \cdot K^{|I_i \cap G_2|} \\ &\leq d_{a_i}(e) \cdot \left(\frac{2K}{3}\right)^{3 \cdot |I_i \cap G_2| - |I_i \cap \bar{G}_2|}. \end{aligned}$$

The first inequality follows from Claims 5 and 15. The second inequality follows from  $K < (2K/3)^3$ , as  $K \geq 2$ . Hence,

$$|I_i \cap \bar{G}_2| \leq 3 \cdot |I_i \cap G_2| + \frac{\log\left(\frac{d_{a_i}(e)}{d_{b_i}(e)}\right)}{\log(2K/3)}$$

Since

$$\frac{d_{a_i}(e)}{d_{b_i}(e)} \leq \begin{cases} f\Delta/K^2 & \text{if } i = 1 \\ 1 & \text{if } i > 1 \end{cases},$$

by summing up over all the disjoint intervals we obtain

$$\sum_{i=1}^z |I_i \cap \overline{G}_2| \leq 3g_2 + \frac{\log(f\Delta/K^2)}{\log(2K/3)}.$$

Since  $h \leq g_2 + \sum_{i=1}^z |I_i \cap \overline{G}_2|$ , the claim follows.  $\blacktriangleleft$

### 4.2.3 Proof of Theorem 8

**Proof of Theorem 8.** Suppose that the algorithm does not terminate after  $T$  rounds because the edge  $e$  remains uncovered. Claims 13, 16, and 14 and the fact that  $K \geq 2$  and  $0 < \varepsilon < 1$  imply that

$$\begin{aligned} T &\leq 3h + g_1 \\ &\leq 3 \left( \frac{\log(f\Delta/K^2)}{\log(\frac{2K}{3})} + 4g_2 \right) + g_1 \\ &= 3 \cdot \frac{\log(f\Delta/K^2)}{\log(\frac{2K}{3})} + 12g_2 + g_1 \\ &\leq 3 \cdot \frac{\log(f\Delta/K^2)}{\log(\frac{2K}{3})} + \frac{26K^3}{\varepsilon}. \end{aligned}$$

Since  $\log(2K/3) = \Omega(\log K)$ , the theorem follows.  $\blacktriangleleft$

## 5 Distributed Implementation

In this section we present a distributed implementation of the algorithm. To simplify the presentation, we present the sequence of computations and messages performed by the vertices and the edges in a combined fashion.

**States.** Every vertex  $v$  has three states: “active” - means that  $v$  did not decide yet if it is in the cover or not, “in cover” - means that  $v$  decided to join the cover, “not in cover” - means that  $v$  decided that it will not join the cover. Every edge  $e$  has two states: “uncovered” and “covered”.

### Distributed Implementation.

1. Every edge processor  $e$  maintains the variables  $x(e)$  and  $y(e)$ . These variables are initialized as follows:  $x(e) \leftarrow 1/K$  and  $y(e) \leftarrow 0$ . The initial state of  $e$  is “uncovered”.
2. Every vertex processor  $v$  maintains a variable  $E'(v) \subseteq E(v)$ , where  $E'(v)$  denotes the subset of edges that are not covered yet. Initialize  $E'(v) \leftarrow E(v)$ . The initial state of a vertex is “active”.
3. Each iteration consists of the following steps:
  - a. For every uncovered  $e$ , send  $x(e)$  and  $y(e)$  to every  $v \in e$ .
  - b. For every active  $v$ , if  $\sum_{e \ni v} y(e) \geq 1 - \varepsilon$ , then  $v$  changes its state of  $v$  to “in cover” and sends every edge  $e \in E(v)$  a message “in cover”.<sup>4</sup>
  - c. For every active  $v$ , send  $x[E(v)] = \sum_{e \in v} x(e)$  to every edge  $e' \in E(v)$ .

<sup>4</sup> The value used for  $y(e)$  is the last value received from  $e$ . If  $e$  is uncovered, then it sends  $y(e)$  in the previous round. If  $e$  is covered, then  $v$  remembers the last received value.

- d. For every edge  $e$ , if  $e$  received an “in cover” message, then  $e$  changes its state to “covered”, and sends a “covered” message to every  $v \in e$ .
- e. For every active vertex  $v$ , if  $v$  received a “covered” message from  $e$ , then deletes  $e$  from  $E'(v)$ . If  $E'(v) = \emptyset$ , then  $v$  changes its state to “not in cover”.<sup>5</sup>
- f. For every uncovered edge  $e$ , let  $d(e) = \sum_{v \in e} x[E(v)]$ . Update  $x(e)$  as follows:

$$x(e) \leftarrow \begin{cases} x(e)/K, & \text{if } d(e) \geq K \\ \min\{Kx(e), 1/K\}, & \text{if } d(e) < K \end{cases}$$

The algorithm terminates when all the edges are covered and all the vertices are not active.

**Bound on Message Length.** The messages in the algorithm are  $x(e), y(e), x[E(v)]$  and information about the state. Our goal is to bound the length of these messages.

- **Observation 17.** For every edge  $e$  and iteration  $t$ ,  $\frac{1}{K^t} \leq x_t(e) \leq \frac{1}{K}$ .
- **Observation 18.** For every vertex  $v$  and iteration  $t$ ,  $\frac{1}{K^t} \leq x_t[E(v)] \leq \frac{\Delta}{K}$ .
- **Observation 19.** For every edge  $e$  and iteration  $t$ , if  $y_t(e) > 0$ , then  $\frac{\varepsilon}{K} \cdot \frac{1}{K^t} \leq y_t(e) \leq \frac{\varepsilon}{K} \cdot \frac{t}{K}$ .

The following lemma is implied by the observations above and by the fact that the number of bits required for encoding the numbers in  $[a, b]$  where consecutive numbers differ by  $1/K$  is  $\log(bK/a)$ .

► **Lemma 20.** Let  $T$  denote the number of rounds of the algorithm until it terminates. Then the message length of the vertex cover algorithm is  $O(\log \Delta + T \cdot \log K)$ .

## 6 Proof of the Main Result

**Proof of Theorem 1.** Setting  $K = \sqrt[3]{\frac{\log(f\Delta)}{\log \log(f\Delta)}}$  in Theorem 8, implies that the round complexity of the algorithm is  $O\left(\frac{\log(f\Delta)}{\varepsilon \log \log(f\Delta)}\right)$ .

Claim 7 implies that the set  $C$  computed by our algorithm is indeed a vertex cover, and that this cover is an  $f(1 + O(\varepsilon))$ -approximate solution.

Lemma 20 implies that the message length of our algorithm is  $O(\varepsilon^{-1} \cdot \log(f\Delta))$ , as required. ◀

## 7 Discussion

In this paper we prove that an approximation of the minimum set cover (or the equivalent vertex cover in hypergraphs) can be computed in CONGEST in a locality-optimal way of performing the primal-dual scheme. The attained approximation ratio and number of rounds are  $f(1 + \varepsilon)$  and  $O\left(\frac{\log(f\Delta)}{\varepsilon \log \log(f\Delta)}\right)$  respectively, where  $\varepsilon$  is a constant in  $(0, 1)$ . Hence, for  $f \leq \text{poly}(\log \Delta)$  the round complexity matches the lower bound of  $\Omega\left(\frac{\log \Delta}{\log \log \Delta}\right)$  by Kuhn, Moscibroda, and Wattenhofer [19].

<sup>5</sup> In fact,  $v$  only needs to count the number of received “covered” messages. Hence, IDs and port numbers are not required.



The updates of the dual set variables are governed by the effective degrees of its elements  $e$ , the natural definition of which is (roughly) the summation over the elements which share a set with  $e$ . Unfortunately, it is not clear how to implement this natural definition in CONGEST. A nice observation is that the analysis also works with an approximated definition of the effective degree above (e.g., it allows double counting of elements) which is implementable in CONGEST. Another outcome of this relaxed definition of the effective degree is that our algorithm does not require distinct IDs, namely, the network is anonymous. Moreover, the algorithm does not even rely on numbering of ports. We are hopeful that dynamics of the same style may lead to improvements for other optimization problems.

---

## References

- 1 Matti Åstrand and Jukka Suomela. Fast distributed approximation algorithms for vertex cover and set cover in anonymous networks. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, pages 294–302. ACM, 2010.
- 2 Reuven Bar-Yehuda, Keren Censor-Hillel, Mohsen Ghaffari, and Gregory Schwartzman. Distributed approximation of maximum independent set and maximum matching. In *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017, Washington, DC, USA, July 25-27, 2017*, pages 165–174, 2017. doi:10.1145/3087801.3087806.
- 3 Reuven Bar-Yehuda, Keren Censor-Hillel, and Gregory Schwartzman. A Distributed  $(2+\epsilon)$ -Approximation for Vertex Cover in  $O(\log \Delta/\epsilon \log \log \Delta)$  Rounds. *J. ACM*, 64(3):23:1–23:11, 2017. doi:10.1145/3060294.
- 4 Reuven Bar-Yehuda and Shimon Even. A linear-time approximation algorithm for the weighted vertex cover problem. *Journal of Algorithms*, 2(2):198–203, 1981.
- 5 Reuven Bar-Yehuda and Shimon Even. *A local-ratio theorem for approximating the weighted vertex cover problem*. Technion-Israel Institute of Technology. Department of Computer Science, 1983.
- 6 Reuven Bar-Yehuda and Dror Rawitz. On the equivalence between the primal-dual schema and the local ratio technique. *SIAM Journal on Discrete Mathematics*, 19(3):762–797, 2005.
- 7 R. Ben-Basat, G. Even, K. Kawarabayashi, and G. Schwartzman. A Deterministic Distributed 2-Approximation for Weighted Vertex Cover in  $O(\log n \log \Delta/\log^2 \log \Delta)$  Rounds. *ArXiv e-prints (Appeared in SIROCCO 2018)*, 2018. arXiv:1804.01308.
- 8 Irit Dinur, Venkatesan Guruswami, Subhash Khot, and Oded Regev. A new multilayered pcp and the hardness of hypergraph vertex cover. *SIAM Journal on Computing*, 34(5):1129–1146, 2005.
- 9 Irit Dinur and Samuel Safra. On the hardness of approximating minimum vertex cover. *Annals of mathematics*, pages 439–485, 2005.
- 10 Uriel Feige. A threshold of  $\ln n$  for approximating set cover. *Journal of the ACM (JACM)*, 45(4):634–652, 1998.
- 11 Mohsen Ghaffari. An improved distributed algorithm for maximal independent set. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*, pages 270–277, 2016. doi:10.1137/1.9781611974331.ch20.
- 12 Mohsen Ghaffari, David G Harris, and Fabian Kuhn. On derandomizing local distributed algorithms. *arXiv preprint arXiv:1711.02194*, 2017.
- 13 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*, pages 784–797. ACM, 2017.

- 14 Johan Håstad. Some optimal inapproximability results. *Journal of the ACM (JACM)*, 48(4):798–859, 2001.
- 15 Jonas Holmerin. Improved inapproximability results for vertex cover on  $k$ -uniform hypergraphs. In *International Colloquium on Automata, Languages, and Programming*, pages 1005–1016. Springer, 2002.
- 16 Subhash Khot and Oded Regev. Vertex cover might be hard to approximate to within  $2 - \epsilon$ . *Journal of Computer and System Sciences*, 74(3):335–349, 2008.
- 17 Christos Koufogiannakis and Neal E Young. Distributed algorithms for covering, packing and maximum weighted matching. *Distributed Computing*, 24(1):45–63, 2011.
- 18 Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. The price of being near-sighted. In *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 980–989. Society for Industrial and Applied Mathematics, 2006.
- 19 Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. Local computation: Lower and upper bounds. *J. ACM*, 63(2):17:1–17:44, 2016. doi:10.1145/2742012.
- 20 Nathan Linial. Distributive graph algorithms global solutions from local data. In *Foundations of Computer Science, 1987., 28th Annual Symposium on*, pages 331–335. IEEE, 1987.
- 21 Carsten Lund and Mihalis Yannakakis. On the hardness of approximating minimization problems. *Journal of the ACM (JACM)*, 41(5):960–981, 1994.
- 22 Dana Moshkovitz. The projection games conjecture and the np-hardness of  $\ln n$ -approximating set-cover. *Theory of Computing*, 11:221–235, 2015. doi:10.4086/toc.2015.v011a007.
- 23 David Peleg. *Distributed computing: a locality-sensitive approach*. SIAM, 2000.
- 24 Jukka Suomela. Survey of local algorithms. *ACM Computing Surveys (CSUR)*, 45(2):24, 2013.
- 25 Luca Trevisan. Non-approximability results for optimization problems on bounded degree instances. In *Proceedings of the thirty-third annual ACM symposium on Theory of computing*, pages 453–461. ACM, 2001.
- 26 Vijay V Vazirani. *Approximation algorithms*. Springer Science & Business Media, 2013.
- 27 David P Williamson and David B Shmoys. *The design of approximation algorithms*. Cambridge university press, 2011.

# Order out of Chaos: Proving Linearizability Using Local Views

**Yotam M. Y. Feldman**

Tel Aviv University, Israel

**Constantin Enea**

IRIF, Univ. Paris Diderot & CNRS, France

**Adam Morrison**

Tel Aviv University, Israel

**Noam Rinetzky**

Tel Aviv University, Israel

**Sharon Shoham**

Tel Aviv University, Israel

---

## Abstract

---

Proving the linearizability of highly concurrent data structures, such as those using optimistic concurrency control, is a challenging task. The main difficulty is in reasoning about the view of the memory obtained by the threads, because as they execute, threads observe different fragments of memory from different points in time. Until today, every linearizability proof has tackled this challenge from scratch.

We present a unifying proof argument for the correctness of unsynchronized traversals, and apply it to prove the linearizability of several highly concurrent search data structures, including an optimistic self-balancing binary search tree, the Lazy List and a lock-free skip list. Our framework harnesses *sequential reasoning* about the view of a thread, considering the thread as if it traverses the data structure without interference from other operations. Our key contribution is showing that properties of reachability along search paths can be deduced for concurrent traversals from such interference-free traversals, when certain intuitive conditions are met. Basing the correctness of traversals on such *local view arguments* greatly simplifies linearizability proofs. At the heart of our result lies a notion of *order on the memory*, corresponding to the order in which locations in memory are read by the threads, which guarantees a certain notion of consistency between the view of the thread and the actual memory.

To apply our framework, the user proves that the data structure satisfies two conditions: (1) acyclicity of the order on memory, even when it is considered across intermediate memory states, and (2) preservation of search paths to locations modified by interfering writes. Establishing the conditions, as well as the full linearizability proof utilizing our proof argument, reduces to simple concurrent reasoning. The result is a clear and comprehensible correctness proof, and elucidates common patterns underlying several existing data structures.

**2012 ACM Subject Classification** Computing methodologies → Shared memory algorithms, Theory of computation → Program verification

**Keywords and phrases** concurrency and synchronization, concurrent data structures, linearizability, optimistic concurrency control, verification and formal methods

**Digital Object Identifier** 10.4230/LIPIcs.DISC.2018.23

**Related Version** An extended version appears in [20], <https://arxiv.org/abs/1805.03992>.



© Yotam M. Y. Feldman, Constantin Enea, Adam Morrison, Noam Rinetzky, and Sharon Shoham; licensed under Creative Commons License CC-BY

32nd International Symposium on Distributed Computing (DISC 2018).

Editors: Ulrich Schmid and Josef Widder; Article No. 23; pp. 23:1–23:21

Leibniz International Proceedings in Informatics



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

**Acknowledgements** This publication is part of a project that has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agreements No [759102-SVIS] and [678177]). The research was partially supported by Len Blavatnik and the Blavatnik Family foundation, the Blavatnik Interdisciplinary Cyber Research Center, Tel Aviv University, the United States-Israel Binational Science Foundation (BSF) grants No. 2016260 and 2012259, and the Israeli Science Foundation (ISF) grant No. 2005/17. We thank the anonymous reviewers whose comments helped improve the paper.

## 1 Introduction

Concurrent data structures must minimize synchronization to obtain high performance [16, 27]. Many concurrent search data structures therefore use *optimistic* designs, which search the data structure without locking or otherwise writing to memory, and write to shared memory only when modifying the data structure. Thus, in these designs, operations that do not modify the same nodes do not synchronize with each other; in particular, searches can run in parallel, allowing for high performance and scalability. Optimistic designs are now common in concurrent search trees [3, 10, 11, 14, 17, 19, 28, 36, 41], skip lists [13, 21, 26], and lists/hash tables [22, 23, 35, 45].

A major challenge in developing an optimistic search data structure is proving *linearizability* [25], i.e., that every operation appears to take effect atomically at some point in time during its execution. Usually, the key difficulty is proving properties of unsynchronized searches [37, 32, 48, 27], as they can observe an *inconsistent* state of the data structure – for example, due to observing only some of the writes performed by an update operation, or only some update operations but not others. Arguing about such searches requires tricky *concurrent reasoning* about the possible interleaving of reads and writes of the operations. Today, every new linearizability proof tackles these problems from scratch, leading to long and complex proofs.

**Our approach: local view arguments.** This paper presents a unifying proof argument for proving linearizability of concurrent data structures with unsynchronized searches that replaces the difficult concurrent reasoning described above with *sequential reasoning* about a search, which does not consider interference from other operations. Our main contribution is a framework for establishing properties of an unsynchronized search in a concurrent execution by reasoning *only* about its *local view* – the (potentially inconsistent) picture of memory it observes as it traverses the data structure. We refer to such proofs as *local view arguments*. We show that under two (widely-applicable) conditions listed below, the existence of a path to the searched node in the local view, deduced with sequential reasoning, also holds at some point during the *actual* (concurrent) execution of the traversal. (This includes the case of non-existence of a key indicated by a path to `null`.) Such *reachability* properties are typically key to the linearizability proofs of many prominent concurrent search data structures with unsynchronized searches [16]. Once these properties are established, the rest of the linearizability proof requires only simple concurrent reasoning.

Applying a local view argument requires establishing two conditions:

- (i) *temporal acyclicity*, which states that the search follows an *order* on the memory that is *acyclic* across intermediate states throughout the concurrent execution; and
- (ii) *preservation*, which states that whenever a node  $x$  is changed, if it was on a search path for some key  $k$  in the past, then it is also on such a search path at the time of the change.

Although these conditions refer to concurrent executions, proving them for the data structures we consider is straightforward.

More generally, these conditions can be established with inductive proofs that are simplified by relying on the very same traversal properties obtained with the local view argument. This seemingly circular reasoning holds because our framework is also proven inductively, and so the case of executions of length  $N + 1$  in both the proof that (1) the data structure satisfies the conditions and (2) the traversal properties follow from the local view argument can rely on the correctness of the other proof's  $N$  case.

**Simplifying linearizability proofs with local view arguments.** To harness local view arguments, our approach uses *assertions* in the code as a way to divide the proof between (1) the linearizability proof that *relies on the assertions*, and (2) the proof of the assertions, where the challenge of establishing properties of unsynchronized searches in concurrent executions is overcome by local view arguments.

Overall, our proof argument yields clear and comprehensible linearizability proofs, whose whole is (in some sense) greater than the sum of the parts, since each of the parts requires a simpler form of reasoning compared to contemporary linearizability proofs. We use local view arguments to devise simple linearizability proofs of a variant of the contention-friendly tree [14] (a self-balancing search tree), lists with lazy [23] or non-blocking [27] synchronization, and a lock-free skip list.

Our framework's *acyclicity* and *preservation* conditions can provide insight on algorithm design, in that their proofs can reveal unnecessary protections against interference. Indeed, our proof attempts exposed (small) parts of the search tree algorithm that were not needed to guarantee linearizability, leading us to consider a simpler variant of its search operation (see Remark 1).

**Contributions.** To summarize, we make the following contributions:

1. We provide a set of conditions under which reachability properties of local views, established using sequential reasoning, hold also for concurrent executions,
2. We show that these conditions hold for non-trivial concurrent data structures that use unsynchronized searches, and
3. We demonstrate that the properties established using local view arguments enable simple linearizability proofs, alleviating the need to consider interleavings of reads and writes during searches.

## 2 Motivating Example

As a motivating example we consider a self-balancing binary search tree with optimistic, read-only searches. This is an example of a concurrent data structure for which it is challenging to prove linearizability “from scratch.” The algorithm is based on the contention-friendly (CF) tree [12, 14]. It is a fine-grained lock-based implementation of a set object with the standard `insert( $k$ )`, `delete( $k$ )`, and `contains( $k$ )` operations. The algorithm maintains an *internal* binary tree that stores a key in every node. Similarly to the lazy list [23], the algorithm distinguishes between the *logical deletion* of a key, which removes it from the set represented by the tree, and the *physical removal* that unlinks the node containing the key from the tree.

We use this algorithm as a running example to illustrate how our framework allows to lift sequential reasoning into assertions about concurrent executions, which are in turn used to prove linearizability. In this section, we present the algorithm and explain the linearizability proof based on the assertions, highlighting the significant role of local view arguments in the proof.

```

1 type N
2 int key
3 N left, right
4 bool del, rem

6 N root ← new N(∞);

8 NXN locate(int k)
9 x, y ← root
10 while (y ≠ null ∧ y.key ≠ k)
11 x ← y
12 if (x.key < k)
13 y ← x.right
14 else
15 y ← x.left
16 {⊠(root  $\overset{k}{\rightsquigarrow}$  x) ∧ ⊠(root  $\overset{k}{\rightsquigarrow}$  y)
   ∧ x.key ≠ k ∧ y ≠ null
   ⇒ y.key = k}
17 return (x, y)

19 bool contains(int k)
20 (_, y) ← locate(k)
21 if (y = null)
22 {⊠(root  $\overset{k}{\rightsquigarrow}$  null)}
23 return false
24 {⊠(root  $\overset{k}{\rightsquigarrow}$  y)}
25 if (y.del)
26 {⊠(root  $\overset{k}{\rightsquigarrow}$  y ∧ y.del) ∧ y.key = k}
27 return false
28 {⊠(root  $\overset{k}{\rightsquigarrow}$  y ∧ ¬y.del) ∧ y.key = k}
29 return true

30 bool delete(int k)
31 (_, y) ← locate(k)
32 if (y = null)
33 {⊠(root  $\overset{k}{\rightsquigarrow}$  null)}
34 return false
35 lock(y)
36 if (y.rem) restart
37 ret ← ¬y.del
38 {root  $\overset{k}{\rightsquigarrow}$  y ∧ y.key = k ∧ ¬y.rem}
39 y.del ← true
40 return ret

42 bool insert(int k)
43 (x, y) ← locate(k)
44 {⊠(root  $\overset{k}{\rightsquigarrow}$  x) ∧ x.key ≠ k}
45 if (y ≠ null)
46 {⊠(root  $\overset{k}{\rightsquigarrow}$  y) ∧ y.key = k}
47 lock(y)
48 if (y.rem) restart
49 ret ← y.del
50 {root  $\overset{k}{\rightsquigarrow}$  y ∧ y.key = k ∧ ¬y.rem}
51 y.del ← false
52 return ret
53 lock(x)
54 if (x.rem) restart
55 if (k < x.key ∧ x.left = null)
56 {root  $\overset{k}{\rightsquigarrow}$  x ∧ ¬x.rem
   ∧ k < x.key ∧ x.left = null}
57 x.left ← new N(k)
58 else if (x.right = null)
59 {root  $\overset{k}{\rightsquigarrow}$  x ∧ ¬x.rem
   ∧ k > x.key ∧ x.right = null}
60 x.right ← new N(k)
61 else
62 restart
63 return true

64 removeRight()
65 (z, _) ← locate(*)
66 lock(z)
67 y ← z.right
68 if (y = null ∨ z.rem)
69 return
70 lock(y)
71 if (y.del)
72 return
73 if (y.left = null)
74 z.right ← y.right
75 else
76 if (y.right = null)
77 z.right ← y.left
78 else return
79 y.rem ← true

81 rotateRightLeft()
82 (p, _) ← locate(*)
83 lock(p)
84 y ← p.left
85 if (y = null ∨ p.rem)
86 return
87 lock(y)
88 x ← y.left
89 if (x = null)
90 return
91 lock(x)
92 z ← duplicate(y)
93 z.left ← x.right
94 x.right ← z
95 p.left ← x
96 y.rem ← true

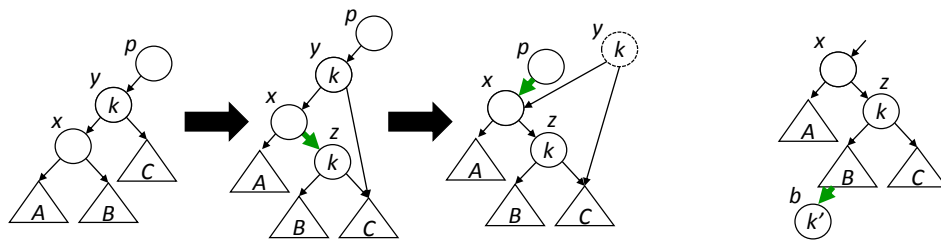
```

■ **Figure 1** Running example. For brevity, **unlock** operations are omitted; a procedure releases all the locks it acquired when it terminates or **restarts**. \* denotes an arbitrary key.

Figure 1 shows the code of the algorithm. (The code is annotated with assertions written inside curly braces, which the reader should ignore for now; we explain them in Section 2.1.) Nodes contain two boolean fields, *del* and *rem*, which indicate whether the node is logically deleted and physically removed, respectively. Modifications of a node in the tree are synchronized with the node’s lock. Every operation starts with a call to `locate(k)`, which performs a standard binary tree search – without acquiring any locks – to locate the node with the target key *k*. This method returns the last link it traverses,  $(x, y)$ . Thus, if *k* is found,  $y.key = k$ ; if *k* is not found,  $y = null$  and *x* is the node that would be *k*’s parent if *k* were inserted. A `delete(k)` logically deletes *y* after verifying that *y* remained linked to the tree after its lock was acquired. An `insert(k)` either revives a logically deleted node or, if *k* was not found, links a new node to the tree. A `contains(k)` returns *true* if it locates a node with key *k* that is not logically deleted, and *false* otherwise.

Physical removal of nodes and balancing of the tree’s height are performed using auxiliary methods.<sup>1</sup> The algorithm physically removes only nodes with at most one child. The `removeRight` method unlinks such a node that is a right child, and sets its *rem* field to notify

<sup>1</sup> The reader should assume that these methods can be invoked at any time; the details of when the algorithm decides to invoke them are not material for correctness. For example, in [12, 14], these methods are invoked by a dedicated restructuring thread.



(a) Right rotation of  $y$ . (The bold green link is the one written in each step. The node with a dashed border has its *rem* bit set.)

(b) Node  $b$  is added after the right rotation of  $y$ , when  $y$  is no longer in the tree.

■ **Figure 2** A right rotation, and how it can lead a search to observe an inconsistent state of the tree. If  $b$  is added after the rotation, a search for  $k'$  that starts before the rotation and pauses at  $x$  during the rotation will traverse the path  $p, y, x, z, \dots, b$ , although  $y$  and  $b$  never exist simultaneously in the tree.

threads that have reached the node of its removal. (We omit the symmetric `removeLeft`.) Balancing is done using rotations. Figure 2a depicts the operation of `rotateRightLeft`, which needs to rotate node  $y$  (with key  $k$ ) down. (We omit the symmetric operations.) It creates a new node  $z$  with the same key and *del* bit as  $y$  to take  $y$ 's place, leaving  $y$  unchanged except for having its *rem* bit set. A similar technique for rotations is used in lock-free search trees [10].

► **Remark 1.** The example of Figure 1 differs from the original contention-friendly tree [12, 14] in a few points. The most notable difference is that our traversals do not consult the `rem` flag, and in particular we do not need to distinguish between a left and right rotate, making the traversals' logic simpler. Checking the `rem` flag is in fact unnecessary for obtaining linearizability, but it allows proving linearizability with a *fixed* linearization point, whereas proving the correctness of the algorithm without this check requires an *unfixed* linearization point. For our framework, the necessity to use an unfixed linearization point incurs no additional complexity. In fact, the simplicity of our proof method allowed us to spot this “optimization.” In addition, the original algorithm performs backtracking by setting pointers from child to parent when nodes are removed. Instead, we restart the operation; see Section 7 for a discussion of backtracking. Lastly, we fix a minor omission in the description of [14], where the `del` field was not copied from a rotated node.

## 2.1 Proving Linearizability

Proving linearizability of an algorithm like ours is challenging because searches are performed with no synchronization. This means that, due to interference from concurrent updates, searches may observe an inconsistent state of the tree that has not existed at any point in time. (See Figure 2.) In our example, while it is easy to see that `locate` in Figure 1 constructs a search path to a node in sequential executions, what this implies for concurrent traversals is not immediately apparent. Proving properties of the traversal – in particular, that a node reached in the traversal truly lies on a search path for key  $k$  – is instrumental for the linearizability proof [48, 37].

Generally, our linearizability proofs consist of two parts: (1) proving a set of *assertions* in the code of the concurrent data structure, and (2) a proof of linearizability based on those assertions. The most difficult part and the main focus of our paper is proving the assertions



using local view arguments, discussed in Section 2.2. In the remaining of this section we demonstrate that having assertions about the actual state during the concurrent execution makes it a straightforward exercise to verify that the algorithm in Figure 1 is a linearizable implementation of a set, *assuming these assertions*.

Consider the assertions in Figure 1. An assertion  $\{\mathbb{P}\}$  means that  $\mathbb{P}$  holds now (i.e., in any state in which the next line of code executes). An assertion of the form  $\{\diamond\mathbb{P}\}$  means that  $\mathbb{P}$  was true at some point between the invocation of the operation and now. The assertions contain predicates about the state of locked nodes, immutable fields, and predicates of the form  $\text{root} \stackrel{k}{\rightsquigarrow} x$ , which means that  $x$  resides on a *valid search path* for key  $k$  that starts at  $\text{root}$ ; if  $x = \text{null}$  this indicates that  $k$  is not in the tree (because a valid search path to  $k$  does not continue past a node with key  $k$ ). Formally, search paths between objects (representing nodes in the tree) are defined as follows:

$$o_r \stackrel{k}{\rightsquigarrow} o_x \stackrel{\text{def}}{=} \exists o_0, \dots, o_m. o_0 = o_r \wedge o_m = o_x \wedge \forall i = 1..m. \text{nextChild}(o_{i-1}, k, o_i), \text{ and} \\ \text{nextChild}(o_{i-1}, k, o_i) = (o_{i-1}.key > k \wedge o_{i-1}.left = o_i) \vee (o_{i-1}.key < k \wedge o_{i-1}.right = o_i) .$$

One can prove linearizability from these assertions by, for example, using an *abstraction function*  $\mathcal{A} : H \rightarrow \wp(\mathbb{N})$  that maps a concrete memory state<sup>2</sup> of the tree,  $H$ , to the *abstract set* represented by this state, and showing that `contains`, `insert`, and `delete` manipulate this abstraction according to their specification. We define  $\mathcal{A}$  to map  $H$  to the set of keys of the nodes that are on a valid search path for their key and are not logically deleted in  $H$ :  $\mathcal{A}(H) = \{k \in \mathbb{N} \mid H \models \exists x. \text{root} \stackrel{k}{\rightsquigarrow} x \wedge x.key = k \wedge \neg x.del\}$ . ( $H \models P$  means that  $P$  is true in  $H$ .)

The assertions almost immediately imply that for every operation invocation  $op$ , there exists a state  $H$  during  $op$ 's execution for which the abstract state  $\mathcal{A}(H)$  agrees with  $op$ 's return value, and so  $op$  can be linearized at  $H$ . We provide a more detailed discussion in the extended version [20].

## 2.2 Proving the Assertions

To complete the linearizability proof, it remains to prove the validity of the assertions in concurrent executions. The most challenging assertions to prove are those concerning properties of unsynchronized traversals, which we target in this paper. In Section 3 we present our framework, which allows to deduce assertions of the form of  $\diamond(\text{root} \stackrel{k}{\rightsquigarrow} x)$  at the end of (concurrent) traversals by considering only interference-free executions. We apply our framework to establish the assertions  $\diamond(\text{root} \stackrel{k}{\rightsquigarrow} x)$  and  $\diamond(\text{root} \stackrel{k}{\rightsquigarrow} y)$  in line 16. In fact, our framework allows to deduce slightly stronger properties, namely, of the form  $\diamond(\text{root} \stackrel{k}{\rightsquigarrow} x \wedge \varphi(x))$ , where  $\varphi(x)$  is a property of a single field of  $x$  (see Remark 2). This is used to prove the assertions  $\diamond(\text{root} \stackrel{k}{\rightsquigarrow} y \wedge y.del)$  in line 26 and similarly in line 28. For completeness, we now show how the proof of the remaining assertions in Figure 1 is attained, when assuming the assertions deduced by the framework. This concludes the linearizability proof.

**Reachability related assertions.** In line 24 the fact that  $\diamond(\text{root} \stackrel{k}{\rightsquigarrow} y)$  is true follows from line 16.

<sup>2</sup> We use standard modeling of the memory state (the *heap*) as a function  $H$  from locations to values; see Section 3.



The writes in `insert` and `delete` (lines 50, 56, 59 and 38) require that a path exists *now*. This follows from the  $\diamond(\text{root} \overset{k}{\rightsquigarrow} x)$  (known from the local view argument) and the fact that  $\neg x.\text{rem}$ , using an invariant similar to preservation (see Example 10): For every location  $x$  and key  $k$ , if  $\text{root} \overset{k}{\rightsquigarrow} x$ , then every write retains this unless it sets  $x.\text{rem}$  before releasing the lock on  $x$  (this happens in lines 95, 77 and 74). Thus, when `insert` and `remove` lock  $x$  and see that it is not marked as removed,  $\text{root} \overset{k}{\rightsquigarrow} x$  follows from  $\diamond(\text{root} \overset{k}{\rightsquigarrow} x)$ . Note that the fact that writes other than lines 95, 77 and 74 do not invalidate  $\text{root} \overset{k}{\rightsquigarrow} x$  follows easily from their annotations.

**Additional assertions.** The invariant that keys are immutable justifies assertions referring to keys of objects that are read earlier, e.g. in line 50 and the rest of the assertion in line 28 ( $y.\text{key}$  is read earlier in `locate`). The rest of the assertions can be attributed to reading a location under the protection of a lock. An example of this is the assertion that  $\neg y.\text{rem}$  in line 38.

### 3 The Framework: Correctness of Traversals Using Local Views

In this section we present the key technical contribution of our framework, which targets proving properties of traversals. We address properties of *reachability along search paths* (formally defined in Section 3.1). Roughly speaking, our approach considers the traversal in concurrent executions as operating without interference on a *local view*: the thread's potentially inconsistent picture of memory obtained by performing reads concurrently with writes by other threads. For a property  $\mathbb{S}_{k,x} = \text{root} \overset{k}{\rightsquigarrow} x$  of reachability along a search path, we introduce conditions under which one can deduce that  $\diamond\mathbb{S}_{k,x}$  holds in the actual global state of the concurrent data structure out of the fact that  $\mathbb{S}_{k,x}$  holds in the local view of a single thread, where the latter is established using sequential reasoning (see Section 3.2). This alleviates the need to reason about intermediate states of the traversal in the concurrent proof.

This section is organized as follows: We start with some preliminary definitions. Section 3.1 defines the abstract, general notion of search paths our framework treats. Section 3.2 defines the notion of a local view which is at the basis of local view arguments. Section 3.3 formally defines the conditions under which local view arguments hold, and states our main technical result. In Section 3.4 we sketch the ideas behind the proof of this result.

**Programming model.** A *global state* (state) is a mapping between *memory locations* (locations) and *values*. A value is either a natural number, a location, or *null*. Without loss of generality, we assume that threads share access to a global state. Thus, memory locations are used to store the values of fields of objects. A *concurrent execution* (execution) is a sequence of states produced by an interleaving of atomic actions issued by threads. We assume that each atomic action is either a *read* or a *write* operation. (We treat synchronization actions, e.g., *lock* and *unlock*, as writes.) A *read*  $r$  consists of a value  $v$  and a location  $\text{read}(r)$  with the meaning that  $r$  reads  $v$  from  $\text{read}(r)$ . Similarly, a *write*  $w$  consists of a value  $v$  and a location  $\text{mod}(w)$  with the meaning that  $w$  sets  $\text{mod}(w)$  to  $v$ . We denote by  $w(H)$  the state resulting from the execution of  $w$  on state  $H$ .

### 3.1 Reachability Along Search Paths

The properties we consider are given by predicates of the form  $\mathbb{S}_{k,x} = \mathbf{root} \xrightarrow{k} x$ , denoting reachability of  $x$  by a  $k$ -search path, where  $\mathbf{root}$  is the entry point to the data structure. A  $k$ -search path in state  $H$  is a sequence of locations that is traversed when searching for a certain element, parametrized by  $k$ , in the data structure. Reachability of an *object*  $x$  along a  $k$ -search path from  $\mathbf{root}$  is understood as the existence of a  $k$ -search path between designated locations of  $x$ , e.g. the key field, and  $\mathbf{root}$ .

Search paths may be defined differently in different data structures (e.g., list, tree or array). For example,  $k$ -search paths in Figure 1 consist of sequences  $\langle x.key, x.left, y.key \rangle$  where  $y.key$  is the address pointed to by  $x.left$  (meaning, the location that is the value stored in  $x.left$ ) and  $x.key > k$ , or  $\langle x.key, x.right, y.key \rangle$  where  $y.key$  is the address pointed to by  $x.right$  and  $x.key < k$ . This definition of  $k$ -search paths reproduces the definition of reachability along search paths from Section 2.1.

Our framework is oblivious to the specific definition of search paths, and only assumes the following properties of search paths (which are satisfied, for example, by the definition above):

- If  $\ell_1, \dots, \ell_m$  is a  $k$ -search path in  $H$  and  $H'$  satisfies  $H'(\ell_i) = H(\ell_i)$  for all  $1 \leq i < m$ , then  $\ell_1, \dots, \ell_m$  is a  $k$ -search path in  $H'$  as well, i.e., the search path depends on the values of locations in  $H$  only for the locations along the sequence itself (but the last).
- If  $\ell_1, \dots, \ell_m$  and  $\ell_m, \dots, \ell_{m+r}$  are both  $k$ -search paths in  $H$ , then  $\ell_1, \dots, \ell_m, \dots, \ell_{m+r}$  is too, i.e., search paths are closed under concatenation.
- If  $\ell_1, \dots, \ell_m$  is a  $k$ -search path in  $H$  then so is  $\ell_i, \dots, \ell_j$  for every  $1 \leq i \leq j \leq m$ , i.e., search paths are closed under truncation.

► **Remark 2.** It is simple to extend our framework to deduce properties of the form  $\diamond(\mathbf{root} \xrightarrow{k} x \wedge \varphi(x))$  where  $\varphi(x)$  is a property of a single field of  $x$ . For example,  $\varphi(x) = x.del$  states that the field *del* of  $x$  is true. As another example, the predicate  $\mathbf{root} \xrightarrow{k} x \wedge (x.next = y)$  says that the *link* from  $x$  to  $y$  is reachable. See the extended version [20] for details.

### 3.2 Local Views and Their Properties

We now formalize the notion of *local view* and explain how properties of local views can be established using sequential reasoning.

**Local view.** Let  $\bar{r} = r_1, \dots, r_d$  be a sequence of read actions executed by some thread. As opposed to the global state, the *local view* of the reading thread refers to the inconsistent picture of the memory state that the thread obtains after issuing  $\bar{r}$  (concurrently with writes). Formally, the sequence of reads  $\bar{r}$  induces a state  $H_{lv}$ , which is constructed by assigning to every location  $x$  which  $\bar{r}$  reads the last value  $\bar{r}$  reads in  $x$ . Namely, when  $\bar{r}$  starts, its local view  $H_{lv}^{(0)}$  is empty, and, assuming its  $i$ th read of value  $v$  from location  $\ell$ , the produced local view is  $H_{lv}^{(i)} = H_{lv}^{(i-1)}[\ell \mapsto v]$ . We refer to  $H_{lv} = H_{lv}^{(d)}$  as the *local view* produced by  $\bar{r}$  (*local view* for short). We emphasize that while technically  $H_{lv}$  is a state, it is *not* necessarily an actual intermediate global state, and may have never existed in memory during the execution.

**Sequential reasoning for establishing properties of local views.** Properties of the local view  $H_{lv}$ , which are the starting point for applying our framework, are established using *sequential* reasoning. Namely, proving that a predicate such as  $\mathbf{root} \xrightarrow{k} x$  holds in the local view at the end of the traversal amounts to proving that it holds in *any sequential* execution

of the traversal, i.e., an execution without interference which starts at an *arbitrary* program state. This is because the concurrent traversal constructing the local view can be understood as a sequential execution that starts with the local view as the program state.

► **Example 3.** In the running example, straightforward sequential reasoning shows that indeed  $\text{root} \stackrel{k}{\sim} x$  holds at line 16 in sequential executions of  $\text{locate}(k)$  (i.e., executions without interference), no matter at which program state the execution starts. This ensures that it holds, in particular, in the local view.

### 3.3 Local View Argument: Conditions & Guarantees

The main theorem underlying our framework bridges the discrepancy between the *local view* of a thread as it performs a sequence of read actions, and the actual *global state* during the traversal.

In the sequel, we fix a sequence of read actions  $\bar{r} = r_1, \dots, r_d$  executed by some thread, and denote the sequence of write actions executed concurrently with  $\bar{r}$  by  $\bar{w} = w_1, \dots, w_n$ . We denote the global state when  $\bar{r}$  starts its execution by  $H_c^{(0)}$ , and the intermediate global states obtained after each prefix of these writes in  $\bar{w}$  by  $H_c^{(i)} = w_1 \dots w_i(H_c^{(0)})$ .

Using the above terminology, our framework devises conditions for showing for a reachability property  $\mathbb{S}_{k,x}$  that if  $\mathbb{S}_{k,x}(H_{lv})$  holds, then there exists  $0 \leq i \leq n$  such that  $\mathbb{S}_{k,x}(H_c^{(i)})$  holds, which means that  $\diamond \mathbb{S}_{k,x}$  holds in the actual global state reached at the end of the traversal. We formalize these conditions below.

#### 3.3.1 Condition I: Temporal Acyclicity

The first requirement of our framework concerns the order on the memory locations representing the data structure, according to which readers perform their traversals. We require that writers maintain this order *acyclic across intermediate states* of the execution. For example, when the order is based on following pointers in the heap, then, if it is possible to reach location  $y$  from location  $x$  by following a path in which every pointer was present at *some* point in time (not necessarily the same point), then it is not possible to reach  $x$  from  $y$  in the same manner. This requirement is needed in order to ensure that the order is robust even from the perspective of a concurrent reading operation, whose local view is obtained from a fusion of fractions of states.

We begin formalizing this requirement with the notion of search order on memory.

**Search order.** The acyclicity requirement is based on a mapping from a state  $H$  to a *partial order* that  $H$  induces on memory locations, denoted  $\leq_H$ , that captures the order in which operations read the different memory locations. Formally,  $\leq_H$  is a *search order*:

- **Definition 4** (Search order).  $\leq_H$  is a *search order* if it satisfies the following conditions:
- (i) It is *locally determined*: if  $\ell_2$  is an immediate successor of  $\ell_1$  in  $\leq_H$ , then for every  $H'$  such that  $H'(\ell_1) = H(\ell_1)$  it holds that  $\ell_1 \leq_{H'} \ell_2$ .
  - (ii) Search paths follow the order: if there is a  $k$ -search path between  $\ell_1$  and  $\ell_2$  in  $H$ , then  $\ell_1 \leq_H \ell_2$ .
  - (iii) Readers follow the order: reads in  $\bar{r}$  always read a location further in the order in the current global state. Namely, if  $\ell'$  is the last location read, the next read  $r$  reads a location  $\ell$  from the state  $H_c^{(m)}$  such that  $\ell' \leq_{H_c^{(m)}} \ell$ .

Note that the locality of the order is helpful for the ability of readers to follow the order: the next location can be known to come forward in the order solely from the last value the thread reads.

► **Example 5.** In the example of Figure 1, the order  $\leq_H$  is defined by following pointers from parent to children, i.e., all the fields of  $x.left$  and  $x.right$  are ordered after the fields of  $x$ , and the fields of an object are ordered by  $x.key < x.del < \{x.left, x.right\}$ . It is easy to see that this is a search order. Locality follows immediately, and so does the property that search paths follow the order. The fact that the read-in-order property holds for all the methods in Figure 1 follows from a very simple syntactic analysis, e.g., in the case of `locate(k)`, children are always read after their parents and the field `key` is always accessed before `left` or `right`.

► **Remark 6.** Different search orders may be used for different traversals and different  $k$ 's when establishing  $\diamond(\text{root} \stackrel{k}{\rightsquigarrow} x)$  at the end of the traversal. In Definition 4, condition (iii) considers (just) the reads performed by the traversal of interest, and condition (ii) considers the possible search paths it constructs in the local view (just) for the  $k$  of interest.

**Accumulated order and acyclicity.** The accumulated order captures the order as it may be observed by concurrent traversals across different intermediate states. Formally, we define the *accumulated order* w.r.t. a sequence of writes  $\hat{w}_1, \dots, \hat{w}_m$ , denoted  $\leq_{\hat{w}_1 \dots \hat{w}_m}^{\cup} (H_c^{(0)})$ , as the transitive closure of  $\bigcup_{0 \leq s \leq m} \leq_{\hat{w}_1 \dots \hat{w}_s} (H_c^{(0)})$ . In our example, the accumulated order consists of all parent-children links created during an execution. We require:

► **Definition 7 (Acyclicity).** We say that  $\leq_H$  satisfies *acyclicity of accumulated order* w.r.t. a sequence  $\bar{w} = w_1, \dots, w_n$  of writes if the accumulated order  $\leq_{w_1 \dots w_n}^{\cup} (H_c^{(0)})$  is a partial order.

► **Example 8.** In our running example, acyclicity holds because `insert`, `remove`, and `rotate` modify the pointers from a node only to point to new nodes, or to nodes that have already been reachable from that node. Modifications to other fields have no effect on the order. Note that `rotate` does not perform the rotation in place, but allocates a new object. Therefore, the accumulated order, which consists of all parent-children links created during an execution, is acyclic, and hence remains a partial order.

### 3.3.2 Condition II: Preservation of Search Paths

The second requirement of our framework is that for every write action  $w$  which happens concurrently with the sequence of reads  $\bar{r}$  and modifies location  $mod(w)$ , if  $mod(w)$  was  $k$ -reachable (i.e.,  $\mathbb{S}_{k, mod(w)}$  was true) at some point in time after  $\bar{r}$  started and before  $w$  occurred, then it also holds right before  $w$  is performed. We note that this must hold in the presence of all possible interferences, including writes that operate on behalf of other keys (e.g. `insert(k')`). Formally, we require:

► **Definition 9 (Preservation).** We say that  $\bar{w}$  ensures *preservation of  $k$ -reachability by search paths* if for every  $1 \leq m \leq n$ , if for some  $0 \leq i < m$ ,  $H_c^{(i)} \models \mathbb{S}_{k, mod(w_m)}$  then  $H_c^{(m-1)} \models \mathbb{S}_{k, mod(w_m)}$ .

Note that  $H_c^{(m-1)} \models \mathbb{S}_{k, mod(w_m)}$  iff  $H_c^{(m)} \models \mathbb{S}_{k, mod(w_m)}$  since the search path to  $mod(w_m)$  is not affected by  $w_m$  (by the basic properties of  $\mathbb{S}_{k, mod(w_m)}$ , see Section 3.1).

► **Example 10.** In our running example, preservation holds because  $w_m$  either modifies a location that has never been reachable (such as line 93), in which case preservation holds vacuously, or holds the lock on  $x$  when  $\neg x.\text{rem}$  (without modifying its predecessor earlier under this lock).<sup>3</sup> In the latter case preservation holds because every previous write  $w'$  retains  $\text{root} \stackrel{k}{\rightsquigarrow} \text{mod}(w_m)$  unchanged unless it sets the field `rem` of  $x$  to `true` before releasing the lock on  $x$ . Therefore,  $\text{root} \stackrel{k}{\rightsquigarrow} \text{mod}(w_m)$  is retained still when  $w_m$  is performed. Preservation follows.

We emphasize that the preservation condition only requires that  $k$ -reachability is retained to modified locations  $\ell$  and only at the point of time when the write  $w$  to  $\ell$  is performed;  $k$ -reachability *may* be lost at later points in time. In particular, locations whose reachability has been reduced may be accessed, as long as they are not modified after the reachability loss. For example, consider a rotation as in Figure 2a. The rotation breaks the  $k$ -reachability of  $y$ :  $\text{root} \stackrel{k}{\rightsquigarrow} y$  holds before the rotation but not afterwards. Indeed, our framework does not establish  $\text{root} \stackrel{k}{\rightsquigarrow} y$ , but infers  $\diamond(\text{root} \stackrel{k}{\rightsquigarrow} y)$ , which does hold. In this example, the preservation condition requires that the left and right pointers of  $y$  are not modified after this rotation is performed.<sup>4</sup> On the other hand, concurrent traversals may *access*  $y$ . In the example, this happens when (1) the traversal continues beyond  $y$  in the search for  $k' \neq k$ , and when (2) the traversal searches for  $k$  and terminates in  $y$ .

### 3.3.3 Local View Arguments' Guarantee

We are now ready to formalize our main theorem, relating reachability in the local view (Section 3.2) to reachability in the global state, provided that the conditions from Definitions 7 and 9 are satisfied.

► **Theorem 11.** *If*

- (i)  $\leq_H$  is a search order satisfying the accumulated acyclicity property w.r.t.  $\bar{w}$ , and
- (ii)  $\bar{w}$  ensures preservation of  $k$ -reachability by search paths,

then for every  $k$  and location  $x$ , if  $\mathbb{S}_{k,x}(H_{lv})$  holds, then there exists  $0 \leq i \leq n$  s.t.  $\mathbb{S}_{k,x}(H_c^{(i)})$  holds.

In the extended version [20] we illustrate how violating these conditions could lead to incorrectness of traversals. Section 3.4 discusses the main ideas behind the proof.

## 3.4 Proof Idea

We now sketch the correctness proof of Theorem 11. (The full details appear in the extended version [20].) The theorem transfers  $\mathbb{S}_{k,x}$  from the local view to the global state. Recall that the local view is a fusion of the fractions of states observed by the thread at different times. To relate the two, we study the local view from the lens of a fabricated state: a state resulting from a subsequence of the interfering writes, which includes the observed local view. We exploit the cooperation between the readers and the writers that is guaranteed by the order  $\leq_H$  (which readers and writers maintain) to construct a fabricated state which is closely related to the global state, in the sense that it *simulates* the global state (Definition 12);

<sup>3</sup> In line 94, because  $x$  is a child of  $y$  which is a child of  $p$  and  $\neg p.\text{rem}$ , it follows that  $\neg x.\text{rem}$  because a node marked with `rem` loses its single parent beforehand.

<sup>4</sup> Modification of  $y.\text{rem}$  is allowed because this field does not affect search paths (see Section 3.1).

simulation depends both on the acyclicity requirement and on the preservation requirement (Lemma 14). Deducing the existence of a search path in an intermediate global state out of its existence in the local view is a corollary of this connection (Lemma 13).

**Fabricated state.** The fabricated state provides a means of analyzing the local view and its relation to the global (true) state. A *fabricated state* is a state consistent with the local view (i.e. it agrees with the value of every location present in the local view) that is constructed by a subsequence  $\bar{w}_f = w_{i_1}, \dots, w_{i_k}$  of the writes  $\bar{w}$ . One possible choice for  $\bar{w}_f$  is the subsequence of writes whose effect was observed by  $\bar{r}$  (i.e.  $\bar{r}$  read-from). For relating the local view to the global state, which is constructed from the entire  $\bar{w}$ , it is beneficiary to include in  $\bar{w}_f$  additional writes except for those directly observed by  $\bar{r}$ . In what follows, we choose the subsequence  $\bar{w}_f$  so that the fabricated state satisfies a consistency property of *forward-agreement* with the global state. This means that although not all writes are included in  $\bar{w}_f$  (as the thread misses some), the writes that are included have the same picture of the “continuation” of the data structure as it really was in the global state.

**Construction of fabricated state based on order.** Our construction of the fabricated state includes in  $\bar{w}_f$  all the writes that occurred *backward* in time and wrote to locations *forward* in the order than the current location read, for every location read. (In particular, it includes all the writes that  $\bar{r}$  reads from directly). Formally, let  $\text{mod}(w)$  denote the location modified by write  $w$ . Then for every read  $r$  in  $\bar{r}$  that reads location  $\ell_r$  from global state  $H_c^{(m)}$ , we include in  $\bar{w}_f$  all the writes  $\{w_j \mid j \leq m \wedge \ell_r \leq_{w_1 \dots w_m(H_c^{(0)})} \text{mod}(w_j)\}$  (ordered as in  $\bar{w}$ ). We use the notation  $H_f^{(j)} = w_{i_1} \dots w_{i_j}(H_c^{(0)})$  for intermediate fabricated states. This choice of  $\bar{w}_f$  ensures *forward-agreement* between the fabricated state and the global state: every write  $w_{i_j}$  in  $\bar{w}_f$ , the states on which it is applied,  $H_c^{(i_j-1)}$  and  $H_f^{(j-1)}$  agree on all locations  $\ell$  such that  $\text{mod}(w_{i_j}) \leq_{H_f^{(j-1)}} \ell$ .

In what follows, we fix the fabricated state to be the state resulting at the end of this particular choice of  $\bar{w}_f$ . It satisfies forward-agreement by construction, and is an extension of the local view, relying on the *acyclicity* requirement.

**Simulation.** As we show next, the construction of  $\bar{w}_f$  ensures that the effect of every write in  $\bar{w}_f$  on  $\mathbb{S}_{k,x}$  is guaranteed to concur with its effect on the real state with respect to changing  $\mathbb{S}_{k,x}$  from false to true. We refer to this property as *simulation*.

► **Definition 12 (Simulation).** For a predicate  $\mathbb{P}$ , we say that the subsequence of writes  $w_{i_1} \dots w_{i_k}$   $\mathbb{P}$ -*simulates* the sequence  $w_1 \dots w_n$  if for every  $1 \leq j \leq k$ , if  $\neg \mathbb{P}(H_f^{(j-1)})$  but  $\mathbb{P}(w_{i_j}(H_f^{(j-1)}))$ , then  $\neg \mathbb{P}(H_c^{(i_j-1)}) \implies \mathbb{P}(w_{i_j}(H_c^{(i_j-1)}))$ .

Simulation implies that the write  $w_{i_j}$  in  $\bar{w}_f$  that changed  $\mathbb{S}_{k,x}$  to true on the local view, would also change it on the corresponding global state  $H_c^{(i_j)}$  (unless it was already true in  $H_c^{(i_j-1)}$ ). This provides us with the desired global state where  $\mathbb{S}_{k,x}$  holds. Using also the fact that  $\mathbb{S}_{k,x}$  is upward-absolute [44] (namely, preserved under extensions of the state), we obtain:

► **Lemma 13.** *Let  $\bar{w}_f$  be the subsequence of  $\bar{w} = w_1, \dots, w_n$  defined above. If  $\mathbb{S}_{k,x}(H_{lv})$  holds and  $\bar{w}_f$   $\mathbb{S}_{k,x}$ -simulates  $\bar{w}$ , then there exists some  $0 \leq i \leq n$  s.t.  $\mathbb{S}_{k,x}(H_c^{(i)})$ .*

Finally, we show that the fabricated state satisfies the simulation property. Owing to the specific construction of  $\bar{w}_f$ , the proof needs to relate the effect of writes on states which have a rather strong similarity: they agree on the contents of locations which come forward of the modified location. Preservation complements this by guaranteeing the existence of a path to the modified location:

► **Lemma 14.** *If  $\bar{w}$  satisfies preservation of  $\mathbb{S}_{k, \text{mod}(w)}$  for all  $w$ , then  $\bar{w}_f \mathbb{S}_{k,x}$ -simulates  $\bar{w}$  for all  $x$ .*

To prove the lemma, we show that preservation, together with forward agreement, implies the simulation property, which in turn implies that  $\mathbb{S}_{k,x}(H_f^{(j-1)}) \implies \exists 0 \leq i \leq i_{j-1} \mathbb{S}_{k,x}(H_c^{(i)})$  (see Lemma 13). To show simulation, consider a write  $w_{i_j}$  that creates a  $k$ -search path  $\zeta$  to  $x$  in  $H_f^{(j)}$ . We construct such a path in the corresponding global state. The idea is to divide  $\zeta$  to two parts: the prefix until  $\text{mod}(w_{i_j})$ , and the rest of the path. Relying on forward agreement, the latter is exactly the same in the corresponding global state, and preservation lets us prove that there is also an appropriate prefix: necessarily there has been a  $k$ -search path to  $\text{mod}(w_{i_j})$  in the fabricated state *before*  $w_{i_j}$ , so by induction, exploiting the fact that simulation up to  $j-1$  implies that  $\mathbb{S}_{k,x}(H_f^{(j-1)}) \implies \exists 0 \leq i \leq i_{j-1} \mathbb{S}_{k,x}(H_c^{(i)})$ , there has been a  $k$ -search path to  $\text{mod}(w_{i_j})$  in some intermediate global state that occurred earlier than the time of  $w_{i_j}$ . Since  $w_{i_j}$  writes to  $\text{mod}(w_{i_j})$ , the preservation property ensures that there is a  $k$ -search path to  $\text{mod}(w_{i_j})$  in the global state also at the time of the write  $w_{i_j}$ , and the claim follows.

## 4 Putting It All Together: Proving Linearizability Using Local Views

Recall that our overarching objective in developing the local view argument (Section 3) is to prove the correctness of assertions used in linearizability proofs (e.g., in Section 2.1). We now summarize the steps in the proof of the assertions. Overall, it is composed of the following steps:

1. Establishing properties of traversals on the local view using sequential reasoning,
2. Establishing the acyclicity and preservation conditions by simple concurrent reasoning, and
3. Proving the assertions when relying on local view arguments, augmented with some concurrent reasoning.

For the running example, step 1 is presented in Example 3, and step 2 consists of Examples 8 and 10 (see the extended version [20] for a full formal treatment). Step 3 concludes the proof as discussed in Section 2.2.

► **Remark 15.** While the local view argument, relying in particular on step 2, was developed to simplify the proofs of the assertions in 3, this goes also in the other direction. Namely, the concurrent reasoning required for proving the conditions of the framework (e.g., preservation) can be greatly simplified by relying on the correctness of the assertions (as they constrain possible interfering writes). Indeed, the proofs may mutually rely on each other. This is justified by a proof by induction: we prove that the current write satisfies the condition in the assertion, assuming that all previous writes did. This is also allowed in proofs of the conditions in Section 3.3, because they refer to the effect of *interfering* writes, that are known to conform to their respective assertions from the induction hypothesis. Hence, carrying these proofs together avoids circular reasoning and ensures validity of the proof.



## 5 Additional Case Studies

### 5.1 Lazy and Optimistic Lists

We successfully applied our framework to prove the linearizability of sorted-list-based concurrent set implementations with unsynchronized reads. Our framework is capable of verifying various versions of the algorithm in which `insert` and `delete` validate that the nodes they locked are reachable using a boolean field, as done in the lazy list algorithm [23], or by rescanning the list, as done in the optimistic list algorithm [27, Chap 9.8]. Our framework is also applicable for verifying implementations of the lazy list algorithm in which the logical deletion and the physical removal are done by the same operation or by different ones. We give a taste of these proofs here.

Figure 3 shows an annotated pseudo-code of the lazy list algorithm. Every operation starts with a call to `locate(k)`, which performs a standard search in a sorted list – without acquiring any locks – to locate the node with the target key  $k$ . This method returns the last link it traverses,  $(x, y)$ . Figure 3 includes two variants of `contains(k)`: In one variant, it returns `true` only if it finds a node with key  $k$  that is not logically deleted (line 139), while in the second variant it returns `true` even if that node is logically deleted (the commented `return` at line 141). Interestingly, the same annotations allow to verify both variants, and the proof differs only in the abstraction function mapping states of the list to abstract sets. Modifications of a node in the list are synchronized with the node’s lock. An `insert(k)` operation calls `locate`, and then links a new node to the list if  $k$  was not found. `delete(k)` logically deletes  $y$  (after validating that  $y$  remained linked to the list after its lock was acquired), and then physically removes it.

As in Section 2, the assertions contain predicates of the form  $\text{root} \stackrel{k}{\rightsquigarrow} x$ , which means that  $x$  resides on a *valid search path* for key  $k$  that starts at `root`; the formal definition of a search path in the lazy list appears below. Note that  $\text{root} \stackrel{k}{\rightsquigarrow} \text{null}$  indicates that  $k$  is not in the list.

$$o_r \stackrel{k}{\rightsquigarrow} o_x \stackrel{\text{def}}{=} \exists o_0, \dots, o_m. o_0 = o_r \wedge o_m = o_x \wedge \forall i = 1..m. o_{i-1}.key < k \wedge o_{i-1}.next = o_i$$

We prove the linearizability of the algorithm using an *abstraction function*. One abstraction function we may use maps  $H$  to the set of keys of the nodes that are on a valid search path for their key and are not logically deleted in  $H$ :

$$\mathcal{A}^{\text{logical}}(H) = \{k \in \mathbb{N} \mid H \models \exists x. \text{root} \stackrel{k}{\rightsquigarrow} x \wedge x.key = k \wedge \neg x.mark\}.$$

Another possibility is to define the abstract set to be the keys of all the reachable nodes:

$$\mathcal{A}^{\text{physical}}(H) = \{k \in \mathbb{N} \mid H \models \exists x. \text{root} \stackrel{k}{\rightsquigarrow} x \wedge x.key = k\}.$$

We note that  $\mathcal{A}^{\text{logical}}(H)$  can be used to verify the code of `contains` as written, while  $\mathcal{A}^{\text{physical}}(H)$  allows to change the algorithm to return `true` in line 141. In both cases, the proof of linearizability is carried out using the same assertions currently annotating the code. In the rest of this section, we discuss the verification of the code in Figure 3 as written, and thus use  $\mathcal{A}(H) = \mathcal{A}^{\text{logical}}$  as the abstraction function. The assertions almost immediately imply that for every operation invocation  $op$ , there exists a state  $H$  during  $op$ ’s execution for which the abstract state  $\mathcal{A}(H)$  agrees with  $op$ ’s return value, and so  $op$  can be linearized at  $H$ ; we need only make the following observations. First, `contains()` and a failed `delete()` or `insert()` do not modify the memory, and so can be linearized at the point in time in which the assertions before their `return` statements hold. Second, in the state  $H$  in which a successful `delete(k)` (respectively, `insert(k)`) performs a write, the assertions on line 156



```

97 type N
98   int key
99   N next
100  bool mark
101
102 N root ← new N(-∞);
103
104 N×N locate(int k)
105   x, y ← root
106   while (y ≠ null ∧ y.key < k)
107     x ← y
108     y ← x.next
109   {◇(root  $\overset{k}{\rightsquigarrow}$  x ∧ x.next = y)}
110   {x.key < k ∧ (y ≠ null ⇒ y.key ≥ k)}
111   return (x, y)
112
113 bool insert(int k)
114   (x, y) ← locate(k)
115   if (y ≠ null ∧ y.key = k)
116     {◇(root  $\overset{k}{\rightsquigarrow}$  y) ∧ y.key = k}
117     return false
118   lock(x)
119   lock(y)
120   if (x.mark ∨ x.next ≠ y)
121     restart
122   {¬x.mark ∧ x.next = y}
123   z ← new N(k)
124   {y ≠ null ⇒ k > y.key}
125   z.next ← y
126   {root  $\overset{k}{\rightsquigarrow}$  x ∧ x.next = y ∧ x.key < k ∧
127     z.next = y ∧ ¬z.mark ∧
128     (y ≠ null ⇒ k > y.key)}
127   x.next ← z
128   return true
129
129 bool contains(int k)
130   (_, y) ← locate(k)
131   if (y = null)
132     {◇(root  $\overset{k}{\rightsquigarrow}$  null)}
133     return false
134   if (y.key ≠ k)
135     {◇(root  $\overset{k}{\rightsquigarrow}$  x ∧ x.next = y) ∧ k < x.key ∧ y.key > k}
136     return false
137   if (¬y.mark)
138     {root  $\overset{k}{\rightsquigarrow}$  y ∧ y.key = k ∧ ¬y.mark}
139     return true
140   {◇(root  $\overset{k}{\rightsquigarrow}$  y) ∧ y.key = k ∧ y.mark}
141   return false // return true
142
143 bool delete(int k)
144   (x, y) ← locate(k)
145   if (y = null)
146     {◇(root  $\overset{k}{\rightsquigarrow}$  null)}
147     return false
148   if (y.key ≠ k)
149     {◇(root  $\overset{k}{\rightsquigarrow}$  x ∧ x.next = y) ∧ x.key < k ∧ y.key > k}
150     return false
151   {y.key = k}
152   lock(x)
153   lock(y)
154   if (x.mark ∨ y.mark ∨ x.next ≠ y)
155     restart
156   {root  $\overset{k}{\rightsquigarrow}$  x ∧ x.next = y ∧ y.key = k ∧ ¬x.mark ∧ ¬y.mark}
157   y.mark ← true
158   {root  $\overset{k}{\rightsquigarrow}$  x ∧ x.next = y ∧ y.key = k ∧ ¬x.mark ∧ y.mark}
159   x.next ← y.next
160   return true

```

■ **Figure 3** Lazy List [23]. The code is annotated with assertions written inside curly braces. For brevity, **unlock** operations are omitted; a procedure releases all the locks it acquired when it terminates or **restarts**.

(respectively, line 126) imply that  $k \in \mathcal{A}(H)$  (respectively,  $k \notin \mathcal{A}(H)$ ). Therefore, these writes change the abstract set, making it agree with the operation's return value of *true*. Finally, it only remains to verify that the physical removal performed by `delete(k)` in state  $H$  does not modify  $\mathcal{A}(H)$ . Indeed, as an operation modifies a field of node  $v$  only when it has  $v$  locked, it is easy to see that for any node  $x$  and key  $k$ , if  $\text{root} \overset{k}{\rightsquigarrow} x$  held before the write, then it also holds afterwards with the exception of the removed node  $y$ . However, `delete(k)` removes a deleted node, and thus does not change  $\mathcal{A}(H)$ .

The proof of the assertions in Figure 3 utilizes a local view argument for the  $\diamond$  assertion in line 109 for the predicate  $\text{root} \overset{k}{\rightsquigarrow} x \wedge x.\text{next} = y$ , using the extension with a single field discussed in Remark 2. The conditions of the local view argument are easy to prove: The acyclicity requirement is evident, as writes modify the pointers from a node only to point to new nodes, or to nodes that have already been reachable from that node. Preservation holds because a write either (i) marks a node, which does not affect the search paths; (ii) modifies a location that has never been reachable (such as line 125), in which case preservation holds vacuously; (iii) removes a marked node  $y$  (line 159) which removes all the search paths that go through it. However, as  $y$  is marked, its fields are not going to be modified later on, and thus  $y$  cannot be the cause of violating preservation. Furthermore, all search paths that reach  $y$ 's successor before the removal are retained and merely get shorter; or (iv) adds a reachable node  $z$  in between two reachable nodes  $x$  and  $y$  (line 127). However, as  $z$ 's key is smaller than  $y$ 's, the insertion preserves any search paths which goes *through*  $y$ 's next pointer.

As for the rest of the assertions, when `insert` and `delete` lock  $x$  and see that it is not marked, the  $\text{root} \overset{k}{\rightsquigarrow} x$  property follows from the  $\diamond(\text{root} \overset{k}{\rightsquigarrow} x)$  deduced above by a local view argument using the same invariant in preservation above.<sup>5</sup> The remainder assertions are attributed to reading a location under the protection of a lock, e.g.  $\neg x.\text{mark}$  in line 122.

## 5.2 Lock-free List and Skip-List

We used our framework to prove the linearizability a sorted lock-free list-based concurrent set algorithm [27, Chapter 9.8] and of a lock-free skip-list-based concurrent set algorithm [27, Chapter 14.4]. In these proofs we use local view arguments to prove the concurrent traversals of the `contains` method, which is the most difficult part of the proofs: `add` and `remove` use the internal `find` which traverses the list and also prunes out marked nodes, and thus their correctness follows easily from an invariant ensuring the reachability of unmarked nodes. The proofs appear in [20].

## 6 Related Work

Verifying linearizability of concurrent data structures has been studied extensively. Some techniques, e.g., [1, 2, 18, 51, 49], apply to a restricted set of algorithms where the linearization point of every invocation is *fixed* to a particular statement in the code. While these works provide more automation, they are not able to deal with the algorithms considered in our paper where for instance, the linearization point of `contains(k)` invocations is not fixed. Generic reductions of verifying linearizability to checking a set of assertions in the code have been defined in [5, 6, 7, 34, 24, 50, 53]. These works apply to algorithms with non-fixed linearization points, but they do not provide a systematic methodology for proving the assertions, which is the main focus of our paper.

Verifying linearizability has also been addressed in the context of defining program logics for *compositional* reasoning about concurrent programs. In this context, the goal is to define a proof methodology that allows composing proofs of program's components to get a proof for the entire program, which can also be reused in every valid context of using that program. Improving on the classical Owicki-Gries [39] and Rely-Guarantee [29] logics, various extensions of Concurrent Separation Logic [4, 9, 38, 40] have been proposed in order to reason compositionally about different instances of fine-grained concurrency, e.g. [30, 33, 15, 42, 46, 47]. However, they focus on the reusability of a proof of a component in a larger context (when composed with other components) while our work focuses on simplifying the proof goals that guarantee linearizability. The concurrent reasoning needed for our framework could be carried out using one of these logics.

The proof of linearizability of the lazy-list algorithm given in [37] is based on establishing the conditions required by the *hindsight lemma* [37, Lemma 5.2]. The lemma states that every link traversed during an unsynchronized traversal was indeed reachable at some point in time between the beginning of the traversal and the moment the link was crossed. This enables verifying the correctness of the `contains` method using, effectively, sequential reasoning. The hindsight lemma is a specific instance of the extension discussed in Remark 2, and its

---

<sup>5</sup> As in Section 5.2, these assertions could also be deduced directly from a slightly stronger invariant that unmarked nodes are reachable and that the list is sorted. This is not the case in the optimistic list of [27, Chap9.8] which rescans instead of using a marked bit. In both cases `contains` requires a local view argument.

assumptions narrows its application to concurrent set algorithms implemented using sorted singly-linked lists. In contrast, we present a fundamental technique which is based on far more generic properties which is applicable to list and tree-based data structures alike.

The proof methodology for proving linearizability of [32] relies on properties of the data structure in sequential executions. The methodology assumes the existence of *base points*, which are points in time during the concurrent execution of a search in which some predicate holds over the shared state. For instance, when applying the methodology to the lazy list, they prove the existence of base points using prior techniques [37, 52] that employ tricky concurrent reasoning. Our work is thus complementary to theirs: our proof argument is meant to replace the latter kind of reasoning, and can thus simplify proofs of the existence of base points.

The *Edgeset* framework of Shasha and Goodman [43], which has recently been formalized using concurrent separation logic [31], provides conditions for the linearizability of concurrent search data structures. It relies on a precondition that for any operation on key  $k$ ,  $\text{root} \stackrel{k}{\rightsquigarrow} x$  holds when the operation looks for, inserts, or deletes  $k$  at  $x$ . However, the optimistic data structures that we consider often do not satisfy this precondition, making the Edgeset framework inapplicable. (Example 10 describes how this precondition does not hold in our search tree example, and a similar issue exists in the lazy-list.) Moreover, the Edgeset precondition implies that the linearization point of an operation occurs at one of its own atomic steps. Our framework does not have this requirement. Shasha and Goodman also describe three algorithm templates and prove, using concurrent reasoning, that these templates satisfy the preconditions of the Edgeset framework. In contrast, our argument uses sequential reasoning for traversals, and our concurrent proofs consider only the effects of interleaving writes – not both reads and writes.

## 7 Conclusions and Future Work

This paper presents a novel approach for constructing linearizability proofs of concurrent search data structures. We present a general proof argument that is applicable to many existing algorithms, uncovering fundamental structure – the acyclicity and preservation conditions – shared by them. We have instantiated our framework for a self-balancing binary search tree, lists with lazy [23] or non-blocking [27] synchronization, and a lock-free skip list. To the best of our knowledge, our work is the first to prove linearizability of a self-balancing binary search tree using a unified proof argument.

An important direction for future work is the mechanism of backtracking. Some algorithms, including the original CF tree [12, 14], backtrack instead of restarting when their optimistic validation fails. In the CF tree, backtracking is implemented by directing pointers from child to parent, breaking our acyclicity requirement. A similar situation arises in the in-place rotation of [8]. Handling these scenarios in our proof argument is an interesting direction for future work.

An additional direction to explore is validations performed during traversals. For example, the SnapTree algorithm [8] performs in-place rotations which violate preservation. The algorithm overcomes this by performing hand-over-hand validation during a lock-free traversal. This validation, consisting of re-reading previous locations and ensuring version numbers have not changed, does not fit our approach of sequential reasoning on traversals.

The preservation of reachability to location of modification arises naturally out of the correctness of traversals in modifying operations, ensuring that the conclusion of the traversal – the existence of a path – holds not only in some point in the past, but also holds at the

time of the modification. We show that, surprisingly, preservation, when it is combined with the order, suffices to reason about the traversal by a local view argument. We base the correctness of read-only operations on the same predicates, and so rely on the same property. It would be interesting to explore different criteria which ensure the simulation of the fabricated state constructed based on the accumulated order.

Finding ways to extend the framework in these directions is an interesting open problem. This notwithstanding, we believe that our framework captures important principles underlying modern highly concurrent data structures that could prove useful both for structuring linearizability proofs and elucidating the correctness principles behind new concurrent data structures.

---

## References

- 1 Parosh Aziz Abdulla, Frédéric Haziza, Lukás Holík, Bengt Jonsson, and Ahmed Rezine. An integrated specification and verification technique for highly concurrent data structures. In *TACAS*, pages 324–338, 2013. doi:10.1007/978-3-642-36742-7\_23.
- 2 Daphna Amit, Noam Rinetzky, Thomas W. Reps, Mooly Sagiv, and Eran Yahav. Comparison under abstraction for verifying linearizability. In *CAV '07*, volume 4590 of *LNCS*, pages 477–490, 2007. doi:10.1007/978-3-540-73368-3\_49.
- 3 Maya Arbel and Hagit Attiya. Concurrent Updates with RCU: Search Tree As an Example. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, pages 196–205, New York, NY, USA, 2014. ACM. doi:10.1145/2611462.2611471.
- 4 Richard Bornat, Cristiano Calcagno, Peter W. O'Hearn, and Matthew J. Parkinson. Permission accounting in separation logic. In Jens Palsberg and Martín Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 259–270. ACM, 2005. doi:10.1145/1040305.1040327.
- 5 Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Jad Hamza. Verifying concurrent programs against sequential specifications. In *ESOP '13*, volume 7792 of *LNCS*, pages 290–309. Springer, 2013.
- 6 Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Jad Hamza. On reducing linearizability to state reachability. In *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part II*, pages 95–107, 2015.
- 7 Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Suha Orhun Mutluergil. Proving linearizability using forward simulations. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, volume 10427 of *Lecture Notes in Computer Science*, pages 542–563. Springer, 2017. doi:10.1007/978-3-319-63390-9\_28.
- 8 Nathan Grasso Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2010, Bangalore, India, January 9-14, 2010*, pages 257–268, 2010.
- 9 Stephen D. Brookes. A semantics for concurrent separation logic. In *CONCUR 2004 - Concurrency Theory, 15th International Conference, London, UK, August 31 - September 3, 2004, Proceedings*, pages 16–34, 2004. doi:10.1007/978-3-540-28644-8\_2.
- 10 Trevor Brown, Faith Ellen, and Eric Ruppert. A general technique for non-blocking trees. In *PPoPP*, 2014.
- 11 Austin T. Clements, M. Frans Kaashoek, and Nikolai Zeldovich. Scalable address spaces using RCU balanced trees. In *ASPLOS*, 2012.

- 12 Tyler Crain, Vincent Gramoli, and Michel Raynal. A contention-friendly binary search tree. In Felix Wolf, Bernd Mohr, and Dieter an Mey, editors, *Euro-Par 2013 Parallel Processing*, pages 229–240, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
- 13 Tyler Crain, Vincent Gramoli, and Michel Raynal. No Hot Spot Non-blocking Skip List. In *ICDCS*, 2013.
- 14 Tyler Crain, Vincent Gramoli, and Michel Raynal. A fast contention-friendly binary search tree. *Parallel Processing Letters*, 26(03):1650015, 2016. doi:10.1142/S0129626416500158.
- 15 Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. Tada: A logic for time and data abstraction. In Richard E. Jones, editor, *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings*, volume 8586 of *Lecture Notes in Computer Science*, pages 207–231. Springer, 2014. doi:10.1007/978-3-662-44202-9\_9.
- 16 Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. In *ASPLOS*, 2015.
- 17 Dana Drachler, Martin Vechev, and Eran Yahav. Practical Concurrent Binary Search Trees via Logical Ordering. In *PPoPP*, 2014.
- 18 Cezara Dragoi, Ashutosh Gupta, and Thomas A. Henzinger. Automatic linearizability proofs of concurrent objects with cooperating updates. In *CAV '13*, volume 8044 of *LNCS*, pages 174–190. Springer, 2013. doi:10.1007/978-3-642-39799-8\_11.
- 19 Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking Binary Search Trees. In *PODC*, 2010.
- 20 Yotam M. Y. Feldman, Constantin Enea, Adam Morrison, Noam Rinetzky, and Sharon Shoham. Order out of chaos: Proving linearizability using local views. *CoRR*, abs/1805.03992, 2018. arXiv:1805.03992.
- 21 Keir Fraser. *Practical lock-freedom*. PhD thesis, University of Cambridge, Computer Laboratory, University of Cambridge, Computer Laboratory, February 2004.
- 22 Timothy L. Harris. A Pragmatic Implementation of Non-blocking Linked-Lists. In *DISC*, 2001.
- 23 Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, Bill Scherer, and Nir Shavit. A lazy concurrent list-based set algorithm. In *OPODIS*, 2005.
- 24 Thomas A. Henzinger, Ali Sezgin, and Viktor Vafeiadis. Aspect-oriented linearizability proofs. In *CONCUR*, pages 242–256, 2013. doi:10.1007/978-3-642-40184-8\_18.
- 25 M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *TOPLAS*, 12(3), 1990.
- 26 Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. A Simple Optimistic Skiplist Algorithm. In *SIROCCO*, 2007.
- 27 Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.
- 28 Shane V. Howley and Jeremy Jones. A Non-blocking Internal Binary Search Tree. In *SPAA*, 2012.
- 29 Cliff B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332, 1983.
- 30 Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 637–650. ACM, 2015. doi:10.1145/2676726.2676980.

- 31 Siddharth Krishna, Dennis E. Shasha, and Thomas Wies. Go with the flow: compositional abstractions for concurrent data structures. *PACMPL*, 2(POPL):37:1–37:31, 2018. doi:10.1145/3158125.
- 32 Kfir Lev-Ari, Gregory V. Chockler, and Idit Keidar. A constructive approach for proving data structures' linearizability. In Yoram Moses, editor, *Distributed Computing - 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings*, volume 9363 of *Lecture Notes in Computer Science*, pages 356–370. Springer, 2015. doi:10.1007/978-3-662-48653-5\_24.
- 33 Ruy Ley-Wild and Aleksandar Nanevski. Subjective auxiliary state for coarse-grained concurrency. In Roberto Giacobazzi and Radhia Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 561–574. ACM, 2013. doi:10.1145/2429069.2429134.
- 34 Hongjin Liang and Xinyu Feng. Modular verification of linearizability with non-fixed linearization points. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 459–470, 2013.
- 35 Maged M. Michael. High Performance Dynamic Lock-free Hash Tables and List-based Sets. In *SPAA*, 2002.
- 36 Aravind Natarajan and Neeraj Mittal. Fast Concurrent Lock-free Binary Search Trees. In *PPoPP*, 2014.
- 37 P. W. O'Hearn, N. Rinetzkky, M. T. Vechev, E. Yahav, and G. Yorsh. Verifying linearizability with hindsight. In *29th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 85–94, 2010.
- 38 Peter W. O'Hearn. Resources, concurrency and local reasoning. In *CONCUR 2004 - Concurrency Theory, 15th International Conference, London, UK, August 31 - September 3, 2004, Proceedings*, pages 49–67, 2004. doi:10.1007/978-3-540-28644-8\_4.
- 39 Susan S. Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM*, 19(5):279–285, 1976. doi:10.1145/360051.360224.
- 40 Matthew J. Parkinson, Richard Bornat, and Peter W. O'Hearn. Modular verification of a non-blocking stack. In Martin Hofmann and Matthias Felleisen, editors, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 297–302. ACM, 2007. doi:10.1145/1190216.1190261.
- 41 Arunmoezhi Ramachandran and Neeraj Mittal. A Fast Lock-Free Internal Binary Search Tree. In *ICDCN*, 2015.
- 42 Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Specifying and verifying concurrent algorithms with histories and subjectivity. In Jan Vitek, editor, *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9032 of *Lecture Notes in Computer Science*, pages 333–358. Springer, 2015. doi:10.1007/978-3-662-46669-8\_14.
- 43 Dennis E. Shasha and Nathan Goodman. Concurrent search structure algorithms. *ACM Trans. Database Syst.*, 13(1):53–90, 1988. doi:10.1145/42201.42204.
- 44 Joseph R Shoenfield. The problem of predicativity. In *Mathematical Logic In The 20th Century*, pages 427–434. World Scientific, 2003.
- 45 Josh Triplett, Paul E. McKenney, and Jonathan Walpole. Resizable, Scalable, Concurrent Hash Tables via Relativistic Programming. In *USENIX ATC*, 2011.
- 46 Aaron Turon, Derek Dreyer, and Lars Birkedal. Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston*,



- MA, USA - September 25 - 27, 2013*, pages 377–390. ACM, 2013. doi:10.1145/2500365.2500600.
- 47 V. Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, 2008.
  - 48 V. Vafeiadis, M. Herlihy, T. Hoare, and M. Shapiro. Proving correctness of highly-concurrent linearisable objects. In *PPoPP*, 2006.
  - 49 Viktor Vafeiadis. Shape-value abstraction for verifying linearizability. In *VMCAI '09: Proc. 10th Intl. Conf. on Verification, Model Checking, and Abstract Interpretation*, volume 5403 of *LNCS*, pages 335–348. Springer, 2009. doi:10.1007/978-3-540-93900-9\_27.
  - 50 Viktor Vafeiadis. Automatically proving linearizability. In *CAV '10*, volume 6174 of *LNCS*, pages 450–464, 2010. doi:10.1007/978-3-642-14295-6\_40.
  - 51 Viktor Vafeiadis, Maurice Herlihy, Tony Hoare, and Marc Shapiro. Proving correctness of highly-concurrent linearisable objects. In *PPOPP '06*, pages 129–136. ACM, 2006. doi:10.1145/1122971.1122992.
  - 52 Viktor Vafeiadis, Maurice Herlihy, Tony Hoare, and Marc Shapiro. A safety proof of a lazy concurrent list-based set implementation. Technical Report UCAM-CL-TR-659, University of Cambridge, Computer Laboratory, 2006.
  - 53 He Zhu, Gustavo Petri, and Suresh Jagannathan. Poling: SMT aided linearizability proofs. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, pages 3–19, 2015.





# Redundancy in Distributed Proofs

**Laurent Feuilloley**

IRIF, CNRS and University Paris Diderot, France  
feuilloley@irif.fr

**Pierre Fraigniaud**

IRIF, CNRS and University Paris Diderot, France  
pierref@irif.fr

**Juho Hirvonen**

University of Freiburg, Germany  
juho.hirvonen@cs.uni-freiburg.de

**Ami Paz**

IRIF, CNRS and University Paris Diderot, France  
amipaz@irif.fr

**Mor Perry**

School of Electrical Engineering, Tel-Aviv University, Israel  
mor@eng.tau.ac.il

---

## Abstract

Distributed proofs are mechanisms enabling the nodes of a network to collectively and efficiently check the correctness of Boolean predicates on the structure of the network (e.g. having a specific diameter), or on data structures distributed over the nodes (e.g. a spanning tree). We consider well known mechanisms consisting of two components: a *prover* that assigns a *certificate* to each node, and a distributed algorithm called *verifier* that is in charge of verifying the distributed proof formed by the collection of all certificates. We show that many network predicates have distributed proofs offering a high level of redundancy, explicitly or implicitly. We use this remarkable property of distributed proofs to establish perfect tradeoffs between the *size of the certificate* stored at every node, and the *number of rounds* of the verification protocol.

**2012 ACM Subject Classification** Networks → Error detection and error correction, Theory of computation → Distributed computing models, Computer systems organization → Redundancy

**Keywords and phrases** Distributed verification, Distributed graph algorithms, Proof-labeling schemes, Space-time tradeoffs, Non-determinism

**Digital Object Identifier** 10.4230/LIPIcs.DISC.2018.24

**Related Version** A full version of the paper is available at <https://arxiv.org/abs/1803.03031>.

**Funding** Research supported by the French-Israeli Laboratory on Foundations of Computer Science (FILOFOCS). The first four authors supported by the ANR project DESCARTES. The first two authors receive additional support from INRIA project GANG. Third author supported by the Ulla Tuominen Foundation. Fourth author supported by the Fondation Sciences Mathématiques de Paris (FSMP).

**Acknowledgements** We thank Seri Khoury and Boaz Patt-Shamir for valuable discussions, and the anonymous reviewers of DISC 2018.



© Laurent Feuilloley, Pierre Fraigniaud, Juho Hirvonen, Ami Paz, and Mor Perry;  
licensed under Creative Commons License CC-BY

32nd International Symposium on Distributed Computing (DISC 2018).

Editors: Ulrich Schmid and Josef Widder; Article No. 24; pp. 24:1–24:18

Leibniz International Proceedings in Informatics



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

## 1 Introduction

### 1.1 Context and Objective

In the context of distributed fault-tolerant computing in large scale networks, it is of the utmost importance that the computing nodes can perpetually check the correctness of distributed data structures (e.g., spanning trees) encoded distributedly over the network. Indeed, such data structures can be the outcome of an algorithm that might be subject to failures, or be a-priori correctly given data structures but subject to later corruption. Several mechanisms exist enabling checking the correctness of distributed data structures (see, e.g., [2, 3, 6, 10–12]). For its simplicity and versatility, we shall focus on one classical mechanism known as *proof-labeling schemes* [31], a.k.a. *locally checkable proofs* [25]<sup>1</sup>.

Roughly, a proof-labeling scheme assigns *certificates* to each node of the network. These certificates can be viewed as forming a distributed proof of the actual data structure (e.g., for a spanning tree, the identity of a root, and the distance to this root in the tree). The nodes are then in charge of collectively verifying the correctness of this proof. The requirements are in a way similar to those imposed on non-deterministic algorithms (e.g., the class NP), namely: (1) on correct structures, the assigned certificates must be accepted, in the sense that every node must accept its given certificate; (2) on corrupted structures, whatever certificates are given to the nodes, they must be rejected, in the sense that at least one node must reject its given certificate. (The rejecting node(s) can raise an alarm, or launch a recovery procedure). Proof-labeling schemes and locally checkable proofs can be viewed as a form of non-deterministic distributed computing (see also [19]).

The main measure of quality for a proof-labeling scheme is the *size* of the certificates assigned to *correct* (a.k.a. *legal*) data structures. Indeed, these certificates are verified using protocols that exchange them between neighboring nodes. Thus using large certificates may result in significant overheads in term of communication. Also, proof-labeling schemes might be combined with other mechanisms enforcing fault-tolerance, including replication. Large certificates may prevent replication, or at the least result in significant overheads in term of space complexity if using replication.

Proof-labeling schemes are extremely versatile, in the sense that they can be used to certify *any* distributed data structure or graph property. For instance, to certify a spanning tree, there are several proof-labeling schemes, each using certificates of logarithmic size [26, 31]. Similarly, certifying a minimum-weight spanning tree (MST) can be achieved with certificates of size  $\Theta(\log^2 n)$  bits in  $n$ -node networks [29, 31]. Moreover, proof-labeling schemes are very *local*, in the sense that the verification procedure performs in just one round of communication, each node accepting or rejecting based solely on its certificate and the certificates of its neighbors. However, this versatility and locality comes with a cost. For instance, certifying rather simple graph property, such as certifying that each node holds the value of the diameter of the network, requires certificates of  $\tilde{\Omega}(n)$  bits [13]<sup>2</sup>. There are properties that require even larger certificates. For instance, certifying that the network is non 3-colorable, or certifying that the network has a non-trivial automorphism both require certificates of  $\tilde{\Omega}(n^2)$  bits [25]. The good news though is that all distributed data structures (and graph properties) can be certified using certificates of  $O(n^2 + kn)$  bits, where  $k$  is the size of the part of the data structure stored at each node – see [25, 31].

<sup>1</sup> These two mechanisms slightly differ: the latter assumes that every node can have access to the whole state of each of its neighbors, while the former assumes that only part of this state is visible from neighboring nodes; nevertheless, the two mechanisms share the same essential features.

<sup>2</sup> The tilde-notation is similar to the big-O notation, but also ignores poly-logarithmic factors.

Several attempts have been made to make proof-labeling schemes more efficient. For instance, it was shown in [9] that randomization helps a lot in terms of *communication* costs, typically by hashing the certificates, but this might actually come at the price of dramatically increasing the certificate size. Sophisticated deterministic and efficient solutions have also been provided for reducing the size of the certificates, but they are targeting specific structures only, such as MST [30]. Another direction for reducing the size of the certificates consists of relaxing the decision mechanism, by allowing each node to output more than just a single bit (accept or reject) [4, 5]. For instance, certifying cycle-freeness simply requires certificates of  $O(1)$  bits with just 2-bit output, while certifying cycle-freeness requires certificates of  $\Omega(\log n)$  bits with 1-bit output [31]. However, this relaxation assumes the existence of a centralized entity gathering the outputs from the nodes, and there are still network predicates that require certificates of  $\tilde{\Omega}(n^2)$  bits even under this relaxation. Another notable approach is using approximation [13], which reduces, e.g., the certificate size for certifying the diameter of the graph from  $\Omega(n)$  down to  $O(\log n)$ , but at the cost of only determining if the given value is up to two times the real diameter.

In this paper, we aim at designing deterministic and generic ways for reducing the certificate size of proof-labeling schemes. This is achieved by following the guidelines of [33], that is, trading time for space by exploiting the inherent redundancy in distributed proofs.

## 1.2 Our Results

As mentioned above, proof-labeling schemes include a verification procedure consisting of a single round of communication. In a nutshell, we prove that using more rounds of communication for verifying the certificates enables to reduce significantly the size of these certificates, often by a factor super-linear in the number of rounds, and sometimes even exponential.

More specifically, a proof-labeling scheme of radius  $t$  (where  $t$  can depend on the size of the input graph) is a proof-labeling scheme where the verification procedure performs  $t$  rounds, instead of just one round as in classical proof-labeling schemes. We may expect that proof-labeling schemes of radius  $t$  should help reduce the size of the certificates. This expectation is based on the intuition that the verification of classical (radius-1) proof-labeling schemes is done by comparing certificates of neighboring nodes or computing some function of them, and accept only if they are consistent with one another (in a sense that depends on the scheme). If the certificates are poorly correlated, then allowing more rounds for the verification should not be of much help as, with a  $k$ -bit certificate per node, the global proof has  $kn$  bits in total in  $n$ -node graphs, leaving little freedom for reorganizing the assignment of these  $kn$  bits to the  $n$  nodes. Perhaps surprisingly, we show that distributed proofs do not only involve partially redundant certificates, but inherently *highly redundant certificates*, which enables reducing their size a lot when more rounds are allowed. To capture this phenomenon, we say that a proof-labeling scheme *scales* with scaling factor  $f(t)$  if its size can be reduced by a factor  $\Omega(f(t))$  when using a  $t$ -round verification procedure; we say that the scheme *weakly scales* with scaling factor  $f(t)$  if the scaling factor is  $\tilde{\Omega}(f(t))$ , i.e.,  $\Omega(f(t)/\text{polylog } n)$  in  $n$ -node networks.

We prove that, in trees and other graph classes including e.g. grids, *all* proof-labeling schemes scale, with scaling factor  $t$  for  $t$ -round verification procedures. In other words, for every boolean predicate  $\mathcal{P}$  on labeled trees (that is, trees whose every node is assigned a label, i.e., a binary string), if  $\mathcal{P}$  has a proof-labeling scheme with certificates of  $k$  bits, for some  $k \geq 0$ , then  $\mathcal{P}$  has a proof-labeling scheme of radius  $t$  with certificates of  $O(k/t)$  bits, for all  $t \geq 1$ .

In addition, we prove that, in any graph, uniform parts of proof-labeling schemes weakly scale optimally. That is, for every boolean predicate  $\mathcal{P}$  on labeled graphs, if  $\mathcal{P}$  has a proof-labeling scheme such that  $k$  bits are identical in all certificates, then the part with these  $k$  bits weakly scales in an optimal manner: it can be reduced into  $\tilde{O}(k/b(t))$  bits by using a proof-labeling scheme of radius  $t$ , where  $b(t)$  denotes the size of the smallest ball of radius  $t$  in the actual graph. Therefore, in graphs whose neighborhoods increase polynomially, or even exponentially with their radius, the benefit in terms of space-complexity of using a proof-labeling scheme with radius  $t$  can be enormous. This result is of particular interest for the so-called *universal* proof-labeling scheme, in which every node is given the full  $n^2$ -bit adjacency matrix of the graph as part of its certificate, along with the  $O(\log n)$ -bit index of that node in the matrix.

We complement these general results by a collection of concrete results, regarding scaling classical boolean predicates on labeled graphs, including spanning tree, minimum-weight spanning tree, diameter, and additive spanners. For each of these predicates we prove tight upper and lower bounds on the certificate size of proof-labeling schemes of radius  $t$  on general graphs.

### 1.3 Our Techniques

Our proof-labeling schemes demonstrate that if we allow  $t$  rounds of verification, it is enough to keep only a small portion of the certificates, while all the rest are redundant. In a path, it is enough to keep only two consecutive certificates out of every  $t$ : two nodes with  $t-2$  missing certificates between them can try all the possible assignments for the missing certificates, and accept only if such an assignment exists. This reduces the *average* certificate size; to reduce the *maximal* size, we split the remaining certificates equally among the certificate-less nodes. This idea is extended to trees and grids, and is at the heart of the proof-labeling schemes presented in Section 3.

On general graphs, we cannot omit certificates from some nodes and let the others check all the options for missing certificates in a similar manner. This is because, for our approach to apply, the parts of missing certificates must be isolated by nodes with certificates. However, if all the certificates are essentially the same, as in the case of the universal scheme, we can simply keep each part of the certificate at some random node<sup>3</sup>, making sure that each node has all parts in its  $t$ -radius neighborhood. A similar, yet more involved idea, applies when the certificates are distances, e.g., when the predicate to check is the diameter, and the (optimal) certificate of a node contains in a distance-1 proof-labeling scheme its distances to all other nodes. While the certificates are not universal in this latter case, we show that it still suffices to randomly keep parts of the distances, such that on each path between two nodes, the distance between two certificates kept is at most  $t$ . These ideas are applied in Sections 4 and 5.

In order to prove lower bounds on the certificate size of proof-labeling schemes and on their scaling, we combine several known techniques in an innovative way. A classic lower bound technique for proof-labeling schemes is called *crossing*, but this cannot be used for lower bounds higher than logarithmic, and is not suitable for our model. A more powerful technique is the use of nondeterministic communication complexity [13, 25], which extends the technique used for the CONGEST model [1, 23]. In these bounds, the nodes are

---

<sup>3</sup> All our proof-labeling schemes are deterministic, but we use the probabilistic method for proving the existence of some of them.

partitioned between two players, who simulate the verification procedure in order to solve a communication complexity problem, and communicate whenever a message is sent over the edges of the cut between their nodes. When proving lower bounds for proof-labeling schemes, the nondeterminism is used to define the certificates: a nondeterministic string for a communication complexity problem can be understood as a certificate, and, when the players simulate verification on a graph, they interpret their nondeterministic strings as node certificates. However, this technique does not seem to be powerful enough to prove lower bounds for our model of multiple rounds verification. When splitting the nodes between the two players, the first round of verification only depends on the certificates of the nodes touching the cut, but arguing about the other verification rounds seems much harder. To overcome this problem, we use a different style of simulation argument, where the node partition is not fixed but evolves over time [14, 36]. More specifically, while there are sets of nodes which are simulated explicitly by either of the two players during the  $t$  rounds, the nodes in the paths connecting these sets are simulated in a decremental manner: both players start by simulating all these nodes, and then simulate less and less nodes as time passes. After the players communicate the certificates of the nodes along the paths at the beginning, they can simulate the verification process without any further communication. In this way, we are able to adapt some techniques used for the CONGEST model to our model, even though proof-labeling schemes are a computing model that is much more similar to the LOCAL model [35].

## 1.4 Previous Work

The mechanism considered in this paper for certifying distributed data structures and predicates on labeled graphs has at least three variants. The original *proof-labeling schemes*, as defined in [31], assume that nodes exchange solely their certificates between neighbors during the verification procedure. Instead, the variant called *locally checkable proofs* [25] imposes no restrictions on the type of information that can be exchanged between neighbors during the verification procedure. In fact, they can exchange their full individual states, which makes the design of lower bounds far more complex. This latter model is the one actually considered in this paper. There is a third variant, called *non-deterministic local decision* [19], which prevents using the actual identities of the nodes in the certificates. That is, the certificate must be oblivious to the actual identity assignment to the nodes. This latter mechanism is weaker than proof-labeling schemes and locally checkable proofs, as there are graph predicates that cannot be certified in this manner. However, all predicates on labeled graphs can be certified by allowing randomization [19], or by allowing just one alternation of quantifiers (the analog of  $\Pi_2$  in the polynomial hierarchy) [7]. A distributed variant of the centralized interactive proofs was recently introduced by Kol et al. [27].

Our work was inspired by [33], which aims at reducing the size of the certificates by trading time for space, i.e., allowing the verification procedure to take  $t$  rounds, for a non-constant  $t$ , in order to reduce the certificate size. They show a trade-off of this kind for example for proving the acyclicity of the input graph. The results in [30] were another source of inspiration, as it is shown that, by allowing  $O(\log^2 n)$  rounds of communication, one can verify MST using certificates of  $O(\log n)$  bits. In fact, [30] even describe an entire (non-silent) self-stabilizing algorithm for MST construction based on this mechanism for verifying MST.

In [17], the authors generalized the study of the class log-LCP introduced in [25], consisting of network properties verifiable with certificates of  $O(\log n)$  bits, to a whole local hierarchy inspired by the polynomial hierarchy. For instance, it is shown that MST is at the second level of that hierarchy, and that there are network properties outside the hierarchy. In [34], the

effect of sending different messages to different neighbors on the communication complexity of verification is analyzed. The impact of the number of errors on the ability to detect the illegality of a data structure w.r.t. a given predicate is studied in [16]. The notion of approximate proof-labeling schemes was investigated in [13], and the impact of randomization on communication complexity of verification has been studied in [9].

Finally, verification mechanisms a la proof-labeling schemes were used in other contexts, including the congested clique [28], wait-free computing [21], failure detectors [22], anonymous networks [18], and mobile computing [8,20]. For more references to work related to distributed verification, or distributed decision in general, see the survey [15]. To our knowledge, in addition to the aforementioned works [30,33], there is no prior work where verification time and certificate size are traded.

## 2 Model and Notations

A labeled graph is a pair  $(G, x)$  where  $G = (V, E)$  is a connected simple graph, and  $x : V \rightarrow \{0, 1\}^*$  is a function assigning a bit-string, called *label*, to every node of  $G$ . When discussing a weighted  $n$ -nodes graph  $G$ , we assume  $G = (V, E, w)$ , where  $w : E \rightarrow [1, n^c]$  for a fixed  $c \geq 1$ , and so  $w(e)$  can be encoded on  $O(\log n)$  bits. An identity-assignment to a graph  $G$  is an assignment  $\text{ID} : V \rightarrow [1, n^c]$ , for some fixed  $c \geq 1$ , of distinct identities to the nodes.

A distributed decision algorithm is an algorithm in which every node outputs accept or reject. We say that such an algorithm accepts if and only if every node outputs accept.

Given a finite collection  $\mathcal{G}$  of labeled graphs, we consider a boolean predicate  $\mathcal{P}$  on every labeled graph in  $\mathcal{G}$  (which may even depend on the identities assigned to the nodes). For instance, AUT is the predicate on graphs stating that there exists a non-trivial automorphism in the graph. Similarly, for any weighted graph with identity-assignment ID, the predicate MST on  $(G, x, \text{ID})$  states whether  $x(v) = \text{ID}(v')$  for some  $v' \in N[v]^4$  for every  $v \in V(G)$ , and whether the collection of edges  $\{\{v, x(v)\}, v \in V(G)\}$  forms a minimum-weight spanning tree of  $G$ . A proof-labeling scheme for a predicate  $\mathcal{P}$  is a pair  $(\mathbf{p}, \mathbf{v})$ , where

- $\mathbf{p}$ , called *prover*, is an oracle that assigns a bit-string called *certificate* to every node of every labeled graph  $(G, x) \in \mathcal{G}$ , potentially using the identities assigned to the nodes, and
- $\mathbf{v}$ , called *verifier*, is a distributed decision algorithm such that, for every  $(G, x) \in \mathcal{G}$ , and for every identity assignment ID to the nodes of  $G$ ,

$$\begin{cases} (G, x, \text{ID}) \text{ satisfies } \mathcal{P} & \implies \mathbf{v} \circ \mathbf{p}(G, x, \text{ID}) = \text{accept}; \\ (G, x, \text{ID}) \text{ does not satisfy } \mathcal{P} & \implies \text{for every prover } \mathbf{p}', \mathbf{v} \circ \mathbf{p}'(G, x, \text{ID}) = \text{reject}; \end{cases}$$

here,  $\mathbf{v} \circ \mathbf{p}$  is the output of the verifier  $\mathbf{v}$  on the certificates assigned to the nodes by  $\mathbf{p}$ . That is, if  $(G, x, \text{ID})$  satisfies  $\mathcal{P}$ , then, with the certificates assigned to the nodes by the prover  $\mathbf{p}$ , the verifier accepts at all nodes. Instead, if  $(G, x, \text{ID})$  does not satisfy  $\mathcal{P}$ , then, whatever certificates are assigned to the nodes, the verifier rejects in at least one node.

The *radius* of a proof-labeling scheme  $(\mathbf{p}, \mathbf{v})$  is defined as the maximum number of rounds of the verifier  $\mathbf{v}$  in the LOCAL model [35], over all identity-assignments to all the instances in  $\mathcal{G}$ , and all arbitrary certificates. It is denoted by  $\text{radius}(\mathbf{p}, \mathbf{v})$ . Often in this paper, the phrase proof-labeling scheme is abbreviated into PLS, while a proof-labeling scheme of radius  $t \geq 1$  is abbreviated into  $t$ -PLS. Note that, in a  $t$ -PLS, one can assume, w.l.o.g., that the verification procedure, which is given  $t$  as input to every node, proceeds at each node in two phases:

<sup>4</sup> In a graph,  $N(v)$  denotes the set of neighbors of node  $v$ , and  $N[v] = N(v) \cup \{v\}$ .



(1) collecting all the data (i.e., labels and certificates) from nodes at distance at most  $t$ , including the structure of the ball of radius  $t$  around that node, and (2) processing all the information for producing a verdict, either accept, or reject. Note that, while the examples in this paper are of highly uniform graphs, and thus the structure of the  $t$ -balls might be known to the nodes in advance, our scaling mechanisms work for arbitrary graphs.

Given an instance  $(G, x, \text{ID})$  satisfying  $\mathcal{P}$ , we denote by  $\mathbf{p}(G, x, \text{ID}, v)$  the certificate assigned by the prover  $\mathbf{p}$  to node  $v \in V$ , and by  $|\mathbf{p}(G, x, \text{ID}, v)|$  its size. We also let  $|\mathbf{p}(G, x, \text{ID})| = \max_{v \in V(G)} |\mathbf{p}(G, x, \text{ID}, v)|$ . The *certificate-size* of a proof-labeling scheme  $(\mathbf{p}, \mathbf{v})$  for  $\mathcal{P}$  in  $\mathcal{G}$ , denoted  $\text{size}(\mathbf{p}, \mathbf{v})$ , is defined as the maximum of  $|\mathbf{p}(G, x, \text{ID})|$ , taken over all instances  $(G, x, \text{ID})$  satisfying  $\mathcal{P}$ , where  $(G, x) \in \mathcal{G}$ . In the following, we focus on the graph families  $\mathcal{G}_n$  of connected simple graphs with  $n$  nodes,  $n \geq 1$ . That is, the size of a proof-labeling scheme is systematically expressed as a function of the number  $n$  of nodes. For the sake of simplifying the presentation, the graph family  $\mathcal{G}_n$  is omitted from the notations.

The minimum certificate size of a  $t$ -PLS for the predicate  $\mathcal{P}$  on  $n$ -node labeled graphs is denoted by  $\text{size-pls}(\mathcal{P}, t)$ , that is,

$$\text{size-pls}(\mathcal{P}, t) = \min_{\text{radius}(\mathbf{p}, \mathbf{v}) \leq t} \text{size}(\mathbf{p}, \mathbf{v}).$$

We also denote by  $\text{size-pls}(\mathcal{P})$  the size of a standard (radius-1) proof-labeling scheme for  $\mathcal{P}$ , that is,  $\text{size-pls}(\mathcal{P}) = \text{size-pls}(\mathcal{P}, 1)$ . For instance, it is known that  $\text{size-pls}(\text{MST}) = \Theta(\log^2 n)$  bits [29, 31], and that  $\text{size-pls}(\text{AUT}) = \tilde{\Omega}(n^2)$  bits [25]. More generally, for every decidable predicate  $\mathcal{P}$ , we have  $\text{size-pls}(\mathcal{P}) = O(n^2 + nk)$  bits [25] whenever the labels produced by  $x$  are of  $k$  bits, and  $\text{size-pls}(\mathcal{P}, D) = 0$  for graphs of diameter  $D$  because the verifier can gather all labels, and all edges at every node in  $D + 1$  rounds.

► **Definition 1.** Let  $\mathcal{I} \subseteq \mathbb{N}^+$ , and let  $f : \mathcal{I} \rightarrow \mathbb{N}^+$ . Let  $\mathcal{P}$  be a boolean predicate on labeled graphs. A set  $(\mathbf{p}_t, \mathbf{v}_t)_{t \in \mathcal{I}}$  of proof-labeling schemes for  $\mathcal{P}$ , with respective radius  $t \geq 1$ , *scales* with scaling factor  $f$  on  $\mathcal{I}$  if  $\text{size}(\mathbf{p}_t, \mathbf{v}_t) = O(\frac{1}{f(t)} \cdot \text{size-pls}(\mathcal{P}))$  bits for every  $t \in \mathcal{I}$ . Also,  $(\mathbf{p}_t, \mathbf{v}_t)_{t \in \mathcal{I}}$  *weakly scales* with scaling factor  $f$  on  $\mathcal{I}$  if  $\text{size}(\mathbf{p}_t, \mathbf{v}_t) = \tilde{O}(\frac{1}{f(t)} \cdot \text{size-pls}(\mathcal{P}))$  bits for every  $t \in \mathcal{I}$ .

In the following, somewhat abusing terminology, we shall say that a proof-labeling scheme (weakly) scales while, formally, it should be a set of proof-labeling schemes that scales.

► **Remark.** At first glance, it may seem that no proof-labeling schemes can scale more than linearly, i.e., one may be tempted to claim that for every predicate  $\mathcal{P}$  we have  $\text{size-pls}(\mathcal{P}, t) = \Omega(\frac{1}{t} \cdot \text{size-pls}(\mathcal{P}))$ . The rationale for such a claim is that, given a proof-labeling scheme  $(\mathbf{p}_t, \mathbf{v}_t)$  for  $\mathcal{P}$ , with radius  $t$  and  $\text{size-pls}(\mathcal{P}, t)$ , one can construct a proof-labeling scheme  $(\mathbf{p}, \mathbf{v})$  for  $\mathcal{P}$  with radius 1 as follows: the certificate of every node  $v$  is the collection of certificates assigned by  $\mathbf{p}_t$  to the nodes in the ball of radius  $t$  centered at  $v$ ; the verifier  $\mathbf{v}$  then simulates the execution of  $\mathbf{v}_t$  on these certificates. In paths or cycles, the certificates resulting from this construction are of size  $O(t \cdot \text{size-pls}(\mathcal{P}, t))$ , from which it follows that no proof-labeling scheme can scale more than linearly. There are several flaws in this reasoning, which make it actually erroneous. First, it might be the case that degree-2 graphs are not the worst case graphs for the predicate  $\mathcal{P}$ ; that is, the fact that  $(\mathbf{p}, \mathbf{v})$  induces certificates of size  $O(t)$  times the certificate size of  $(\mathbf{p}_t, \mathbf{v}_t)$  in such graphs may be uncorrelated to the size of the certificates of these proof-labeling schemes in worst case instances. Second, in  $t$  rounds of verification every node learns not only the certificates of its  $t$ -neighborhood, but also its structure, which may contain valuable information for the verification; this idea stands out when the lower bounds for  $\text{size-pls}(\mathcal{P})$  are established using labeled graphs of constant diameter, in which

case there is no room for studying how proof-labeling schemes can scale. The take away message is that establishing lower bounds of the type  $\text{size-pls}(\mathcal{P}, t) = \Omega(\frac{1}{t} \cdot \text{size-pls}(\mathcal{P}))$  for  $t$  within some non-trivial interval requires specific proofs, which often depend on the given predicate  $\mathcal{P}$ .

**Communication Complexity.** In the set-disjointness (DISJ) problem on  $k$  bits, each of the two players Alice and Bob is given a  $k$ -bit string, denoted  $S_A$  and  $S_B$  respectively. They aim at deciding whether  $S_A \cap S_B = \emptyset$ , i.e. whether there does not exist  $i \in \{1, \dots, k\}$  such that  $S_A[i] = S_B[i] = 1$ . We consider nondeterministic protocols for the problem, i.e. protocols where the players also get an auxiliary string from an oracle that knows both inputs, and they may use it in order to verify that their inputs are disjoint. The communication complexity of a nondeterministic protocol for DISJ is the number of bits the players exchange on two input strings that are disjoint, in the worst case, when they are given optimal nondeterministic strings. The nondeterministic communication complexity of DISJ is the minimum, among all nondeterministic protocols for DISJ, of the communication complexity of that protocol. The nondeterministic communication complexity of DISJ is  $\Omega(k)$  (e.g., as a consequence of Example 1.23 and Definition 2.3 in [32]).

### 3 All Proof-Labeling Schemes Scale Linearly in Trees

This section is entirely dedicated to the proof of one of our main results, stating that *every* predicate on labeled trees has a proof that scales linearly. Further in the section, we also show how to extend this result to cycles and to grids, and, more generally, to multi-dimensional grids and toruses.

► **Theorem 2.** *Let  $\mathcal{P}$  be a predicate on labeled trees, and let us assume that there exists a (distance-1) proof-labeling scheme  $(\mathbf{p}, \mathbf{v})$  for  $\mathcal{P}$ , with  $\text{size}(\mathbf{p}, \mathbf{v}) = k$ . Then there exists a proof-labeling scheme for  $\mathcal{P}$  that scales linearly, that is,  $\text{size-pls}(\mathcal{P}, t) = O(\frac{k}{t})$ .*

The rest of this subsection is dedicated to the proof of Theorem 2. So, let  $\mathcal{P}$  be a predicate on labeled trees, and let  $(\mathbf{p}, \mathbf{v})$  be a proof-labeling scheme for  $\mathcal{P}$  with  $\text{size}(\mathbf{p}, \mathbf{v}) = k$ . First, note that we can restrict attention to trees with diameter  $> t$ . Indeed, predicates on labeled trees with diameter  $\leq t$  are easy to verify since every node can gather the input of the entire tree in  $t$  rounds. More precisely, if we have a scheme that works for trees with diameter  $> t$ , then we can trivially design a scheme that applies to all trees, by adding a single bit to the certificates, indicating whether the tree is of diameter at most  $t$  or not.

The setting of the certificates in our scaling scheme is based on a specific decomposition of the given tree  $T$ . Let  $T$  be a tree of diameter  $> t$ , and let  $h = \lfloor t/2 \rfloor$ . For assigning the certificates, the tree  $T$  is rooted at some node  $r$ . A node  $u$  such that  $\text{dist}_T(r, u) \equiv 0 \pmod{h}$ , and  $u$  possesses a subtree of depth at least  $h - 1$  is called a *border* node. Similarly, a node  $u$  such that  $\text{dist}_T(r, u) \equiv -1 \pmod{h}$ , and  $u$  possesses a subtree of depth at least  $h - 1$  is called an *extra-border* node. A node that is a border or an extra-border node is called a *special* node. All other nodes are *standard* nodes. For every border node  $v$ , we define the *domain* of  $v$  as the set of nodes in the subtree rooted at  $v$  but not in subtrees rooted at border nodes that are descendants of  $v$ . The proof of the following lemma is omitted from this extended abstract.

► **Lemma 3.** *The domains form a partition of the nodes in the tree  $T$ , every domain forms a tree rooted at a border node, with depth in the range  $[h - 1, 2h - 1]$ , and two adjacent nodes of  $T$  are in different domains if and only if they are both special.*



The certificates of the distance- $t$  proof-labeling scheme contain a 2-bit field indicating to each node whether it is a root, border, extra-border, or standard node. Let us show that this part of the certificate can be verified in  $t$  rounds. The prover orients the edges of the tree towards the root  $r$ . It is well-known that such an orientation can be given to the edges of a tree by assigning to each node its distance to the root, modulo 3. These distances can obviously be checked locally, in just one round. So, in the remaining of the proof, we assume that the nodes are given this orientation upward the tree. The following lemma (whose proof is omitted) shows that the decomposition into border, extra-border, and standard nodes can be checked in  $t$  rounds.

► **Lemma 4.** *Given a set of nodes marked as border, extra-border, or standard in an oriented tree, there is a verification protocol that checks whether that marking corresponds to a tree decomposition such as the one described above, in  $2h < t$  rounds.*

We are now ready to describe the distance- $t$  proof-labeling scheme. From the previous discussions, we can assume that the nodes are correctly marked as root, border, extra-border, and standard, with a consistent orientation of the edges towards the root. We are considering the given predicate  $\mathcal{P}$  on labeled trees, with its proof-labeling scheme  $(\mathbf{p}, \mathbf{v})$  using certificates of size  $k$  bits. Before reducing the size of the certificates to  $O(k/t)$  by communicating at distance  $t$ , we describe a proof-labeling scheme at distance  $t$  which still uses large certificates, of size  $O(k)$ , but stored at a few nodes only, with all other nodes storing no certificates.

► **Lemma 5.** *There exists a distance- $t$  proof-labeling scheme for  $\mathcal{P}$ , in which the prover assigns certificates to special nodes only, and these certificates have size  $O(k)$ .*

**Sketch of proof.** On legally labeled trees, the prover provides every special node (i.e., every border or extra-border node) with the same certificate as the one provided by  $\mathbf{p}$ . All other nodes are provided with no certificates. On arbitrary labeled trees, the verifier is active at border nodes only, and all non-border nodes systematically accept (in zero rounds). At a border node  $v$ , the verifier first gathers all information at distance  $2h$ . This includes all the labels of the nodes in its domain, and of the nodes that are neighbors of some node in its domain. Then  $v$  checks whether there exists an assignment of  $k$ -bit certificates to the standard nodes in its domain that results in  $\mathbf{v}$  accepting at every node in its domain. If this is the case, then  $v$  accepts, else it rejects. Since the standard nodes form non-overlapping regions well separated by the border and extra-border nodes, this results in a correct distance- $t$  proof-labeling scheme. ◀

We now show how to spread out the certificates of the border and extra-border nodes to obtain smaller certificates. The following lemma is the main tool for doing so. As this lemma is also used further in the paper, we provide a generalized version of its statement, and we later show how to adapt it to the setting of the current proof.

We say that a local algorithm  $\mathcal{A}$  *recovers* an assignment of certificates provided by some prover  $\mathbf{q}$  from an assignment of certificates provided by another prover  $\mathbf{q}'$  if, given the certificates assigned by  $\mathbf{q}'$  as input to the nodes,  $\mathcal{A}$  allows every node to reconstruct the certificate that would have been assigned to it by  $\mathbf{q}$ . We define a *special* prover as a prover that assigns certificates only to the special nodes, while all other nodes are given empty certificates.

► **Lemma 6.** *There is a local algorithm  $\mathcal{A}$  satisfying the following. For every  $s \geq 1$ , for every oriented marked tree  $T$  of depth at least  $s$ , and for every assignment of  $b$ -bit certificates provided by some special prover  $\mathbf{q}$  to the nodes of  $T$ , there exists an assignment of  $O(b/s)$ -bit certificates provided by a prover  $\mathbf{q}'$  to the nodes of  $T$  such that  $\mathcal{A}$  recovers  $\mathbf{q}$  from  $\mathbf{q}'$  in  $s$  rounds.*

**Sketch of proof.** The prover  $\mathbf{q}'$  spreads the certificate assigned to each border node  $v$  along a path starting from  $v$ , of length  $s - 1$ , going downward the tree. The algorithm  $\mathcal{A}$  gathers the certificates spread along these paths. ◀

**Proof of Theorem 2.** In the distance- $t$  proof-labeling scheme, the prover chooses a root and an orientation of the tree  $T$ , and provides every node with a counter modulo 3 in its certificate allowing the nodes to check the consistency of the orientation. Then the prover constructs a tree decomposition of the rooted tree, and provides every node with its type (root, border, extra-border, or standard) in its certificates. Applying Lemmas 5 and 6, the prover spreads the certificates assigned to the special nodes by  $\mathbf{p}$ . Every node will get at most two parts, because only the paths associated to a border node and to its parent (an extra-border node) can intersect. Overall, the certificates have size  $O(k/h) = O(k/t)$ . The verifier checks the orientation and the marking, then recovers the certificates of the special nodes, as in Lemma 6, and performs the simulation as in Lemma 5. This verification can be done with radius  $t \leq 2h$ , yielding the desired distance- $t$  proof labeling scheme. ◀

**Linear scaling in cycles and grids.** For the proof techniques of Theorem 2 to apply to other graphs, we need to compute a partition of the nodes into the two categories, special and standard, satisfying three main properties. First, the partition should split the graph into regions formed by standard nodes, separated by special nodes. Second, each region should have a diameter small enough for allowing special nodes at the border of the region to simulate the standard nodes in that region, as in Lemma 5. Third, the regions should have a diameter large enough to allow efficient spreading of certificates assigned to special nodes over the standard nodes, as in Lemma 6. For any graph family in which one can define such a decomposition, an analogue of Theorem 2 holds. We show that this is the case for cycles and grids (the proof is omitted).

► **Corollary 7.** *Let  $\mathcal{P}$  be a predicate on labeled cycles, and let us assume that there exists a (distance-1) proof-labeling scheme  $(\mathbf{p}, \mathbf{v})$  for  $\mathcal{P}$  with  $\text{size}(\mathbf{p}, \mathbf{v}) = k$ . Then there exists a proof-labeling scheme for  $\mathcal{P}$  that scales linearly, that is,  $\text{size-pls}(\mathcal{P}, t) = O\left(\frac{k}{t}\right)$ . The same holds for predicates on 2-dimensional labeled grids.*

By the same techniques, Corollary 7 can be generalized to toroidal 2-dimensional labeled grids, as well as to  $d$ -dimensional labeled grids and toruses, for every  $d \geq 2$ .

## 4 Universal Scaling of Uniform Proof-Labeling Schemes

It is known [33] that, for every predicate  $\mathcal{P}$  on labeled graphs with  $\text{size-pls}(\mathcal{P}) = \tilde{\Omega}(n^2)$ , there is a proof-labeling scheme that scales linearly on the interval  $[1, D]$  in graphs of diameter  $D$ . We show that, in fact, the scaling factor can be much larger. We say that a graph  $G = (V, E)$  has *growth*  $b = b(t)$  if, for every  $v \in V$  and  $t \in [1, D]$ , we have that  $|B_G(v, t)| \geq b(t)$ . We say that a proof-labeling scheme is *uniform* if the same certificate is assigned to all nodes by the prover.

► **Theorem 8.** *Let  $\mathcal{P}$  be a predicate on labeled graphs, fix a uniform 1-PLS  $(\mathbf{p}, \mathbf{v})$  for  $\mathcal{P}$  and denote  $k = \text{size}(\mathbf{p}, \mathbf{v})$ . Then there is a proof-labeling scheme for  $\mathcal{P}$  that weakly scales with scaling factor  $b(t)$  on graphs of growth  $b(t)$ . More specifically, let  $G$  be a graph, let  $t_0 = \min\{t \mid b(t) \geq \log n\}$ , and  $t_1 = \max\{t \mid k \geq b(t)\}$ . Then, in  $G$ , for every  $t \in [t_0, t_1]$ ,  $\text{size-pls}(\mathcal{P}, t) = \tilde{O}\left(\frac{k}{b(t)}\right)$ .*

**Proof.** Let  $s = (s_1, \dots, s_k)$ , where  $s_i \in \{0, 1\}$  for every  $i = 1, \dots, k$ , be the  $k$ -bit certificate assigned to every node of  $G$ . Let  $t \geq 1$  be such that  $k \geq b(t) \geq c \log n$  for a constant  $c$  large enough. For every node  $v \in V$ , set the certificate of  $v$ , denoted  $s^{(v)}$ , as follows: for every  $i = 1, \dots, k$ ,  $v$  stores the pair  $(i, s_i)$  in  $s^{(v)}$  with probability  $\frac{c \log n}{b(t)}$ . Recall the following Chernoff bounds: Suppose  $Z_1, \dots, Z_m$  are independent random variables taking values in  $\{0, 1\}$ , and let  $Z = \sum_{i=1}^m Z_i$ . For every  $0 \leq \delta \leq 1$ , we have  $\Pr[Z \leq (1 - \delta)\mathbb{E}Z] \leq e^{-\frac{1}{2}\delta^2\mathbb{E}Z}$ , and  $\Pr[Z \geq (1 + \delta)\mathbb{E}Z] \leq e^{-\frac{1}{3}\delta^2\mathbb{E}Z}$ .

- On the one hand, for every  $v \in V$ , let  $X_v$  be the random variable equal to the number of pairs stored in  $s^{(v)}$ . By a Chernoff bound, we have  $\Pr[X_v \geq \frac{2ck \log n}{b(t)}] \leq e^{-\frac{c \log n}{3b(t)}} = n^{-\frac{c}{3b(t)}}$ . Therefore, by union bound, the probability that a node  $v$  stores more than  $\frac{2ck \log n}{b(t)}$  pairs  $(i, s_i)$  is at most  $n^{1 - \frac{c}{3b(t)}}$ , which is less than  $\frac{1}{2}$  for  $c$  large enough.
- On the other hand, for every  $v \in V$ , and every  $i = 1, \dots, k$ , let  $Y_{v,i}$  be the number of occurrences of the pair  $(i, s_i)$  in the ball of radius  $t$  centered at  $v$ . By a Chernoff bound, we have  $\Pr[Y_{v,i} \leq \frac{1}{2}c \log n] \leq e^{-\frac{c \log n}{8}} = n^{-c/8}$ . Therefore, by union bound, the probability that there exists a node  $v \in V$ , and an index  $i \in \{1, \dots, k\}$  such that none of the nodes in the ball of radius  $t$  centered at  $v$  store the pair  $(i, s_i)$  is at most  $kn^{1-c/8}$ , which is less than  $\frac{1}{2}$  for  $c$  large enough.

It follows that, for  $c$  large enough, the probability that no node stores more than  $\tilde{O}(k/b(t))$  pairs  $(i, s_i)$ , and every pair  $(i, s_i)$  is stored in at least one node of each ball of radius  $t$ , is positive. Therefore, there is a way for a prover to distribute the pairs  $(i, s_i)$ ,  $i = 1, \dots, k$ , to the nodes such that (1) no node stores more than  $\tilde{O}(k/b(t))$  bits, and (2) every pair  $(i, s_i)$  appears at least once in every  $t$ -neighborhood of each node. At each node  $v$ , the verification procedure first collects all pairs  $(i, s_i)$  in the  $t$ -neighborhood of  $v$ , in order to recover  $s$ , and then runs the verifier of the original (distance-1) proof-labeling scheme.

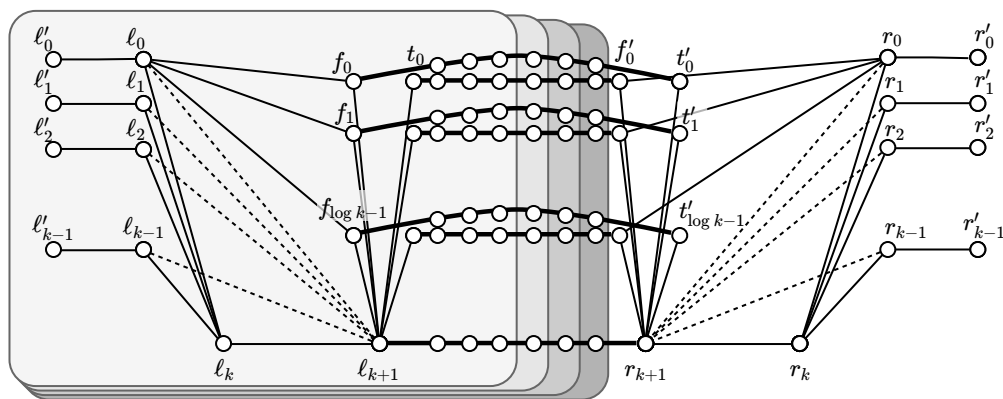
Finally, we emphasize that we only use probabilistic arguments as a way to prove the existence of certificate assignment, but the resulting proof-labeling scheme is deterministic and its correctness is not probabilistic. ◀

Theorem 8 finds direct application to the *universal* proof-labeling scheme [25, 31], which uses  $O(n^2 + kn)$  bits in  $n$ -node graphs labeled with  $k$ -bit labels. The certificate of each node consists of the  $n \times n$  adjacency matrix of the graph, an array of  $n$  entries each equals to the  $k$ -bit label at the corresponding node, and an array of  $n$  entries listing the identities of the  $n$  nodes. It was proved in [33] that the universal proof-labeling scheme can be scaled by a factor  $t$ . Theorem 8 significantly improves that result, by showing that the universal proof-labeling scheme can actually be scaled by a factor  $b(t)$ , which can be exponential in  $t$ .

► **Corollary 9.** *For every predicate  $\mathcal{P}$  on labeled graphs, there is a proof-labeling scheme for  $\mathcal{P}$  as follows. For every graph  $G$  with growth  $b(t)$ , let  $t_0 = \min\{t \mid b(t) \geq \log n\}$ . Then, for every  $t \geq t_0$  we have  $\text{size-pls}(\mathcal{P}, t) = \tilde{O}\left(\frac{n^2 + kn}{b(t)}\right)$ .*

Theorem 8 is also applicable to proof-labeling scheme where the certificates have the same sub-certificate assigned to all nodes; in this case, the size of this common sub-certificate can be drastically reduced by using a  $t$ -round verification procedure. This is particularly interesting when the size of the common sub-certificate is large compared to the size of the rest of the certificates. An example of such a scheme is in essence the one described in [31, Corollary 2.4] for  $\text{ISO}_k$ . Given a parameter  $k \in \Omega(\log n)$ , let  $\text{ISO}_k$  be the predicate on graph stating that there exist two vertex-disjoint isomorphic induced subgraphs of size  $k$  in the given graph. The proof of the next corollary appears in the full version of our paper.





■ **Figure 2** The lower bound graph construction for  $t = 3$ , and the sets of nodes simulated by Alice in the three rounds of verification (from dark gray to lighter gray). Alice eventually knows the outputs of all the nodes in the light-most gray shaded set.

paths also between  $\ell_{k+1}$  and all  $t_h \in T$  and  $f_h \in F$ , and between  $r_{k+1}$  and all  $t'_h \in T'$  and  $f'_h \in F'$ . Connect by a  $P$ -path each  $\ell_i \in L$  with the nodes representing its binary encoding, that is, connect  $\ell_i$  to each  $t_h$  that satisfies  $i[h] = 1$ , and to each  $f_h$  that satisfies  $i[h] = 0$ , where  $i[h]$  is bit  $h$  of the binary encoding of  $i$ . Add similar paths between each  $r_i \in R$  and its encoding by nodes  $t'_h$  and  $f'_h$ . In addition, for each  $0 \leq h \leq \log k - 1$ , add a  $(2t + 1)$ -path from  $t_h$  to  $f'_h$  and from  $f_h$  to  $t'_h$ , and a similar path from  $\ell_{k+1}$  to  $r_{k+1}$ .

Assume Alice and Bob want to solve the DISJ problem for two  $k$ -bit strings  $S_A$  and  $S_B$  using a non-deterministic protocol. They build the graph described above, and add the following edges:  $(\ell_i, \ell_{k+1})$  whenever  $S_A[i] = 0$ , and  $(r_i, r_{k+1})$  whenever  $S_B[i] = 0$ . The next claim is at the heart of our proof.

► **Claim 12.** *If  $S_A$  and  $S_B$  are disjoint then  $D = 4P + 2t + 2$ , and otherwise  $D \geq 6P + 2t + 1$ .*

The proof of this claim follows similar lines of the proof of [1, Lemma 2], and appears in the full version of our paper. We can now prove the lower bound from Theorem 11.

**Proof of lower bound from Theorem 11.** Fix  $t \in [1, n/\log n]$ , and let  $S_A$  and  $S_B$  be two input strings for the DISJ problem on  $k$  bits. We show how Alice and Bob can solve DISJ on  $S_A$  and  $S_B$  in a nondeterministic manner, using the graph described above and a  $t$ -PLS for  $\text{DIAM} = 4P + 2t + 2$ .

Alice and Bob simulate the verifier on the labeled graph (see Figure 2). The nodes simulated by Alice, denoted  $A$ , are  $L \cup L' \cup T \cup F \cup \{\ell_k, \ell_{k+1}\}$  and all the paths between them, and by Bob, denoted  $B$ , are  $R \cup R' \cup T' \cup F' \cup \{r_k, r_{k+1}\}$  and the paths between them. For each pair of nodes  $(a, b) \in A \times B$  that are connected by a  $(2t + 1)$ -path, let  $P_{ab}$  be this path, and  $\{P_{ab}(i)\}$ ,  $i = 0, \dots, 2t + 1$  be its nodes in consecutive order, where  $P_{ab}(0) = a$  and  $P_{ab}(2t + 1) = b$ . Let  $C$  be the set of all  $(2t + 1)$ -path nodes, i.e.  $C = V \setminus (A \cup B)$ . The nodes in  $C$  are simulated by both players, in a decremental way described below.

Alice interprets her nondeterministic string as the certificates given to the nodes in  $A \cup C$ , and she sends the certificates of  $C$  to Bob. Bob interprets his nondeterministic string as the certificates of  $B$ , and gets the certificates of  $C$  from Alice. They simulate the verifier execution for  $t$  rounds, where, in round  $r = 1, \dots, t$ , Alice simulates the nodes of  $A$  and all nodes  $P_{ab}(i)$  with  $(a, b) \in A \times B$  and  $i \leq 2t + 1 - r$ , while Bob simulates the nodes of  $B$  and all nodes  $P_{ab}(i)$  with  $i \geq r$ .

Note that this simulation is possible without further communication. The initial state of nodes in  $A$  is determined by  $S_A$ , the initial state of the nodes  $P_{ab}(i)$  with  $i \leq 2t$  is independent of the inputs, and the certificates of both node sets are encoded in the nondeterministic string of Alice. In each round of verification, all nodes whose states may depend on the input of Bob or on his nondeterministic string are omitted from Alice's simulation, and so she can continue the simulation without communication with Bob. Similar arguments apply to the nodes simulated by Bob. Finally, each node is simulated for  $t$  rounds by at least one of the players. Thus, if the verifier rejects, that is, at least one node rejects, then at least one of the players knows about this rejection.

Using this simulation, Alice and Bob can determine whether DISJ on  $(S_A, S_B)$  is true as, from Claim 12, we know that if it is true then  $\text{DIAM} = 4P + 2t + 2$ , and the verifier of the PLS accepts, while otherwise it rejects. The nondeterministic communication complexity of the true case of DISJ on  $k$ -bit strings is  $\Omega(k) = \Omega(n)$ , so Alice and Bob must communicate this amount of bits. From the graph definition,  $|C| = \Theta(t \log n)$  which implies  $\text{size-pls}(\text{DIAM}, t) = \Omega\left(\frac{n}{t \log n}\right)$ , as desired. ◀

Let  $k$  be a non-negative integer. For any labeled graph  $(G, x)$ ,  $k$ -SPANNER is the predicate on  $(G, x)$  that states whether the collection of edges  $E_H = \{\{v, w\}, v \in V(G), w \in x(v)\}$  forms a  $k$ -additive spanner of  $G$ , i.e., a subgraph  $H$  of  $G$  such that, for every two nodes  $s, t$ , we have  $\text{dist}_H(s, t) \leq \text{dist}_G(s, t) + k$ . There is a proof-labeling scheme for additive-spanner that weakly scales linearly, or more precisely,  $\text{size-pls}(k\text{-SPANNER}, t) = \tilde{\Theta}\left(\frac{n}{t}\right)$  for any constant  $k$  and  $t \in [1, n/\log n]$ . In the full version of our paper we prove this result, its optimality, as well as slightly weaker results for general spanners.

## 6 Distributed Proofs for Spanning Trees

In this section, we study two specific problems which are classical in the domain of proof-labeling schemes: the verification of a spanning tree, and of a minimum-weight spanning tree. The predicates ST and MST are the sets of labeled graphs where some edges are marked and these edges form a spanning tree, and a minimum spanning tree, respectively. For these predicates, we present proof-labeling schemes that scale linearly in  $t$ . Note that ST and MST are problems on general labeled graphs and not on trees, i.e., the results in this section improve upon Section 4 (for these specific problems), and are incomparable with the results of Section 3.

Formally, let  $\mathcal{F}$  be the family of all connected undirected, weighted, labeled graphs  $(G, x)$ . Each label  $x(v)$  contains a (possibly empty) subset of edges adjacent to  $v$ , which is consistent with the neighbors of  $v$ , and we denote the collection of edges represented in  $x$  by  $T_x$ . In the ST (respectively, MST) problem, the goal is to decide for every labeled graph  $(G, x) \in \mathcal{F}$  whether  $T_x$  is a spanning tree of  $G$  (respectively, whether  $T_x$  is a spanning tree of  $G$  with the sum of all its edge-weights minimal among all spanning trees of  $G$ ). For these problems we have the following results.

► **Theorem 13.** *For every  $t \in O(\log n)$ , we have that  $\text{size-pls}(\text{ST}, t) = O\left(\frac{\log n}{t}\right)$ .*

**Proof sketch.** To prove that a marked subgraph  $T_x$  is a spanning tree, we verify it has the following properties: (1) spanning the graph, (2) acyclic, (3) connected. The first property is local – every node verifies that it has at least one incident marked edge. For the second property, we use the  $t$ -distance proof-labeling scheme for acyclicity designed by Ostrovsky et al. [33, Theorem 8], where oriented trees are verified and every root knows that it is a root, using  $O(\log n/t)$ -bit certificates. Finally, we use Theorem 2 within the tree in order to split the root ID; the nodes then verify they all agree on the root, which implies connectivity. ◀



► **Theorem 14.** *For every  $t \in O(\log n)$ , we have that  $\text{size-pls}(\text{MST}, t) = O\left(\frac{\log^2 n}{t}\right)$ .*

Our theorem only applies for  $t \in O(\log n)$ , meaning that we can get from proofs of size  $O(\log^2 n)$  to proofs of size  $O(\log n)$ , but not to a constant. For the specific case  $t = \Theta(\log n)$ , our upper bound matches the lower bound of Korman et al. [30, Corollary 3]. In the same paper, the authors also present an  $O(\log^2 n)$ -round verification scheme for MST using  $O(\log n)$  bits of memory at each node (both for certificates and for local computation). Removing the restriction of  $O(\log n)$ -bit memory for local computation, one may derive an  $O(\log n)$ -round verification scheme with  $O(\log n)$  proof size out of the aforementioned  $O(\log^2 n)$ -round scheme, which matches our result for  $t = \Theta(\log n)$ . The improvement we present is two-folded: our scheme is scalable for different values of  $t$  (as opposed to schemes for only  $t = 1$  and  $t = \Theta(\log n)$ ), and our construction is much simpler, as described next.

Our upper bound is based on a famous 1-round PLS for MST [29, 30], which in turn builds upon the algorithm of Gallager, Humblet, and Spira (GHS) [24] for a distributed construction of an MST. The idea behind this scheme is, given a labeled graph  $(G, x)$ , to verify that  $T_x$  is consistent with an execution of the GHS algorithm in  $G$ .

The GHS algorithm maintains a spanning forest that is a subgraph of the minimum spanning tree, i.e., the trees of the forest are fragments of the desired minimum spanning tree. The algorithm starts with a spanning forest consisting of all nodes and no edges. At each phase each of the fragments adds the minimum-weight edge going out of it, thus merging several fragments into one. After  $O(\log n)$  iterations, all the fragments are merged into a single component, which is the desired minimum-weight spanning tree. We show that each phase can be verified with  $O(\log n/t)$  bits, giving a total complexity of  $O(\log^2 n/t)$  bits.

The GHS algorithm assumes distinct edge weights, which implies a unique minimum-weight spanning tree and a unique (synchronous) execution of the algorithm. The case of non-unique edge weights is easily resolved in the algorithm by any consistent tie-breaking (e.g., using node IDs); handling non-unique edge weights in verification is not as easy, since the tie-breaking mechanism must result in the specified spanning tree. Theorem 14 is true without the assumption of distinct edge weights, but we prove it here only under this assumption, and leave the proof of the general case to the full version of our paper.

**Proof of Theorem 14.** Let  $(G, x)$  be a labeled graph such that  $T_x$  is a minimum-weight spanning tree. If  $t$  is greater than the diameter  $D$  of  $G$ , every node can see the entire labeled graph in the verification process, and we are done; we henceforth assume  $t \leq D$ . The certificates consist of four parts.

First, we choose a root and orient the edges of  $T_x$  towards it. We give each node its distance from the root modulo 3, which allows it to obtain the ID of its parent and the edge pointing to it in one round. Second, we assign the certificate described above for ST (Theorem 13), which certifies that  $T_x$  is indeed a spanning tree. This uses  $O(\log n/t)$  bits.

The third part of the certificate tells each node the phase in which the edge connecting it to its parent is added to the tree in the GHS algorithm, and which of the edge's endpoints added it to the tree. Note that after one round of verification, each node knows for every incident edge, at which phase it is added to the spanning tree, and by which of its endpoints. This part uses  $O(\log \log n)$  bits.

The fourth part of the certificate consists of  $O(\log^2 n/t)$  bits,  $O(\log n/t)$  for each of the  $O(\log n)$  phases of the GHS algorithm. To define the part of a certificate of every phase, fix a phase, a fragment  $F$  in the beginning of this phase, and let  $e = (u, v)$  be the minimum-weight edge going out of  $F$ , where  $u \in F$  and  $v \notin F$ . Our goal is that the nodes of  $F$  verify together that  $e$  is the minimum-weight outgoing edge of  $F$ , and that no other edge was added by  $F$  in

this phase. To this end, we first orient the edges of  $F$  towards  $u$ , i.e. set  $u$  as the root of  $F$ . If the depth of  $F$  is less than  $t$ , then in  $t - 1$  rounds the root  $u$  can see all of  $F$  and check that  $(u, v)$  is the lightest outgoing edge. All other nodes just have to verify that no other edge is added by the nodes of  $F$  in this phase. Otherwise, if the depth of  $F$  is at least  $t$ , by Theorem 2, the information about  $ID(u)$  and  $w(e)$  can be spread on  $F$  such that in  $t$  rounds it can be collected by all nodes of  $F$ . With this information known to all the nodes of  $F$ , the root can locally verify that it is named as the node that adds the edge and that it has the named edge with the right weight. The other nodes of  $F$  can locally verify that they do not have incident outgoing edges with smaller weights, and that no other edge is added by  $F$ .

Overall, our scheme verifies that  $T_x$  is a spanning tree, and that it is consistent with every phase of the GHS algorithm. Therefore, the scheme accepts  $(G, x)$  if and only if  $T_x$  is a minimum spanning tree. ◀

## 7 Conclusion

We have proved that, for many classical boolean predicates on labeled graphs (including MST), there are proof-labeling schemes that linearly scale with the radius of the scheme, i.e., the number of rounds of the verification procedure. More generally, we have shown that for every boolean predicate on labeled trees, cycles and grids, there is a proof-labeling scheme that scales linearly with the radius of the scheme. This yields the following question:

► **Open Problem 1.** *Prove or disprove that, for every predicate  $\mathcal{P}$  on labeled graphs, there is a proof-labeling scheme for  $\mathcal{P}$  that (weakly) scales linearly.*

In fact, the scaling factor might even be larger than  $t$ , and be as large as  $b(t)$  in graphs with ball growth  $b$ . We have proved that the uniform part of any proof-labeling scheme can be scaled by such a factor  $b(t)$  for  $t$ -PLS. This yields the following stronger open problem:

► **Open Problem 2.** *Prove or disprove that, for every predicate  $\mathcal{P}$  on labeled graphs, there is a proof-labeling scheme for  $\mathcal{P}$  that scales with factor  $\Omega(b)$  in graphs with ball growth  $b$ .*

We are tempted to conjecture that the answer to the first problem is positive (as it holds for trees and cycles). However, we believe that the answer to the second problem might well be negative. In particular, it seems challenging to design a proof-labeling scheme for DIAM that would scale with the size of the balls. Indeed, checking diameter is strongly related to checking shortest paths in the graph, and this restricts significantly the way the certificates can be redistributed among nodes in a ball of radius  $t$ . Yet, there might be some other way to certify DIAM, so we let the following as an open problem:

► **Open Problem 3.** *Is there a proof-labeling scheme for DIAM that scales by a factor greater than  $t$  in all graphs where  $b(t) \gg t$ ?*

---

## References

- 1 Amir Abboud, Keren Censor-Hillel, and Seri Khoury. Near-linear lower bounds for distributed distance computations, even in sparse networks. In *30th Int. Symposium on Distributed Computing (DISC)*, pages 29–42, 2016. Full version at arXiv:1605.05109.
- 2 Yehuda Afek and Shlomi Dolev. Local stabilizer. *J. Parallel Distrib. Comput.*, 62(5):745–765, 2002. doi:10.1006/jpdc.2001.1823.
- 3 Yehuda Afek, Shay Kutten, and Moti Yung. The local detection paradigm and its applications to self-stabilization. *Theoretical Computer Science*, 186(1):199–229, 1997. doi:10.1016/S0304-3975(96)00286-1.




- 4 Heger Arfaoui, Pierre Fraigniaud, David Ilcinkas, and Fabien Mathieu. Distributedly testing cycle-freeness. In *40th Int. Workshop on Graph-Theoretic Concepts in Computer Science (WG)*, volume 8747 of *LNCS*, pages 15–28. Springer, 2014.
- 5 Heger Arfaoui, Pierre Fraigniaud, and Andrzej Pelc. Local decision and verification with bounded-size outputs. In *15th Symp. on Stabilization, Safety, and Security of Distributed Systems (SSS)*, volume 8255 of *LNCS*, pages 133–147. Springer, 2013.
- 6 Baruch Awerbuch, Boaz Patt-Shamir, and George Varghese. Self-stabilization by local checking and correction. In *32nd Symposium on Foundations of Computer Science (FOCS)*, pages 268–277. IEEE, 1991.
- 7 Alkida Balliu, Gianlorenzo D’Angelo, Pierre Fraigniaud, and Dennis Olivetti. What can be verified locally? In *34th Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 66 of *LIPICs*, pages 8:1–8:13, 2017. doi:10.4230/LIPICs.STACS.2017.8.
- 8 Evangelos Bampas and David Ilcinkas. On mobile agent verifiable problems. In *12th Latin American Symposium on Theoretical Informatics (LATIN)*, LNCS 9644, pages 123–137. Springer, 2016. doi:10.1007/978-3-662-49529-2\_10.
- 9 Mor Baruch, Pierre Fraigniaud, and Boaz Patt-Shamir. Randomized proof-labeling schemes. In *24th Symposium on Principles of Distributed Computing (PODC)*, pages 315–324. ACM, 2015. doi:10.1145/2767386.2767421.
- 10 Joffroy Beauquier, Sylvie Delaët, Shlomi Dolev, and Sébastien Tixeuil. Transient fault detectors. *Distributed Computing*, 20(1):39–51, 2007. doi:10.1007/s00446-007-0029-x.
- 11 Lélia Blin and Pierre Fraigniaud. Space-optimal time-efficient silent self-stabilizing constructions of constrained spanning trees. In *35th Int. Conference on Distributed Computing Systems (ICDCS)*, pages 589–598. IEEE, 2015. doi:10.1109/ICDCS.2015.66.
- 12 Lélia Blin, Pierre Fraigniaud, and Boaz Patt-Shamir. On proof-labeling schemes versus silent self-stabilizing algorithms. In *16th Int. Symp. on Stabilization, Safety, and Security of Distributed Systems (SSS)*, LNCS, pages 18–32. Springer, 2014.
- 13 Keren Censor-Hillel, Ami Paz, and Mor Perry. Approximate proof labeling schemes. In *24th Int. Colloquium on Structural Information and Communication Complexity (SIROCCO)*, 2017.
- 14 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. *SIAM J. Comput.*, 41(5):1235–1265, 2012.
- 15 Laurent Feuilloley and Pierre Fraigniaud. Survey of distributed decision. *Bulletin of the EATCS*, 119, 2016. URL: <http://bulletin.eatcs.org/index.php/beatcs/article/view/411/391>.
- 16 Laurent Feuilloley and Pierre Fraigniaud. Error-sensitive proof-labeling schemes. In *31st International Symposium on Distributed Computing (DISC)*, pages 16:1–16:15, 2017. doi:10.4230/LIPICs.DISC.2017.16.
- 17 Laurent Feuilloley, Pierre Fraigniaud, and Juho Hirvonen. A Hierarchy of Local Decision. In *43rd Int. Colloquium on Automata, Languages, and Programming (ICALP)*, LIPICs, pages 118:1–118:15, 2016. doi:10.4230/LIPICs.ICALP.2016.118.
- 18 Klaus-Tycho Foerster, Thomas Luedi, Jochen Seidel, and Roger Wattenhofer. Local checkability, no strings attached: (a)cyclicity, reachability, loop free updates in sdns. *Theor. Comput. Sci.*, 709:48–63, 2018. doi:10.1016/j.tcs.2016.11.018.
- 19 Pierre Fraigniaud, Amos Korman, and David Peleg. Towards a complexity theory for local distributed computing. *J. ACM*, 60(5):35, 2013.
- 20 Pierre Fraigniaud and Andrzej Pelc. Decidability classes for mobile agents computing. *J. Parallel Distrib. Comput.*, 109:117–128, 2017. doi:10.1016/j.jpdc.2017.04.003.
- 21 Pierre Fraigniaud, Sergio Rajsbaum, and Corentin Travers. Minimizing the number of opinions for fault-tolerant distributed decision using well-quasi orderings. In *12th Latin*

- American Symposium on Theoretical Informatics (LATIN)*, pages 497–508. Springer, 2016. doi:10.1007/978-3-662-49529-2\_37.
- 22 Pierre Fraigniaud, Sergio Rajsbaum, Coentin Travers, Petr Kuznetsov, and Thibault Rieutord. Perfect failure detection with very few bits. In *18th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, LNCS 10083, pages 154–169. Springer, 2016. doi:10.1007/978-3-319-49259-9\_13.
  - 23 Silvio Frischknecht, Stephan Holzer, and Roger Wattenhofer. Networks cannot compute their diameter in sublinear time. In *23rd Symposium on Discrete Algorithms (SODA)*, pages 1150–1162. ACM-SIAM, 2012.
  - 24 R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(1):66–77, 1983. doi:10.1145/357195.357200.
  - 25 Mika Göös and Jukka Suomela. Locally checkable proofs in distributed computing. *Theory of Computing*, 12(1):1–33, 2016. doi:10.4086/toc.2016.v012a019.
  - 26 Gene Itkis and Leonid A. Levin. Fast and lean self-stabilizing asynchronous protocols. In *35th Symposium on Foundations of Computer Science (FOCS)*, pages 226–239. IEEE, 1994. doi:10.1109/SFCS.1994.365691.
  - 27 Gillat Kol, Rotem Oshman, and Raghuvansh R. Saxena. Interactive distributed proofs. In *37th ACM Symposium on Principles of Distributed Computing (PODC 2018)*, to appear.
  - 28 Janne H. Korhonen and Jukka Suomela. Brief announcement: Towards a complexity theory for the congested clique. In *31st International Symposium on Distributed Computing (DISC)*, pages 55:1–55:3, 2017. doi:10.4230/LIPIcs.DISC.2017.55.
  - 29 Amos Korman and Shay Kutten. Distributed verification of minimum spanning trees. *Distributed Computing*, 20:253–266, 2007.
  - 30 Amos Korman, Shay Kutten, and Toshimitsu Masuzawa. Fast and compact self-stabilizing verification, computation, and fault detection of an MST. *Distributed Computing*, 28(4):253–295, 2015. doi:10.1007/s00446-015-0242-y.
  - 31 Amos Korman, Shay Kutten, and David Peleg. Proof labeling schemes. *Distributed Computing*, 22(4):215–233, 2010. doi:10.1007/s00446-010-0095-3.
  - 32 Eyal Kushilevitz and Noam Nisan. *Communication complexity*. Cambridge University Press, New York, 1997.
  - 33 Rafail Ostrovsky, Mor Perry, and Will Rosenbaum. Space-time tradeoffs for distributed verification. In *24th International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, pages 53–70, 2017. doi:10.1007/978-3-319-72050-0\_4.
  - 34 Boaz Patt-Shamir and Mor Perry. Proof-labeling schemes: Broadcast, unicast and in between. In *19th Int. Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, LNCS 10616, pages 1–17. Springer, 2017. doi:10.1007/978-3-319-69084-1\_1.
  - 35 David Peleg. *Distributed Computing: A Locality-Sensitive Approach*. Discrete Mathematics and Applications. SIAM, Philadelphia, 2000.
  - 36 David Peleg and Vitaly Rubinfeld. A near-tight lower bound on the time complexity of distributed MST construction. In *40th Symp. on Foundations of Computer Science (FOCS)*, pages 253–261. IEEE, 1999. doi:10.1109/SFFCS.1999.814597.

# Local Verification of Global Proofs

Laurent Feuilloley<sup>1</sup>

University Paris Diderot, France  
feuilloley@irif.fr

 <https://orcid.org/0000-0002-3994-0898>

Juho Hirvonen<sup>2</sup>

University of Freiburg, Germany  
juho.hirvonen@cs.uni-freiburg.de

---

## Abstract

In this work we study the cost of local and global proofs on distributed verification. In this setting the nodes of a distributed system are provided with a nondeterministic proof for the correctness of the state of the system, and the nodes need to verify this proof by looking at only their local neighborhood in the system.

Previous works have studied the model where each node is given its own, possibly unique, part of the proof as input. The cost of a proof is the maximum size of an individual label. We compare this model to a model where each node has access to the same global proof, and the cost is the size of this global proof.

It is easy to see that a global proof can always include all of the local proofs, and every local proof can be a copy of the global proof. We show that there exists properties that exhibit these relative proof sizes, and also properties that are somewhere in between. In addition, we introduce a new lower bound technique and use it to prove a tight lower bound on the complexity of reversing distributed decision and establish a link between communication complexity and distributed proof complexity.

**2012 ACM Subject Classification** Theory of computation → Distributed computing models, Theory of computation → Proof complexity

**Keywords and phrases** Proof-labeling schemes, distributed verification, non-determinism, local proofs

**Digital Object Identifier** 10.4230/LIPIcs.DISC.2018.25

**Acknowledgements** The authors would like to thank Jukka Suomela for mentioning that AMOS could be an interesting problem in this context, and the anonymous reviewers for their feedback.

## 1 Introduction

In distributed decision a distributed system must decide if its own state satisfies a given property. When compared to classical decision problems, the crucial difference is that each node of the distributed system must make its own local decision based only on information available in its local neighborhood. We say that the system *accepts* if all nodes accept, and otherwise the system *rejects*.

A distributed system is modeled as a communication graph where edges denote nodes that can directly communicate with each other. The setting where each node only gets to see its constant-radius neighborhood in the graph is called *local decision* [8]. It is possible to

---

<sup>1</sup> Additional support from ANR project DESCARTES and INRIA project GANG.

<sup>2</sup> Supported by Ulla Tuominen Foundation.



decide local properties of the system in this manner, for example whether a given coloring is correct. On the other hand, it is impossible to decide global properties like whether a given set of edges forms a spanning tree.

To decide global properties, nodes can be provided with a nondeterministic proof, called a *proof-labeling scheme* [14] or a *locally checkable proof* [11]. Each node gets its own proof string as an additional input. Nodes can gather all the information in their constant-radius neighborhood, including these local proof strings, and decide to accept or reject. We say that such a scheme decides a property if there exists an assignment of local proofs to make all nodes accept if and only if the system satisfies the required property.

For example, to prove that a given set of edges forms a spanning tree, each node can be provided with the name of a designated root of the tree, and its distance to the root. Nodes can check that they agree on the identity of the root, and that they have exactly one neighbor with smaller distance to the root along the edges of the spanning tree.

We are interested in minimizing the size of the proof. In particular, we want to minimize the size of the largest label given to a single node. The local proofs often contain redundant information between the different local proof strings. For example, in the previous case each node must know the name of the root. The distances to the root are also highly correlated between neighbors. We approach this question by comparing the size of local proofs to *global* proofs. In this setting each node has access to the same universal proof string. The decision mechanism remains otherwise the same.

It is easy to see that minimum sizes of global and local proofs bound each other. A global proof can simply be a list of the local proof strings. Conversely, a local proof can copy the same global proof for each node. In this work, we study how these two proof sizes relate to each other for different properties.

Unlike in the centralized setting, distributed decision cannot be reversed trivially. This is due to the fact that the distributed decision mechanism is asymmetric: all nodes must accept a correct input, but a failure might only be detectable locally. Decision can be reversed using a logarithmic number of additional nondeterminism [11, 7]: when deciding a language  $\bar{L}$ , a spanning tree rooted at a rejecting node for  $L$  is constructed to convince the remaining nodes that such a node exists. This is an even more general primitive in distributed proofs: the proof must convince the nodes that a local defect exists somewhere in the graph, and only the nodes that are located close to this defect can verify its existence. We show that the existing upper bounds are asymptotically tight: reversing decision requires a local proof of logarithmic size, and global proofs do not help.

Since the distributed verification of a proof happens locally, a distributed proof of a global property must carry information between distant parts of the input graph. This has led to the use of lower bound techniques from communication complexity for distributed decision. On the other hand proving lower bounds inside the nondeterministic hierarchy of local decision [7] with multiple levels of nondeterminism seems to be hard. This is partially due to the fact that current lower bound techniques from communication complexity cease to work. We formalize this intuition by establishing a connection between the nondeterministic local decision hierarchy and the nondeterministic communication complexity hierarchy [1]. This connection exists for local proofs but is even stronger when considering global proofs.

**Motivation.** The first proof-labeling schemes were designed in the context of self-stabilizing algorithms, where a distributed algorithm would, in addition to the output, keep some information to verify that the state of the network is not corrupted. Similar scenarios exist for global proofs. For example, one may consider a network where the machines compute in

a distributed fashion, but an external operator with a view of the whole network can once in a while broadcast a piece of information, such as the name of a leader. As one expects this type of update to be costly, the focus is on minimizing the size of such broadcast information.

Our research belongs to a recent line of work that establishes the foundations of a theory of complexity for distributed network computation. In this context, the certificate comes from a prover, and one studies the impact of non-determinism on computation and the minimal amount of information needed from the prover to decide a task. Global proofs are a natural alternative form of non-determinism. Moreover, in proof-labeling schemes a part of the certificate is often global. For example, the name of a leader is given to all nodes. Global proofs can be used to study how much of such redundant information a local proof must have. Finally, one may consider that global proofs are the most natural equivalent of classical non-determinism: only the algorithm is distributed and we ask what is the cost if distributing the proof.

**Related work.** Proof-labeling schemes have been defined in [14, 15]. An important result in the area is the tight bound on the size of the proofs certifying minimum spanning tree [13]. Recently, several variations have been defined, for verifying approximation [3], with non-constant verification radius [20], with a dependency between the number of errors and the distance to the language [6], and variations on the communication model [22]. An analogue of the polynomial hierarchy for distributed decision has been defined [7].

Another line of work uses a slightly different notion of non-determinism. Fraigniaud et al. [8] consider a similar kind of scheme with a prover and local verifier, but with the constraint that the certificates should not depend on the identifiers of the network. For these works, and more generally the complexity theory of distributed decision, we refer to a recent survey [5].

The idea of a prover for computation in a network, or in a system with several computational units, appears outside of distributed computing, and usually with a global proof. In property testing, models where a prover provides a certificate to the machine that queries the graph have been considered [19, 12]. In two-party communication complexity, non-determinism comes as a global proof that both players can access. Along with non-determinism the authors of [1] define a hierarchy. Separating the levels of this hierarchy is still a major open problem [10].

**Our contributions.** We formalize the notion of *global proofs* for nondeterministic local verification. We study them, in particular comparing the global and local proof complexities of distributed verification.

One main goal of this line of research is to understand the price of locality in nondeterministic distributed verification – that is, how much information must be repeated in the local proofs of the nodes in order to allow local decision of global problems.

1. We show that the price of locality can exhibit the extreme possible values. An example of a maximally *global* property for distributed verification is the language where at most one node is selected. This is one of the core primitives in distributed verification: proving that at most one event of a given type happens in the whole graph. On the other hand, we show that when verifying that at least one node is selected, a global proof must use enough bits to essentially copy every local proof label.
2. We introduce a new proof technique for proving lower bounds for local verification. This proof technique is based on analyzing the neighborhood graph labeled with the local proofs. We use it to show that reversing decision requires  $\Omega(\log n)$ -bit local proof and a  $\Omega(n \log n)$ -bit global proof. Our proof technique is somewhat similar to the one used by

Göös and Suomela to prove their  $\Omega(\log n)$  lower bounds for local proofs [11]. Their proof technique relies on combining several fragments of *yes*-instances to produce an accepting *no*-instance. This is not sufficient for our results, since we want to prove lower bounds for languages for which several fragments of *yes*-instances joined together might still produce a *yes*-instance.

3. We establish a connection between nondeterministic verification and nondeterministic communication complexity. Proving separations for the hierarchy of nondeterministic communication complexity has been an open question since its introduction over 30 years ago [1]. We show that proving similar separations for the hierarchy of nondeterministic local decision is connected to this question: for every boolean function  $f$  we construct a distributed language such that it can be decided on the  $k$ th level if  $f$  can be decided on the  $k$ th level of the communication complexity hierarchy. Considering global proofs instead of local proofs allows to strengthen this result as in this new setting one can also show that verification schemes imply communication protocols. This formalizes the previous intuition that proving lower bounds for nondeterministic local verification is potentially hard as it would imply proving lower bounds for nondeterministic communication complexity.

## 2 Model and definitions

The network is modeled by a simple graph  $G = (V, E)$ . The size of the graph  $|V|$  is denoted by  $n$ . The nodes are given unique identifiers from a range that is polynomial in  $n$  and therefore can be encoded with  $O(\log n)$  bits.

**Distributed decision.** A *distributed language* is a set of labeled graphs  $(G, x)$ , where  $x$  is a function that assigns input labels to nodes and edges of  $G$ . Distributed languages are often assumed to be computable (from the centralized computing perspective), but this is irrelevant for the current paper. An example is the language SPANNING TREE, which is the set of graphs whose edges are labeled with 1 or 0 such that edges labeled with 1 form a spanning tree of the graph.

A *distributed proof*  $\ell: V(G) \rightarrow \{0, 1\}^k$  is a function that assigns a string of bits (also called a *certificate*) to each node of the graph. Each node gets its own string as a part of its input. The *size* of a proof is the length of the proof strings  $k$ .

A *local decision algorithm* (also called a *verifier*) with radius  $t$  is a distributed algorithm  $A$ , in the synchronous message passing model, in which every node  $v$  first gathers all the information about its  $t$ -radius neighborhood (the structure of the graph, the identifiers of the nodes, the local inputs), and possibly some proofs given by one or several provers, and outputs a decision, *accept* or *reject*, based on this information. The distance  $t$  is constant independent of  $n$ , the size of the network, and therefore the algorithm can be seen as a function  $A(v, x, \ell)$  on the local graph topology, the inputs labels, and the possible proof. The verifier is assumed to be uniform, that is, it does not know the size of the graph.

A *local decision scheme* is simply a local decision algorithm, and we say that it decides a language  $L$  if, for every labeled graph, all nodes accept if and only if the labeled graph belongs to  $L$ :

$$\forall(G, x) : (G, x) \in L \iff \forall v \in V(G), A(v, x) = \text{accept}.$$

A *nondeterministic decision scheme* consists of a local decision algorithm and a *prover* that assigns a distributed proof to each node. A nondeterministic scheme exists for a language  $L$

if there is an algorithm  $A$  such that for every labelled graph  $(G, x)$  there exists a proof that makes every node accept if and only if  $(G, x)$  belongs to the language:

$$\forall(G, x) : (G, x) \in L \iff \exists \ell \forall v \in V(G), A(v, x, s) = \text{accept}.$$

In particular, if  $(G, x) \notin L$ , then there must not exist a proof that makes all nodes of  $G$  accept.

**Different types of proofs.** We study three different variants of distributed proofs. In a (*purely*) *local proof*, the prover provides every node with its own label. The local proofs have the same size which depends only on the language and on the size of the network  $n$ . For a given language, the minimum, over all proofs, of the maximum proof size at a single node is denoted by  $s_\ell(n)$ . This is the classic framework of proof-labeling schemes. We introduce (*purely*) *global proofs*, where the prover provides a single certificate, and every node can access it. This can also be seen as a local proof that must assign the same string to every node. Its minimum size is denoted by  $s_g(n)$ . Finally, in *mixed proofs*, the prover provides a global proof and local proofs. The size considered, denoted by  $s_m(n)$ , is the sum of the size of the global proof, and the size of the concatenation of the local proofs in an optimal scheme.

### 3 The price of locality

In this section, we study the size of local, mixed and global proofs for different problems, and the price of locality that follows.

#### 3.1 Proof sizes

In this subsection some general inequalities between the sizes of the different proof sizes are proven. We then discuss the definition of the price of locality.

► **Theorem 1.** *For any language, the optimal proof sizes respect the following inequalities.*

$$s_\ell(n) \leq s_m(n) \leq s_g(n) \tag{1}$$

$$s_m(n) \leq n \cdot s_\ell(n) \tag{2}$$

$$s_g(n) \leq n \cdot s_\ell(n) + O(n \log n). \tag{3}$$

**Proof.** The first line of inequalities mainly follows from the definitions. Suppose one is given a mixed certificate for a language, with local certificates of size  $f(n)$  each, and a global certificate of size  $g(n)$ . The size of this mixed certificate is  $s_m(n) = n \cdot f(n) + g(n)$ . Then one can create a local proof of size  $f(n) + g(n)$ , by giving to every node its local part concatenated with the global part. Thus  $s_\ell(n) \leq s_m(n)$ . The inequality  $s_m(n) \leq s_g(n)$  holds because the mixed proof is a generalization of the global proof. Similarly, if there exists local certificates of size  $s_\ell(n)$ , then one can use them in the mixed model. The size measured in the mixed model will then be  $n \cdot s_\ell(n)$ . Finally, given local certificates, one can craft a global certificate. The global certificate consists of a list of pairs, each pair consisting of an ID and the local certificate of the node with this ID. The size is in  $n \cdot s_\ell(n) + O(n \log n)$  because identifiers are on  $O(\log n)$  bits. ◀



### 3.2 Price of locality

We define the *Price of Locality* for distributed proofs, by analogy with the *Price of Anarchy* in algorithmic game theory [16, 21]. Note that this is not the same as the price of locality that appears in the title of [17]. The price of locality (PoL) of a language is defined as the ratio between the size of the concatenation of the purely local certificates, divided by the size of the mixed certificate. That is:

$$PoL(n) = \frac{n \cdot s_\ell(n)}{s_m(n)}.$$

It may come as a surprise that we use mixed proofs instead of global proofs for this definition. There are several reasons for this. First, the inequalities above insure that with this definition the ratio is between 1 and  $n$ , whereas with global proofs the price could be smaller than 1, thus not a price per se. We study this possibility in Section 5. More fundamentally, mixed proofs are a better way to measure how much it costs to fully distribute a proof, as they are a proper generalization of the local proofs, which is not the case of global proofs. Second, our upper bounds use purely global proofs, and our lower bounds (except in Section 5) consider mixed proofs, thus we get the strongest results on both sides.

► **Remark.** Note that we assume that the local proofs given to the nodes are of the same size, and thus the concatenation is exactly  $n$  times larger than the size of one local certificate. The interesting question of whether the average proof size could be asymptotically better, if proofs of different sizes were allowed, is outside of the scope of this paper.

### 3.3 High price of locality

In this section, we prove that it can be very costly to distribute the proof. This is easy and is a warm-up for the rest of the paper. A scheme uses *uniform local proofs*, if the local proofs given to the nodes of the network are all equal. It is simple to change such proof system into a global proof: just take the uniform local proof and make it global. The verifier has the same behaviour and the scheme is correct. This implies the following theorem.

► **Theorem 2.** *For languages where an optimal proof-labeling scheme uses uniform local proofs, the price of locality is  $\Theta(n)$ .*

This theorem applies to the language SYMMETRY, the set of graphs that do not admit a non-trivial automorphism, which has an optimal scheme with  $O(n^2)$  uniform local proofs [11]. We now consider the language AT-MOST-ONE-SELECTED (AMOS), that has been defined and used in [9]. In this problem, the nodes are given binary inputs, and the *yes*-instances are the ones such that at most one node has input 1. We prove that this language meets the hypothesis of the previous theorem.

► **Theorem 3.** *The language AMOS has an optimal proof-labeling scheme with uniform proofs of size  $O(\log n)$ .*

**Proof.** We describe the scheme. The prover's strategy on *yes*-instances is the following. If there is exactly one selected node, the prover provides the ID of the node as uniform certificate, otherwise it provides an empty label. The verification algorithm is, for every node  $v$ : if  $v$  is selected and the certificate is not its ID, then reject, otherwise accept. It is easy to check that this scheme is correct. First, if no node is selected, all nodes accept, for all certificate. Second, if one nodes is selected, then the prover provides its ID as a certificate, and thus the selected node accepts, and all the other nodes too. Finally, if two or more nodes are selected, at most one of them has its ID written in the global certificate, because the IDs are distinct and thus at least one node is rejecting. ◀



In [11], the authors prove that the language LEADER ELECTION, where exactly one node is labelled 1 and the remaining nodes are labelled 0, requires  $\Omega(\log n)$  local certificate. The proof basically shows that without this amount of proof, an instance with two leaders would be accepted. This reasoning holds for AMOS, and we can derive a  $\Omega(\log n)$  lower bound for local certificates as well.

► **Corollary 4.** *The price of locality for AMOS is in  $\Theta(n)$ .*

### 3.4 Intermediate price of locality

In this subsection, we show that the language MINIMUM SPANNING TREE (MST) has price of locality  $\Theta(\log n)$ . It is an intermediate price, between  $n$  (the previous subsection), and constant (the next section). The language MST is the set of weighted graphs in which the subset of the edges labelled with 1 form a minimum spanning tree of the graph. The edge weights are assumed to be polynomial in  $n$  and for simplicity we assume that the edge weights are distinct.

In [13], the authors show that there exist local proofs of size  $O(\log^2 n)$  for MST, and that this bound is tight. We show a simple global proof that has size  $O(n \log n)$ . As a mixed proof for the simpler language SPANNING TREE requires  $\Omega(n \log n)$  (see Section 4), this bound is also tight.

► **Theorem 5.** *The global proof size for MINIMUM SPANNING TREE is in  $O(n \log n)$ .*

**Proof.** We describe the scheme. On a *yes*-instance the prover provides a list of the selected edges with their weights. This global certificate has size  $O(n \log n)$ , because the edge weights and the identifiers can be written on  $O(\log n)$  bits. Then every node first checks that the certificate is correct regardless of the graph. That is, every node checks that:

- The certificate is a well-formed edge list. Let  $L$  be this list.
- The list  $L$  describes an acyclic graph. That is that there is no set of nodes  $w_1, w_2, \dots, w_k$  such that  $(w_1, w_2), (w_2, w_3), \dots, (w_{k-1}, w_k)$ , and  $(w_k, w_1)$  appear in the list.
- The list  $L$  describes a connected graph. That is for any pair of nodes present in the list, there exists a path in the list that connects them.

Then every node  $v$  of the graph checks locally that:

- The  $L$  is consistent with the selected edges that are adjacent to it.
- It has an adjacent selected edge.
- For every  $e = (v, w)$  in the graph but not in the list, and every edge  $e'$  on the path from  $v$  to  $w$  in  $L$ , the weight of  $e'$  is smaller than the weight of  $e$ .

We now prove the correctness of the scheme. The first part of the verification insures that the set of edges described by  $L$  form an acyclic connected graph. The two first checks of the second part insure that it contains the selected edges and that it is spanning the graph. As it is a spanning tree, it must then be exactly the set of selected edges. Finally, remember that the so-called *cycle property* states that a spanning tree verifying the last item of the previous algorithm is a minimum spanning tree [4]. ◀

## 4 Locality for free and reversing decision

In this section, we show that for some languages there exists local proofs of size  $O(\log n)$  and that any mixed proof has size  $O(n \log n)$ . It follows that in this case, the price of locality is constant, that is the locality of the proofs comes for free.

The language we consider, called AT LEAST ONE SELECTED (ALOS), consists of all labeled graphs such that at least one node has a non-zero input label. We say that a node with a non-zero input label is *selected*. Proving that at least one node has some special property (being the root, having some input, being part of some special subgraph) is an important subroutine in many schemes.

On a more fundamental perspective, reversing decision basically deals with proving that some node is rejecting, which falls into the scope of the ALOS. It has long been known that  $O(\log n)$  local proof is sufficient for reversing decision, and the current section shows that not only is this optimal, but also one cannot gain by using global proofs.

► **Theorem 6.** *A mixed proof for the language ALOS requires  $\Omega(n \log n)$  bits.*

The theorem is equivalent to stating that the language requires either  $\Omega(\log n)$  bits per local proof or an  $\Omega(n \log n)$  bit global proof.

**Proof of Theorem 6.** The proof is essentially a counting argument that shows that for any proof scheme that uses small certificates we can find a graph in which no nodes are selected, but there is a proof that makes the verifier accept the input. This is done by analyzing the structure of the graph where nodes are all possible accepting labelled cycle fragments, and two nodes are adjacent if the verifier accepts locally when they are placed one after the other on a cycle. Finally we show that this graph contains an accepting cycle that has no selected nodes.

Consider a mixed scheme with local certificates of size  $f(n)$  and a global certificate of size  $g(n)$ . Let  $r$  be the verification radius of the scheme.

**Blocks.** The lower bound instances are consistently oriented cycles of length at most  $n = (b + 1)(2r + 1)$ , for some integer  $b$ . Cycles are constructed from blocks of  $2r + 1$  nodes: the  $i$ th block is a path  $B_i = (v_j, v_{j+1}, \dots, v_{j+2r})$ , where  $j = i(2r + 1) + 1$ , oriented consistently from  $v_j$  to  $v_{j+2r}$ . Each node  $v_j$  is labeled with the unique identifier  $j$ .

**Constructing instances from blocks.** Let  $\pi: [b] \rightarrow [b]$  be a permutation on the set of the first  $b$  blocks. Each permutation defines a cycle  $C_\pi$  where we take the blocks in the order given by  $\pi$ , and finally take the  $(b + 1)$ th block. Each pair of consecutive blocks in  $\pi$  is connected by an edge, and  $B_{b+1}$  is connected to  $B_{\pi^{-1}(1)}$ .

Finally, the center node  $v_{b(2r+1)+r+1}$  of  $B_{b+1}$  is labeled with a non-zero label, making the instance a *yes*-instance. All other nodes are labeled with the zero-label. Denote this family of permuted *yes*-instances by  $\mathcal{C} = \{C_\pi\}_\pi$ .

**Labeled blocks.** The prover assigns a local proof of  $f(n)$  bits to each node. Thus, there are  $2^{f(n)(2r+1)}$  different possible labeled versions of each block. We call these *labeled blocks*. Denote by  $B_{i,\ell}$  the block  $B_i$  labeled according to  $\ell$ . We call  $B_i$  the *type* of  $B_{i,\ell}$ .

Consider two labeled blocks,  $B_{i,\ell}$  and  $B_{j,\ell'}$ , in this order, linked by an edge. We say that labeled blocks are accepting from  $B_{i,\ell}$  to  $B_{j,\ell'}$  with global certificate  $L$  if, when we run the verifier on the nodes that are at distance at most  $r$  from an endpoint of the connecting edge, all these nodes accept. We denote this by  $B_{i,\ell} \rightarrow_L B_{j,\ell'}$ .

For each choice  $L$  of the global certificate, this edge relation defines a graph  $G_{\mathcal{B},L}$  on the set of labeled blocks. A path in  $G_{\mathcal{B},L}$  corresponds to a labeled path fragment in which all nodes at least  $r$  steps away from the path's endpoints accept. Finally, an accepting cycle is a cycle in  $G_{\mathcal{B},L}$  such that all nodes accept.

**Bounding the overlap of certificates.** For each  $C_\pi \in \mathcal{C}$ , there must exist an accepting assignment of certificates to the nodes. Let  $L$  denote the global part of this accepting certificate. Such a  $C_\pi$  corresponds to a directed cycle in  $G_{\mathcal{B},L}$ . Note that in this cycle the last edge can be omitted as it would always link the last block to the first block. Then  $C_\pi$  corresponds to a directed path  $P(C_\pi, L)$  of length  $b$  in  $G_{\mathcal{B},L}$ . Denote the set of labeled blocks on this path by  $S(C_\pi, L)$ .

Let  $\mathcal{C}_L$  denote the set of instances such that there exists an accepting local certification given the global certificate  $L$ . Every *yes*-instance has an accepting certification, so there must exist  $L^*$  with

$$|\mathcal{C}_{L^*}| \geq |\mathcal{C}|/2^{g(n)}.$$

Now consider any two instances  $C_\pi$  and  $C_{\pi'}$  in  $\mathcal{C}_{L^*}$ . We drop the specification of the global certificate from the notation. We have the following lemma.

► **Lemma 7.** *For all pairs of instances  $C_\pi, C_{\pi'}$  with the same accepting global certificate  $L$ , we have that  $S(C_\pi, L) \neq S(C_{\pi'}, L)$ .*

**Proof.** Assume that  $C_\pi$  and  $C_{\pi'}$  use the same set of blocks, that is  $S(C_\pi, L^*) = S(C_{\pi'}, L^*)$ . Also assume without loss of generality, that  $\pi$  is the identity permutation. Now in  $P(C_{\pi'})$  there must exist a *back edge* with respect to  $\pi$ , that is, an edge between labeled blocks  $B$  and  $B'$ , of types  $B_{\pi'^{-1}(i)}$  and  $B_{\pi'^{-1}(i+1)}$  respectively, such that  $\pi'^{-1}(i) > \pi'^{-1}(i+1)$ . This is because we assumed that the instances consist of the same blocks, but are different. Therefore at some point an edge of  $C_{\pi'}$  must go backwards in the order of  $\pi$ . We also have that  $B, B' \neq B_{b+1}$  as if there is no back edge before reaching  $B_{b+1}$ , we must have  $C_\pi = C_{\pi'}$ .

This implies that there is an accepting cycle formed by taking first the path from  $B$  to  $B'$  along  $P(C_\pi)$  and then an edge from  $B'$  to  $B$ . This cycle does not contain a selected node. It follows that there is a *no*-instance of size at least  $2(2r+1)$  and a certification that causes the verifier to accept the instance, a contradiction. ◀

► **Remark.** Note that the contradicting instances can be of size  $2(2r+1)$  but the identifiers can be of size  $n$  and the certificates of size  $f(n)$ . Therefore the lower bound only holds for uniform verifiers that do not get any guarantees except that 1) the identifiers come from the set  $[n+c]$ , for some constant  $c$ , and 2) the certificates are of size at least  $f(n)$ .

Alternatively it is possible to consider ALOS on possibly disconnected instances so that every connected component must have at least one node selected. In this case the proof will fool even a non-uniform scheme (that is, one that has information about the real size of the instance).

**Counting argument.** By Lemma 7, each pair of permutations  $\pi, \pi'$  in  $\mathcal{C}_{L^*}$  must induce a different set of labeled blocks that form the accepting certifications of instances  $C_\pi$  and  $C_{\pi'}$ . The number of different permutations in  $\mathcal{C}_{L^*}$  is at least  $b!/2^{g(n)}$ . On the other hand, the number of different sets of labeled blocks, selecting a block of each type, is  $2^{f(n)(2r+1)b}$ . As shown in Lemma 7, to have a legal certification, we must have that  $2^{f(n)(2r+1)b+g(n)} \geq b!$ .

Using Stirling's approximation we get that  $f(n)(2r+1)b + g(n) \geq b \log_2 b - (\log_2 e)b + O(\ln b)$ . Since  $b = \Theta(n)$  and  $r = O(1)$ , this implies that either  $f(n) = \Omega(\log n)$  or  $g(n) = \Omega(n \log n)$ . Thus the mixed proof has size  $\Omega(n \log n)$ . ◀

Theorem 6 implies that reversing decision requires  $\Omega(\log n)$  bit certificates in the following sense.

► **Corollary 8.** *There exists a language, NONE SELECTED, that can be decided locally without nondeterministic proofs and its complement is ALOS, which requires local certificates of size  $\Omega(\log n)$  or global certificates of size  $\Omega(n \log n)$ .*

**Proof.** Consider the language NONE SELECTED, that is, the language of labeled graphs such that all nodes have the zero label. This language is locally decidable without nondeterminism, that is, NONE SELECTED  $\in$  LD [8] or  $\Lambda_0$  in the notation of Section 6. The language ALOS is its complement. Finally, by Theorem 6, deciding ALOS, that is, reversing the decision of NONE SELECTED requires local certificates with  $\Omega(\log n)$  bits or global certificates with  $\Omega(n \log n)$  bits. ◀

The proof can be adapted to several other problems, namely leader election, spanning tree and the set of odd cycles, giving a lower bound for mixed proof systems.

► **Corollary 9.** *Any mixed proof system for LEADER ELECTION requires local certificates of size  $\Omega(\log n)$  or global certificates of size  $\Omega(n \log n)$ .*

**Proof of Corollary 9.** Consider the proof of Theorem 6. The family  $\mathcal{C}$  of *yes*-instances for ALOS is also a family of *yes*-instances for LEADER ELECTION. Since LEADER ELECTION  $\subsetneq$  ALOS, the proof of Theorem 6 produces *no*-instances of LEADER ELECTION that the verifier accepts. ◀

► **Corollary 10.** *Any mixed proof system for SPANNING TREE requires local certificates of size  $\Omega(\log n)$  or global certificates of size  $\Omega(n \log n)$ .*

**Proof sketch.** Consider two types of instances: the cycles where all the edges are selected, and the the cycles where all edges but one are selected. The first instances are not in the language, the second are. We can rephrase this restricted problem as: there is at least one non-selected edge. Then the same type of proof works. ◀

► **Corollary 11.** *Any mixed proof system for ODD-CYCLE requires local certificates of size  $\Omega(\log n)$  or global certificates of size  $\Omega(n \log n)$ .*

**Proof sketch.** The proof of Corollary 11 consists in a refinement of the proof for ALOS. We can build on an odd number of blocks, each block being of odd length itself. Then we can give a colour to each block so that half of the blocks are black and half are white. Finally we can force the paths to alternate between white and black blocks. The cycles obtained will then be of even length, and thus be *no*-instances. The number of possible paths is reduced, but only by term of the form  $2^b$ , which is negligible compared with the  $b!$  term. The calculation then still gives the  $\Omega(n \log n)$  lower bound. ◀

A consequence of these corollaries is that all the  $\Omega(\log n)$  lower bounds obtained in [11] for local certificates can be lifted to  $\Omega(n \log n)$  mixed proofs with our technique. However for the problem AMOS we studied in the previous section, our technique does not work, which is consistent with the fact that an  $\Omega(n \log n)$  lower bound would contradict the  $O(\log n)$  upper bound we show. As already said, the technique of [11] works for AMOS, and provides the  $\Omega(\log n)$  bound for local proofs. The reason our technique fails is because we show that if the certificates are too short then one can shorten the cycles that are *yes*-instances, which is not useful for AMOS, as a ‘subinstance’ of this problem is still in the language: one can only remove selected nodes. The authors of [11] show that one can glue different *yes*-instances together and get a configuration that is still accepted by the nodes, and for AMOS this means one can glue different instances with one node selected, and then get an instance with more

than one node selected, and this instance is still accepted, which raises a contradiction. Note that because of this duality, the proof technique of [11] does not give a lower bound for ALOS, even for the case of local proofs.

It is also worth noting that the intersection of the languages AMOS and ALOS is LEADER ELECTION. It is known that LEADER ELECTION has a proof-labeling scheme of size  $\Theta(\log n)$ , constructed with a spanning tree, along with the ID of the leader given to all the nodes. The results of the current and previous sections show that this decomposition is mandatory: one needs a global part of size  $\Theta(\log n)$ , and a local part of size  $\Theta(\log n)$ .

## 5 Beyond free locality

The language BIPARTITE is the set of bipartite graphs. Local proofs of constant size exist for this language: the prover can just describe a 2-coloring of the graph by giving a bit to each node, and every node can check that its neighbors are given a color different from its own. We conjecture that for this language, even when restricting the topology to cycles, optimal purely global proofs are larger than the sum of the optimal local proof sizes. More precisely this sum is  $\Theta(n)$ , and we conjecture that purely global proofs take  $\Theta(n \log n)$  bits.

► **Conjecture 12.** For BIPARTITE, purely global proofs have size  $\Theta(n \log n)$ .

We are not able to prove the lower bound of the conjecture, but we can prove weaker inequalities. For this problem, the range of the identifiers is important, and that is why we consider the maximum identifier to be a parameter  $M$ , that we do not bound by a polynomial any more.

► **Theorem 13.** For BIPARTITE, there exist two constants  $\alpha$  and  $\beta$  such that, for identifiers bounded by  $M$ :

$$\alpha \max\{n, \log \log M\} \leq s_g(n) \leq \beta \min\{M, n \log M\}.$$

Note that if  $M = n$  then we get a tight  $\Theta(n)$  bound. The  $\Omega(n)$  lower bound holds for any ID range, but the  $\log \log M$  bound shows that this cannot be tight for every ID range: we can get an arbitrarily large lower bound if we allow arbitrarily large identifiers.

**Proof.** We start with the upper bounds. The  $O(n \log M)$  upper bound comes from the certificate made by concatenating the couples (ID, local proof) for every node, as in Theorem 1. For the  $O(M)$  upper bound, the prover strategy is to provide a vector with  $M$  cells, where cell  $i$  will contain a bit indicating the color of the node with ID  $i$ . In both cases the nodes will get their own colors and the colors of their neighbors from the certificate, and they can check locally the consistency of the coloring.

We now prove the lower bounds for the restricted case of cycles. Note that bipartiteness on cycles boils down to distinguishing between odd and even length cycles. A priori, in a scheme for this language, the prover is not forced to explicitly provide a coloring to the nodes. We show that a proof always implies a coloring. More precisely, a node can always extract from the proof its color and the colors of its neighbors, and then check the consistency of the coloring. As in Section 4, we will use blocks of nodes to build a large a number of instances. The blocks are paths of  $2r + 1$  nodes. The  $i$ -th block, noted  $b_i$  has consecutive IDs from  $i(2r + 1) + 1$  up to  $(i + 1)(2r + 1)$ . Every block is oriented in the direction of increasing IDs. A *block-based cycle* is a cycle made by concatenating blocks, with a consistent orientation.

► **Lemma 14.** *For every global proof  $c$ , there exists a coloring function  $f_c : [M] \mapsto \{0, 1\}$ , such that for every block-based cycle  $H$  that is accepting with certificate  $c$ ,  $f_c$  defines a proper coloring of  $H$ .*

**Proof of Lemma 14.** First, note that as the blocks have odd length, a block-based cycle has even length if and only if it is composed of an even number of blocks. Then, for block-based cycles, replacing virtually each block by a vertex, and trying to 2-color the resulting cycle is equivalent to 2-color the nodes of the original instance.

Fix a certificate  $c$ . Consider the directed graph  $G_c$ , whose nodes are the blocks  $(b_i)_i$ . There is an oriented edge  $(b_i, b_j)$  if and only if there exists a block-based cycle for which  $c$  is an accepting certificate, and where the block  $b_i$  is followed by the block  $b_j$ .

► **Claim 15.** *The graph  $G_c$  contains no directed odd cycle.*

**Proof of Claim 15.** Suppose the graph  $G_c$  contains a directed odd cycle. Consider the corresponding block-based cycle  $C$ . Because it has odd length, it is a *no*-instance. Consider a node  $v$  of this instance that is rejecting with certificate  $c$ . Without loss of generality, assume it is in the first half of its block  $b_i$  (that is, its ID is between  $i(2r+1)+1$  and  $i(2r+1)+r+1$ ). Let  $b_h$  be the block preceding  $b_i$  in  $C$ . The node  $v$  can only see (parts of) of  $b_h$  and  $b_i$ , because its radius is  $r$ . As  $(b_h, b_i)$  belongs to  $G_c$ , there exists a *yes*-instance  $C'$ , in which every node accepts with proof  $c$ , and in which  $b_i$  follows  $b_h$ . This is a contradiction, because with certificate  $c$ ,  $v$  is accepting in  $C'$ , and rejecting in  $C$ , although it has the exact same view in both instances. Thus the graph  $G_c$  contains no directed odd cycle. ◀

► **Claim 16.** *Every connected component of the graph  $G_c$  is strongly connected.*

**Proof of Claim 16.** Consider the following way of building  $G_c$ : take an arbitrary ordering of the cycles that accept with  $c$ , and add them (i.e. add their edges) to  $G_c$ , one by one. We show the strong connectivity of the connected components by induction. The property holds for the empty graph. Suppose every connected component is strongly connected until some step, and that we add a new cycle. As a directed cycle is strongly connected, merging it with one or several strongly connected components, keeps the strong connectivity. ◀

It is known that a strongly connected digraph with no odd length directed cycles can be 2-colored (see e.g. Theorem 1.8.1 in [2]). Thus, from Claim 15 and Claim 16, we get that  $G_c$  has a 2-coloring. This 2-coloring induces a 2-coloring on all the block-based cycles accepting with  $c$ , thus it defines the function  $f_c$  of the lemma. ◀

Now fix a size  $n$ , for an even  $n$ , and consider the following table. The columns are indexed by the blocks, thus there are  $M/(2r+1)$  of them. The rows are indexed by all the possible certificates, that is all the strings on  $s_g(n)$  bits. The cell that corresponds to block  $b$  and certificate  $c$  contains the color given by  $f_c$  to the center node of  $b$ . We will now give two simple properties of this table that will imply the two lower bounds.

Let a *balanced binary vector* be a vector of bits with the same number of zeros and ones. Let the *complement* of a binary vector be the same binary vector where ones and zeros have been complemented.

► **Lemma 17.** *For every balanced binary vector  $p$  of length  $n$ , there exists a row of the table such that the vector made by the  $n$  first cells is equal to either  $p$  or its complement.*

**Proof of Lemma 17.** Consider a balanced binary vector  $p$ . Consider a cycle  $H$  made by concatenating the  $n$  first blocks, in an ordering such that coloring block  $i$  with the  $i^{\text{th}}$  bit of  $p$ , defines a proper coloring of the cycle. Note that, as  $p$  is balanced, such a cycle must exist. This cycle  $H$  has even length, thus it belongs to the language and there exists an accepting certificate  $c$ . The first  $n$  cells of the row of  $c$  must describe a proper coloring of  $H$ , and there are only two such colorings:  $p$  and its complement. ◀

For every balanced vector of length  $n$  there exists a row that it matches (or its complement matches) on the  $n$  first cells. A row can only correspond to one such vector (up to complement), and since there are at least  $2^{n/2}/2$  balanced binary vectors (first  $n/2$  bits can be chosen freely) the table must have at least  $2^{n/2}$  rows. This means that there are at least  $2^{n/2}$  different certificates, thus the certificate size is lower bounded by  $n$ , up to multiplicative constants.

► **Lemma 18.** *Two columns of the table cannot be equal.*

**Proof of Lemma 18.** Suppose columns  $i$  and  $j$  are equal. Consider an even-length block-based cycle  $C$ , where the block  $i$  is linked to the block  $j$ . Such a cycle always exists. For every certificate  $c$ , the same color is given to both blocks  $i$  and  $j$  in  $f_c$ , because the columns are equal. Thus no certificate provides a proper coloring of  $C$ , which is a contradiction because  $C$  belongs to the language. ◀

As there are  $M$  different columns, there is at least order of  $\log(M)$  certificates. Then the length of a certificate is in  $\Omega(\log \log(M))$ . This finishes the proof of Theorem 13. ◀

## 6 Local decision and communication complexity

In this section we present the nondeterministic hierarchies for local decision and communication complexity.

**Nondeterministic hierarchy of local decision.** Feuilloley et al. [7] introduced a nondeterministic *hierarchy of local decision*. It is the distributed computing analogue of the classical polynomial hierarchy. A prover and a disprover take turns, providing each node with proofs of size  $O(\log n)$ . Once the proof labels have been assigned, the nodes look at their constant-radius neighborhood, including the nondeterministic proofs, and decide whether they accept or not.

The classes  $\Sigma_k$  and  $\Pi_k$  correspond to the languages that can be decided using  $k$  levels of nondeterminism – in  $\Sigma_k$  the prover goes first, and in  $\Pi_k$  the disprover. Let  $\ell_1, \ell_2, \dots, \ell_k$  denote the  $k$  levels of nondeterministic labels provided to the nodes. A language  $L \in \Sigma_k$  if and only if there exists a verifier  $A$  such that

$$(G, x) \in L \iff \exists \ell_1, \forall \ell_2, \dots, \forall \ell_k, \forall v \in V(G), A \text{ accepts.}$$

Here  $\exists$  denotes the existential quantifier if  $k$  is odd and the universal quantifier otherwise. The classes  $\Pi_k$  are defined similarly, but with the disprover (i.e. universal quantifier) going first.

The classes that corresponds to the disprover talking last collapse to the previous level, and the only interesting levels are  $\Sigma_1, \Pi_2, \Sigma_3, \dots$ , which are denoted by  $(\Lambda_k)_{k \in \mathbb{N}}$ . The complements of these classes are denoted by  $\text{co-}\Lambda_i$  and we have that  $\text{co-}\Lambda_k \subseteq \Lambda_{k+1}$  [7], i.e., decision can always be reversed using an extra quantifier with  $O(\log n)$  bits. As shown in Theorem 6, in general,  $\Omega(\log n)$  bits are also required for reversing decision.



The main open question of Feuilleley et al. [7] was whether  $\Lambda_2$  and  $\Lambda_3$  were different or not. As in the polynomial hierarchy, the equality  $\Lambda_k = \Lambda_{k+1}$  of two levels would imply a collapse of the local hierarchy down to the  $k$ th level. We show that this question is related to long-standing open questions nondeterministic communication complexity [1].

**A hierarchy for global certificates.** Similar to the hierarchy of local certificates, we can define a hierarchy for the global certificates. Define  $\Sigma_k^G$ ,  $\Pi_k^G$ , and  $\Lambda_k^G$  as previously, except that the labels  $\ell_1, \ell_2, \dots, \ell_k$  are global certificates seen by all nodes.

**Communication complexity.** We will compare the hierarchies of nondeterministic local decision to the hierarchy of nondeterministic communication complexity defined by Babai et al. [1].

In the communication complexity setting we are given a boolean function  $f$  on  $2n$  bits. Two entities, Alice and Bob, are each given  $n$ -bit vectors  $x$  and  $y$ , and have to decide if  $f(x \cup y) = 1$ . They can communicate through a reliable channel and have unlimited computational resources. The measure of complexity is the number of bits Alice and Bob need to communicate in order to decide  $f$ . For more details, see for example the book [18].

In nondeterministic communication complexity Alice and Bob have access to nondeterministic advice (we will say that it is given by a *prover*). The cost of a protocol is the sum of the number of bits communicated by Alice and Bob and the number of advice bits given by the prover. This means that messages of Alice and Bob can equivalently be encoded in the advice.

Babai et al. defined a hierarchy of nondeterministic communication complexity [1]. In addition to Alice and Bob we have two players, whom we will call *prover* and *disprover* for consistency, giving nondeterministic advice to Alice and Bob. Prover and disprover will alternate  $k$  times and each time give an advice string of  $g(n)$  bits. Now we define the class  $\Sigma_k^{cc}(g(n))$  of boolean functions as the set of functions such that there exists an algorithm  $A$  for Alice, and an algorithm  $B$  for Bob such that if  $f \in \Sigma_k^{cc}(g(n))$ , then

$$\forall x, y, \exists \ell_1, \forall \ell_2, \dots, \mathcal{Q} \ell_k A(\ell_1, \ell_2, \dots, \ell_k, x) = B(\ell_1, \ell_2, \dots, \ell_k, y) = 1 \iff f(x, y) = 1.$$

Again  $\mathcal{Q}$  denotes the existential quantifier if  $k$  is odd and the universal quantifier otherwise. The classes  $\Pi_k^{cc}(g(n))$  are defined similarly, but with the disprover going first. We are particularly interested in this hierarchy when  $g(n) = O(\log n)$ . Note that in their work, Babai et al. consider the hierarchy for  $g(n) = O(\text{poly}(\log n))$  [1].

## 6.1 Connecting local decision and communication complexity

In this section we partially formalize the intuition that complexity of local verification is connected to communication complexity. We show that general lower bound proof techniques for nondeterministic local verification will also apply to communication complexity. We then show that if one considers global proofs instead of local ones, the result can be strengthened.

► **Theorem 19.** *For every boolean function  $f$ , there exists a distributed language  $L_f$  such that if  $f \in \Sigma_k^{cc}(g(n))$  for odd  $k$  or  $f \in \Pi_k^{cc}(g(n))$  for even  $k \geq 2$ , then  $L_f \in \Lambda_k(g(n))$ .*

The proof is by showing that there exists a family of languages such that a nondeterministic verification scheme can simulate a nondeterministic communication protocol. The theorem partially explains why it is difficult to separate the different levels of the local decision hierarchy – the question is inherently tied to long-standing open questions in communication complexity [1].



**Proof of Theorem 19.** Let  $f$  be a boolean function on  $2n$  variables. We will construct an infinite family of graphs  $\mathcal{G}_n = (G(n, t, x, y))_{t, x, y}$  and a related language  $L_f$ .

The graph  $G(n, t, x, y)$  consists of a path  $P_{2t+1} = (v_1, v_2, \dots, v_{2t+1})$  of length  $2t + 1$ , and two sets of nodes,  $V_A$  and  $V_B$  of size  $n$ . Let us denote  $v_A = v_1$  and  $v_B = v_{2t+1}$ . We add an edge between each  $v \in V_A$  and  $v_A$ , and an edge between each  $u \in V_B$  and  $v_B$ . The nodes  $v_A$  and  $v_B$  are labelled with their respective identities.

Parameters  $x$  and  $y$  are bit vectors of length  $n$ , corresponding to the inputs of players  $A$  and  $B$  in the communication complexity setting. To encode the input vectors, we use graphs on  $V_A$  and  $V_B$ , respectively. There are  $2^n$  possible input vectors. We'll define a function  $\phi$  that maps each graph on  $n$  nodes to an  $n$ -bit vector. Since the encoding of the input cannot depend on the unique identifiers,  $\phi$  must map all graphs of the same isomorphism class to the same vector. Finally, since there are at least  $2^{\binom{n}{2}}/n! = \Omega(2^{n^2})$  such graph isomorphism classes, we can find a  $\phi$  such that for all  $x \neq y$ , we have that  $\phi^{-1}(x) \cap \phi^{-1}(y) = \emptyset$ .

Given  $\phi$ ,  $x$ , and  $y$ , we can choose two graphs  $G_A \in \phi^{-1}(x)$  and  $G_B \in \phi^{-1}(y)$ , identify the node sets  $V_A$  and  $V_B$  with  $V(G_A)$  and  $V(G_B)$ , respectively, and add the corresponding edges to the graph  $G(n, t, x, y)$ . We will use  $G_A$  and  $G_B$ , respectively, to denote these graphs on node sets  $V_A$  and  $V_B$ . Nodes  $v_A$  and  $v_B$  are labelled as special nodes so that the structure of  $G_A$  and  $G_B$  can be detected. We denote this graph construction by  $G(n, t, x, y)$ .

**Local verification of  $\mathcal{G}_f$ .** A single  $O(\log n)$ -bit certificate is enough to verify the structure of  $G(n, t, x, y)$ . It first consists of a spanning tree of  $P_{2t+1}$ : node  $v_A$  is marked as root, and each node  $v_i$  has a pointer to  $v_{i-1}$  and a counter  $i$ , its distance to the root. It also contains the value  $n$ . The nodes  $v_A$  and  $v_B$  can check that the sizes of the graph  $G_A$  and  $G_B$  are consistent with this value. They also check that there are no other outgoing edges from  $G_A$  and  $G_B$ . Nodes  $v_A$  and  $v_B$  can see all nodes of  $G_A$  and  $G_B$ , and determine their isomorphism classes, and compute  $x = \phi(G_A)$  and  $y = \phi(G_B)$ , respectively.

**Deciding  $L_f$ .** We say that  $G \in L_f$  if and only if

1. the structure of  $G$  is that of  $G(n, t, x, y)$  for some setting of the parameters, and
2. the function  $f$  evaluates to 1 on  $\phi(G_A) \cup \phi(G_B)$ .

Now assume that  $f$  is on the  $k$ th level of the communication complexity hierarchy with  $s = \Omega(\log n)$  bits of nondeterminism. We can use this implied protocol  $P$  to solve  $L_f$  on the  $k$ th level. If the graph structure is correct, the prover and disprover essentially simulate their counterparts from the communication complexity setting, and label *all* nodes on  $P_{2t+1}$  as if in  $P$ . Then  $v_A$  can simulate  $A$  and  $v_B$  can simulate  $B$ , accepting if and only if  $f(x, y) = 1$ . If the prover tries to deviate from this strategy, nodes can see that its labelling of  $P_{2t+1}$  is not constant, and reject. If the disprover tries to deviate, the prover can construct a certificate pointing to this error, and all nodes will accept. ◀

**Global proofs and communication complexity.** In the setting of global proofs we can show a slightly stronger theorem.

► **Theorem 20.** *For every boolean function  $f$  and every  $g(n) = \Omega(\log n)$  there exists a distributed language  $L_f$  such that  $L_f \in \Lambda_k^G(g(n))$ , for  $k \geq 1$  if and only if  $f$  is in the  $k$ th level of the communication complexity hierarchy with  $O(g(n))$  bits of nondeterminism, in particular  $f \in \Sigma_k^{\text{cc}}$  for  $k$  odd or  $f \in \Pi_k^{\text{cc}}$  for  $k$  even.*

In particular, this theorem implies that any collapse in the hierarchy for global certificates implies a collapse in the corresponding communication complexity hierarchy.

**Proof of Theorem 20.** We show that with respect to the language  $L_f$  defined in the proof of Theorem 19, the communication complexity model and the global verification model can simulate each other.

1. *Communication protocol implies a global verification protocol.* The proof proceeds essentially as in the proof of Theorem 19. Using  $O(t \log n)$  bits the global certificate can give the list of nodes on the path between  $v_A$  and  $v_B$ . If a node has degree 2, it must see its own name on this list. Nodes  $v_A$  and  $v_B$  can again locally verify the structure of  $G_A$  and  $G_B$  and recover  $x$  and  $y$ . Finally the prover and disprover follow the communication protocol  $P$  on instance  $(x, y)$ , allowing nodes  $v_A$  and  $v_B$  to simulate Alice and Bob.

2. *Global verification scheme implies a communication protocol.* Assume there is a  $k$ th level global verification scheme with  $g(n)$ -bit certificates for  $L_f$ .

Alice and Bob will simulate this scheme as follows. Construct a virtual graph  $G(x, y)$  consisting of three parts: the nodes  $v_A$  and  $v_B$ , a path  $P_{2t+1}$  of length  $2t + 1$  between them, and graphs  $H(x)$  and  $H(y)$  that are the first elements (in some order) of  $\phi^{-1}(x)$  and  $\phi^{-1}(y)$ , respectively. Finally, all nodes of  $H(x)$  are connected to  $v_A$  and all nodes of  $H(y)$  to  $v_B$ . Only Alice will know  $H(x)$  and only Bob  $H(y)$ .

This graph is in  $L_f$  if and only if  $f(x, y) = 1$ : the structure is exactly as in the definition of  $L_f$ .

Now the nondeterministic prover and disprover can simulate their counterparts in the global verification scheme. Alice and Bob accept if and only if the prover can force all nodes they control to accept. Thus the complexity is bounded by the complexity  $g(n)$  of the global verification scheme. ◀

---

## References

- 1 László Babai, Peter Frankl, and Janos Simon. Complexity classes in communication complexity theory. In *Proc. 27th Annual Symposium on Foundations of Computer Science (FOCS 1986)*, pages 337–347, 1986. doi:10.1109/SFCS.1986.15.
- 2 Jørgen Bang-Jensen and Gregory Gutin. *Digraphs - theory, algorithms and applications*. Springer, 2002.
- 3 Keren Censor-Hillel, Ami Paz, and Mor Perry. Approximate proof-labeling schemes. In *Structural Information and Communication Complexity - 24th International Colloquium, SIROCCO 2017, Porquerolles, France, June 19-22, 2017, Revised Selected Papers*, pages 71–89, 2017. doi:10.1007/978-3-319-72050-0\_5.
- 4 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. URL: <http://mitpress.mit.edu/books/introduction-algorithms>.
- 5 Laurent Feuilloley and Pierre Fraigniaud. Survey of distributed decision. *Bulletin of the EATCS*, 119, 2016. URL: <http://bulletin.eatcs.org/index.php/beatcs/article/view/411/391>.
- 6 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*, pages 16:1–16:15, 2017. doi:10.4230/LIPIcs.DISC.2017.16.
- 7 Laurent Feuilloley, Pierre Fraigniaud, and Juho Hirvonen. A Hierarchy of Local Decision. In *Proc. 43rd International Colloquium on Automata, Languages, and Programming (ICALP 2016)*, volume 55 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 118:1–118:15, 2016. doi:10.4230/LIPIcs.ICALP.2016.118.
- 8 Pierre Fraigniaud, Amos Korman, and David Peleg. Local distributed decision. In *IEEE 52nd Annual Symposium on Foundations of Computer Science, FOCS 2011, Palm Springs, CA, USA, October 22-25, 2011*, pages 708–717, 2011. doi:10.1109/FOCS.2011.17.

- 9 Pierre Fraigniaud, Amos Korman, and David Peleg. Towards a complexity theory for local distributed computing. *J. ACM*, 60(5):35, 2013. doi:10.1145/2499228.
- 10 Mika Göös, Toniann Pitassi, and Thomas Watson. The landscape of communication complexity classes. In *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, July 11-15, 2016, Rome, Italy*, pages 86:1–86:15, 2016. doi:10.4230/LIPIcs.ICALP.2016.86.
- 11 Mika Göös and Jukka Suomela. Locally checkable proofs in distributed computing. *Theory of Computing*, 12(1):1–33, 2016. doi:10.4086/toc.2016.v012a019.
- 12 Tom Gur and Ron D. Rothblum. Non-interactive proofs of proximity. In *Proceedings of the 2015 Conference on Innovations in Theoretical Computer Science, ITCS 2015, Rehovot, Israel, January 11-13, 2015*, pages 133–142, 2015. doi:10.1145/2688073.2688079.
- 13 Amos Korman and Shay Kutten. Distributed verification of minimum spanning trees. *Distributed Computing*, 20(4):253–266, 2007. doi:10.1007/s00446-007-0025-1.
- 14 Amos Korman, Shay Kutten, and David Peleg. Proof labeling schemes. In *Proceedings of the Twenty-Fourth Annual ACM Symposium on Principles of Distributed Computing, PODC 2005, Las Vegas, NV, USA, July 17-20, 2005*, pages 9–18, 2005. doi:10.1145/1073814.1073817.
- 15 Amos Korman, Shay Kutten, and David Peleg. Proof labeling schemes. *Distributed Computing*, 22(4):215–233, 2010. doi:10.1007/s00446-010-0095-3.
- 16 Elias Koutsoupias and Christos H. Papadimitriou. Worst-case equilibria. *Computer Science Review*, 3(2):65–69, 2009. doi:10.1016/j.cosrev.2009.04.003.
- 17 Fabian Kuhn. *The price of locality: exploring the complexity of distributed coordination primitives*. PhD thesis, ETH Zurich, 2005. URL: <http://d-nb.info/977273725>.
- 18 Eyal Kushilevitz and Noam Nisan. *Communication complexity*. Cambridge University Press, 1997.
- 19 László Lovász and Katalin Vesztegombi. Non-deterministic graph property testing. *Combinatorics, Probability & Computing*, 22(5):749–762, 2013. doi:10.1017/S0963548313000205.
- 20 Rafail Ostrovsky, Mor Perry, and Will Rosenbaum. Space-time tradeoffs for distributed verification. In *Structural Information and Communication Complexity - 24th International Colloquium, SIROCCO 2017, Porquerolles, France, June 19-22, 2017, Revised Selected Papers*, pages 53–70, 2017. doi:10.1007/978-3-319-72050-0\_4.
- 21 Christos H. Papadimitriou. Algorithms, games, and the internet. In *Proceedings on 33rd Annual ACM Symposium on Theory of Computing, July 6-8, 2001, Heraklion, Crete, Greece*, pages 749–753, 2001. doi:10.1145/380752.380883.
- 22 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 2017, Boston, MA, USA, November 5-8, 2017, Proceedings*, pages 1–17, 2017. doi:10.1007/978-3-319-69084-1\_1.



# A Simple Parallel and Distributed Sampling Technique: Local Glauber Dynamics

**Manuela Fischer**

ETH Zurich, Switzerland  
manuela.fischer@inf.ethz.ch

**Mohsen Ghaffari**

ETH Zurich, Switzerland  
ghaffari@inf.ethz.ch

---

## Abstract

*Sampling* constitutes an important tool in a variety of areas: from machine learning and combinatorial optimization to computational physics and biology. A central class of sampling algorithms is the *Markov Chain Monte Carlo* method, based on the construction of a Markov chain with the desired sampling distribution as its stationary distribution. Many of the traditional Markov chains, such as the *Glauber dynamics*, do not scale well with increasing dimension. To address this shortcoming, we propose a simple local update rule based on the Glauber dynamics that leads to efficient parallel and distributed algorithms for sampling from Gibbs distributions.

Concretely, we present a Markov chain that mixes in  $O(\log n)$  rounds when Dobrushin's condition for the Gibbs distribution is satisfied. This improves over the *LubyGlauber* algorithm by Feng, Sun, and Yin [PODC'17], which needs  $O(\Delta \log n)$  rounds, and their *LocalMetropolis* algorithm, which converges in  $O(\log n)$  rounds but requires a considerably stronger mixing condition. Here,  $n$  denotes the number of nodes in the graphical model inducing the Gibbs distribution, and  $\Delta$  its maximum degree. In particular, our method can sample a uniform proper coloring with  $\alpha\Delta$  colors in  $O(\log n)$  rounds for any  $\alpha > 2$ , which almost matches the threshold of the sequential Glauber dynamics and improves on the  $\alpha > 2 + \sqrt{2}$  threshold of Feng et al.

**2012 ACM Subject Classification** Theory of computation → Distributed algorithms

**Keywords and phrases** Distributed Graph Algorithms, Parallel Algorithms, Local Algorithms, Locality, Sampling, Glauber Dynamics, Coloring

**Digital Object Identifier** 10.4230/LIPIcs.DISC.2018.26

**Related Version** A full version of the paper is available at <https://arxiv.org/abs/1802.06676>.

## 1 Introduction

*Locally Checkable Labeling* (LCL) [17] problems have been studied extensively for more than three decades [14]. Sampling from the solution space of such LCLs, however, has not attracted a lot of attention and has been investigated only by a recent work [7], despite its numerous motivations, which we will outline in the following.

**Markov Chain Monte Carlo Method.** The *Markov Chain Monte Carlo* (MCMC) method is a central class of algorithms for *sampling*, that is, for randomly drawing an element from a ground set according to a certain probability distribution. It works by constructing a Markov chain with the targeted sampling distribution as its stationary distribution. Within a number of steps, known as the mixing time, the Markov chain converges; its state then



© Manuela Fischer and Mohsen Ghaffari;  
licensed under Creative Commons License CC-BY  
32nd International Symposium on Distributed Computing (DISC 2018).  
Editors: Ulrich Schmid and Josef Widder; Article No. 26; pp. 26:1–26:11  
Leibniz International Proceedings in Informatics



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

(approximately) follows this distribution. Besides the intrinsic interest of such a general sampling method, in particular for complex distributions where simple sampling techniques fail, the MCMC method gives rise to efficient approximation algorithms in a variety of areas: enumerative combinatorics (due to the fundamental connection between sampling and counting established by Jerrum, Valiant, and Vazirani [13]), simulated annealing [16] in combinatorial optimization, Monte Carlo simulations [15] in statistical physics, computation of intractable integrals for, among many others, Bayesian inference [1] in machine learning.

**Parallel and Distributed Sampling.** The employment of MCMC methods is particularly important when confronted with high-dimensional data, where traditional (exact) approaches quickly become intractable. Such data sets are not only increasingly frequent, but also critical for the success of many applications. For instance in machine learning, higher-dimensional models help expressibility and hence predictability. It is thus central that MCMC algorithms scale well with increasing dimensions. This is not the case, however, for most sequential methods, as they process and update the variables one by one, that is, a single site per step. To speed up the sampling process, Markov chain updates can be parallelized by spreading the variables across several processors. In other settings, such as distributed machine learning, the (data associated to) variables might already be naturally distributed among several machines, and the overhead of aggregating them into one machine, if they fit there in the first place, would be untenable.

**Local Sampling.** In either case, to avoid overhead in communication and coordination, local update rules for Markov chains are needed: a machine must be able to change the value of its variables without knowing all the values of the variables on other machines. Yet, the joint distribution, over all variables in the system, must converge to a certain distribution. This local sampling problem was introduced in a recent work by Feng, Sun, and Yin [7], whose title asks “*What can be sampled locally?*”. We address this question by providing a simple and generic sampling technique – the *Local Glauber Dynamics*, informally introduced in Section 1.2 and formally described in Section 2 – which is applicable for a wide range of distributions, as stated in Section 1.1. This moves us a step closer towards an answer of this question, thus towards the goal of generally understanding what can be sampled locally. Besides its many practical ramifications, especially on the area of distributed machine learning, this gives us a theoretical insight about the *locality* of problems, whose systematic study has been initiated by the seminal works of Linial [14] and Naor and Stockmeyer [17] with the pithy title “*What can be computed locally?*”.

## 1.1 Our Result, and Related Work.

For the sake of succinctness and comprehensibility of the presentation, we state and prove our main result in terms of the special case that gets most attention for sequential sampling: sampling proper colorings of a graph. We refer to [8] for a survey on sequential sampling of proper colorings. Our result applies to a more general set of distributions, however, as explained in the remark at the end of this section. Note that independently and simultaneously, Feng, Hayes, and Yin [6] arrived at the same result.

► **Theorem 1.** *A uniform proper  $q$ -coloring of an  $n$ -node graph with maximum degree  $\Delta$  can be sampled within total variation distance  $\varepsilon > 0$  in  $O(\log(\frac{n}{\varepsilon}))$  rounds, where  $q = \alpha\Delta$  for any  $\alpha > 2$ .*

Our parallel and distributed sampling algorithm improves over the *LubyGlauber* algorithm by Feng, Sun, and Yin [7], which needs  $O(\Delta \log(\frac{n}{\epsilon}))$  rounds, and their *LocalMetropolis* algorithm, which converges in  $O(\log(\frac{n}{\epsilon}))$  rounds but requires a considerably stronger mixing condition of  $\alpha > 2 + \sqrt{2}$ . They state that “*We also believe that the  $2 + \sqrt{2}$  threshold is of certain significance to this [LocalMetropolis] chain as the Dobrushin’s condition to the Glauber dynamics.*”, thus implying that this value is a barrier for their approach. This is also justified by the supposedly easiest special case of a tree that leads to the same threshold for their algorithm. Our result gets rid of the additional  $\sqrt{2}$  while not incurring any loss in the round complexity, with a considerably easier and more natural update rule. Not only is our proof simpler and shorter, our algorithm is also asymptotically best possible, as there is an  $\Omega(\log(\frac{n}{\epsilon}))$  lower bound [10, 7] due to the exponential correlation between variables.

The threshold of  $\alpha > 2$  corresponds to Dobrushin’s condition, thus almost matches the threshold of the sequential Glauber dynamics [12, 19] at  $2\Delta + 1$ . In other words, we present a technique that fully parallelizes the Glauber dynamics, speeding up the mixing time from poly  $n$  steps to  $O(\log n)$  rounds<sup>1</sup>. In terms of number of colors needed, Dobrushin’s condition can be undercut: Vigoda [20] and two very recent works [3, 4] showed that, when resorting to a different highly non-local Markov chain,  $\alpha = \frac{11}{6}$  is enough. This gives rise to the question whether efficient distributed algorithms intrinsically need to be stuck at Dobrushin’s condition, which would imply that this bound is inherent to the locality of the sampling process, or whether our threshold is an artifact of our possibly suboptimal dynamics.

► **Remark.** In fact, our technique directly applies for sampling from the Gibbs distribution induced by a Markov random field<sup>2</sup> if Dobrushin’s condition [5] is satisfied. More generally, it can be used for sampling from any local (that is, constant-radius) constraint satisfaction problems, which is universal for conditional independent joint distributions, due to Hammersley-Clifford’s fundamental theorem [11]. Moreover, our proof presented here captures all the difficulties that arise in these more general cases, thus can be adapted in a straight-forward manner. We defer this generalization to the full version of the paper.

## 1.2 Our Sampling Technique, and Related Approaches

Over the past few years, several methods to parallelize sequential Markov chains have been proposed. Most of them rely on heavy coordination machinery, are special purpose, and/or do not provide any theoretical guarantees. In the following, we briefly outline two of the most promising and more generic parallel and distributed sampling techniques, in the context of colorings.

The most natural one follows a standard decentralization approach, also implemented in the *LubyGlauber* algorithm by [7]: an independent set of nodes (e.g., a color class of a proper coloring) simultaneously updates their color [7], ensuring that no two neighboring nodes change their color at the same time. This approach mainly suffers from the limitation that the number of independent sets needed to cover all nodes might be large, which slows down mixing. In particular, a multiplicative  $\Delta$ -term in the mixing time seems inevitable [9, 7]. In the worst case of a clique, this approach falls back to sequential sampling, updating one node after the other. Moreover, this method requires an independent set to be computed, which incurs a significant amount of additional communication and coordination.

<sup>1</sup> Note that our parallel and distributed algorithm directly gives rise to a centralized algorithm with running time  $O(n \log n)$ . The number of colors, however, is slightly larger than what state-of-the-art centralized algorithms require.

<sup>2</sup> This captures many graph problems – such as independent set, vertex cover, graph homomorphism – and physical models – such as Ising model, Potts model, general spin systems, and hardcore gas model.



An orthogonal direction was pursued by [18, 21, 7], where methods are introduced to update the colors of all nodes simultaneously. One example is the *LocalMetropolis* algorithm by [7]. This extreme parallelism, however, comes at a cost of either introducing a bias in the stationary distribution, resulting in a non-uniform coloring [18, 21], or having to demand stronger mixing conditions [7].

**Our Local Sampling Technique.** We aim for the middle ground between these two approaches, motivated by the following observation: we do not need to prevent simultaneous updates of adjacent nodes, only simultaneous *conflicting* updates of adjacent nodes. Preventing two adjacent nodes in the first place from picking a new color in the same round seems to be way too restrictive, in particular because it is unlikely that both nodes aim for the same new color. On the other hand, if all nodes update their colors simultaneously, a node is expected to have a conflict with at least one of its neighbors, which prevents progress.

We interpolate between the two extreme cases by introducing a marking probability, so that only a small fraction of a node's neighbors is expected to update the color, and hence also, in worst case, only these can conflict with its update. Concretely, we propose the following generic sampling method, which we call *Local Glauber Dynamics*: In every step, every variable independently marks itself at random with a certain (low) probability. If it is marked, it samples a proposal at random and checks with its neighbors whether the proposal leads to a conflict with their current state or their new proposals (if any). If there is a conflict, the variable rolls back and stays with its current state, otherwise the state is updated. As opposed to sequential sampling, where only one variable per step updates its value, here the expected number of variables simultaneously updating their value is  $\Omega(n)$ , resulting in the desired speed-up from  $O(n \log n)$ , say, to  $O(\log n)$ . Of course, the main technical aspect lies in showing that this simple update rule converges to the uniform distribution in  $O(\log n)$  rounds, which we prove in Section 2.

### 1.3 Notation and Preliminaries

**Model.** We work with the standard distributed message-passing model for the study of *locality*: the LOCAL model introduced by Linial [14], defined as follows. Given a graph  $G = (V, E)$  on  $n$  nodes with maximum degree  $\Delta$ , the computation proceeds in rounds. In every round, every node can send a message to each of its neighbors. We do not limit the message sizes, but for the algorithm that we present,  $O(\log n)$ -bit messages suffice. In the end of the computation, every node  $v$  outputs a color. The quantity of main interest is the round complexity, i.e., the number of rounds until the joint output of all nodes satisfies a certain condition.

**Markov Chain.** We consider a Markov chain  $\mathbf{X} = (X^{(t)})_{t \geq 0}$ , where  $X^{(t)} = (X_v^{(t)})_{v \in V} \in [q]^V$  is the coloring of the graph in round  $t$ . We will omit the round index, and use  $X = (X_v)_{v \in V} \in [q]^V$  for the coloring at time  $t$  and  $X' = (X'_v)_{v \in V} \in [q]^V$  for the coloring at time  $t + 1$ , for a  $t \geq 0$ , instead.

**Mixing Time.** For a Markov chain  $(X^{(t)})_{t \geq 0}$  with stationary distribution  $\mu$ , let  $\pi_\sigma^{(t)}$  denote the distribution of the random coloring  $X^{(t)}$  of the chain at time  $t \geq 0$ , conditioned on  $X^{(0)} = \sigma$ . The mixing time  $\tau_{\text{mix}}(\varepsilon) = \max_{\sigma \in \Omega} \min \left\{ t \geq 0 : d_{\text{TV}} \left( \pi_\sigma^{(t)}, \mu \right) \leq \varepsilon \right\}$  is defined to be the minimum number of rounds needed so that the Markov chain is  $\varepsilon$ -close (in

terms of total variation distance) to its stationary distribution  $\mu$ , regardless of  $X^{(0)}$ . The total variation distance between two distributions  $\mu, \nu$  over  $\Omega$  is defined as  $d_{\text{TV}}(\mu, \nu) = \sum_{\sigma \in \Omega} \frac{1}{2} |\mu(\sigma) - \nu(\sigma)|$ .

**Path Coupling.** The *Path Coupling Lemma* by Bubley and Dyer [2, Theorem 1] (also see [7, Lemma 4.3]) gives rise to a particularly easy way of designing couplings. In a simplified version, it says that it is enough to define the coupling of a Markov chain only for pairs of colorings that are adjacent, that is, differ at exactly one node. The expected number of differing nodes after one coupling step then can be used to bound the mixing time of the Markov chain.

► **Lemma 2** (Path Coupling [2], simplified). *For  $\sigma, \sigma' \in [q]^V$ , let  $\phi(\sigma, \sigma') := |\{v \in V : \sigma_v \neq \sigma'_v\}|$ . If there is a coupling  $(X, Y) \rightarrow (X', Y')$  of the Markov chain, defined only for  $(X, Y)$  with  $\phi(X, Y) = 1$ , that satisfies  $\mathbb{E}[\phi(X', Y') \mid X, Y] \leq 1 - \delta$  for some  $0 < \delta < 1$ , then  $\tau_{\text{mix}}(\varepsilon) = O\left(\frac{1}{\delta} \cdot \log\left(\frac{n}{\varepsilon}\right)\right)$ .*

## 2 Local Glauber Dynamics

**Local Glauber Dynamics.** We define a transition from  $X = (X_v)_{v \in V}$  to  $X' = (X'_v)_{v \in V}$  in one round as follows. Every node  $v \in V$  marks itself independently with probability  $0 < \gamma < 1$ . If it is marked, it proposes a new color  $c_v \in [q]$  uniformly at random, independently from all the other nodes. If this proposed color does not lead to a conflict with the current and the proposed colors of any neighbor, that is,  $c_v \notin \bigcup_{u \in N(v)} \{X_u, c_u\}$  and  $c_u \notin \{X_v, c_v\}$  for any  $u \in N(v)$ <sup>3</sup>, then  $v$  accepts color  $c_v$ , thus sets  $X'_v = c_v$ . Otherwise,  $v$  keeps its current color, that is, sets  $X'_v = X_v$ . Note that the condition  $c_v \notin \bigcup_{u \in N(v)} \{X_u, c_u\}$  is necessary to guarantee reversibility of the Markov chain.

**Stationary Distribution.** The local Glauber dynamics is ergodic: it is aperiodic, as there is always a positive probability of not changing any of the colors, and irreducible, since any (proper) coloring can be reached from any coloring. Moreover, the chain might possibly start from an improper coloring, but it will never move from a proper to an improper coloring, that is, it is absorbing to proper colorings. It is easy to verify that this local Glauber dynamics, due to its symmetric update rule, satisfies the detailed balance equation for the uniform distribution, meaning that the transition from  $X$  to  $X'$  has the same probability as a transition from  $X'$  to  $X$  for proper colorings. The chain thus is reversible and has the uniform distribution over all proper colorings as unique stationary distribution.

**Mixing Time.** Informally speaking, the Path Coupling Lemma says that if for all  $X$  and  $Y$  which differ in one node, we can define a coupling  $(X, Y) \rightarrow (X', Y')$  in such a way that the expected number of nodes at which  $X'$  and  $Y'$  differ is bounded away from 1 from above, then the chain converges quickly. In Section 2.1, we formally describe such a path coupling, in Section 2.2, we list necessary (but not necessarily sufficient) conditions for a node to have two different colors after one coupling step, which is then used in Section 2.3 to bound the expected number of differing nodes by  $1 - \delta$  for some constant  $0 < \delta < 1$ , depending on  $\alpha$ . Application of Lemma 2 then concludes the proof of Theorem 1.

<sup>3</sup> To simplify notation, we assume that  $c_u = X_u$  in case  $u$  is not marked.

## 2.1 Description of Path Coupling.

We look at two colorings  $X$  and  $Y$  that differ at a node  $v_0 \in V$  only. That is,  $r = X_{v_0} \neq Y_{v_0} = b$ , for some  $r \neq b \in [q]$ , which we will naturally refer to as red and blue, respectively, and  $X_v = Y_v$  for all  $v \neq v_0 \in V$ . In the following, we explain how every node  $v \in V$  comes up with a pair  $(c_v^X, c_v^Y)$  of new proposals, which then will be accepted or rejected based on the local Glauber dynamics rules.

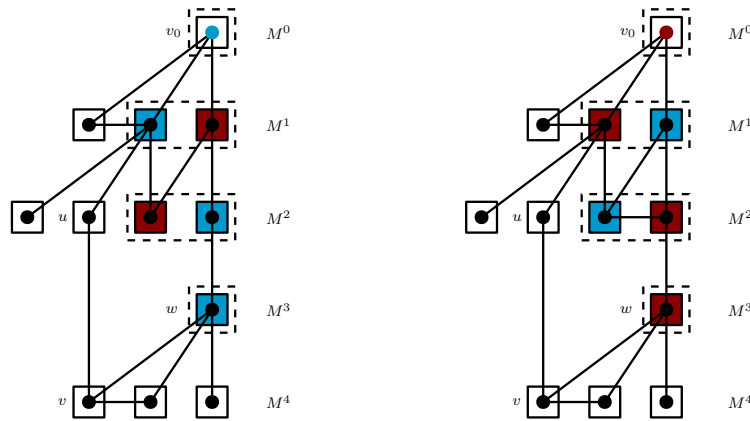
**Marking.** In both chains, every node  $v \in V$  is marked independently with probability  $\gamma$ , using the same randomness in both chains. In the following, we restrict our attention to marked nodes only; non-marked nodes are thought of proposing their current color as new color, i.e.,  $c_v^X = X_v$  and  $c_v^Y = Y_v$ .

**Consistent, Mirrored, and Flipped Proposals.** We introduce two possible ways of how proposals for a node  $v$  can be sampled: *consistently* and *mirroredly*. For the consistent proposals, both chains propose the same randomly chosen color, that is,  $c_v^X = c_v^Y = c$  for a u.a.r.  $c \in [q]$ . For the mirrored proposals, both chains assign the same random proposal if it is neither red nor blue, and a *flipped* proposal (i.e., red to one and blue to the other chain) otherwise. More formally,  $c_v^X = c$  and  $c_v^Y = \bar{c}$  if  $c \in \{r, b\}$  and  $\bar{c}$  the element in  $\{r, b\} \setminus \{c\}$ , and  $c_v^X = c_v^Y = c$  if  $c \notin \{r, b\}$ , for a u.a.r.  $c \in [q]$ . We say that  $v$  has *flipped* proposals if  $c_v^X \neq c_v^Y$ . Note that we say mirrored proposal to refer to the process of sampling mirroredly, and we say flipped if, as a result of sampling mirroredly, a node proposes different colors in the two chains.

**Breadth-First Assignment of Proposals.** Let  $B = \{v \in V \setminus \{v_0\} : X_v \in \{r, b\}\} \subseteq V \setminus \{v_0\}$  be the set of nodes  $v \neq v_0$  with current color red or blue, as well as  $K = (\bigcup_{v \in B} N^+(v)) \setminus \{v_0\}$  its inclusive neighborhood, without  $v_0$ , where  $N^+(v) := N(v) \cup \{v\}$ . We ignore this set  $K$  for the moment, and focus on the set  $S \subseteq V \setminus K$  of marked nodes that are not adjacent to a node with color red or blue (except for possibly  $v_0$ ). Informally speaking, we will go through these nodes in a breadth-first manner, with increasing distance  $d \geq 0$  to node  $v_0$ , and fix their proposals layer by layer, but defer the assignment of nodes not (yet) adjacent to a node with flipped proposals, as follows. We repeatedly add all (still remaining) nodes that have a node in the last layer with flipped proposals to a new layer, and sample their proposals mirroredly, thus perform a breadth-first assignment on nodes with flipped proposals only. All remaining nodes sample their proposals consistently. Note that this in particular guarantees that a node is sampled consistently only if it not adjacent to a node with flipped proposals.

More formally, this can be described as follows. We define  $M^0 = F^0 = \{v_0\}$ , even if  $v_0$  is not marked, and  $M^1 = N(v_0)$ . For the subsequent layer, we restrict the attention to (new) neighbors of nodes in  $M^d$  with flipped proposals only, i.e., consider  $M^{d+1} = N(F^d) \setminus \bigcup_{i=0}^d M^i$  for  $F^d = \{v \in M^d : c_v^X \neq c_v^Y\}$ . For node  $v_0$ , if marked, the proposals are sampled consistently. For  $d \geq 1$  and  $v \in M^d$ , the proposals are sampled mirroredly. For all remaining (marked) nodes, that is, nodes in  $S \setminus M$  and nodes in  $K$ , proposals are sampled consistently. See Figure 1 for an illustration of this breadth-first-based approach.

**Accept Proposals.** The proposals  $(c_v^X)_{v \in V}$  and  $(c_v^Y)_{v \in V}$  in the chains  $X$  and  $Y$  are accepted or rejected based on the local Glauber dynamics rules, leading to colorings  $X', Y' \in [q]^V$ .



■ **Figure 1** The breadth-first layers  $M^d$  for  $d \geq 0$  of two chains that differ at  $v_0 \in M^0$ . The disk color corresponds to the node’s current color, where black means any color except red and blue. The color of the box around a node shows this node’s proposed color, where white stands for any color (possibly also red or blue, but consistent). Dashed boxes indicate the sets  $F^d$  of nodes with flipped proposals. Note that node  $v$  appears in layer 4 even though it has distance 3 to  $v_0$ . This is because we perform the breadth-first assignment only on nodes with flipped proposals.  $v$ ’s neighbor  $u$  does not have flipped proposals, thus is in  $M^2 \setminus F^2$ , which means that  $u$ ’s neighbors are not added to the next layer. Only  $v$ ’s neighbor  $w \in F^3$  leads to  $v$  being added to  $M^4$ .

## 2.2 Properties of the Coupling

The main observation is the following. If we ignore nodes with current colors red and blue for the moment, one can argue that  $X'$  and  $Y'$  can only differ at a node different from  $v_0$  if its proposals are flipped. Flipped proposals, however, can only arise when the proposals are sampled mirroredly, which happens only if there is a node in the preceding layer with flipped proposals (due to the breadth-first order in which we assign the proposals). A node thus can lead to an inconsistency only if there is path in  $G$  from  $v_0$  to this node consisting of nodes with flipped proposals, called a *flip path*.

We will next make this intuition with the flip paths more precise, in two parts: for nodes in  $S$  (that sample their proposals mirroredly if adjacent to a node with flipped proposals) in Lemma 3 and for nodes in  $K$  (that always sample their proposals consistently) in Lemma 4. See Figure 2 for an illustration of these two cases.

► **Lemma 3.** *If  $X'$  and  $Y'$  differ at  $v \neq v_0 \in S$ , there is a flip path  $(v_0, \dots, v_\ell = v) \in F^0 \times \dots \times F^\ell$  of length  $\ell \geq 1$  in  $G$ , with the additional property that the proposal of  $v$  is the opposite of the last color (red or blue) seen on this path, in both chains. More formally,  $c_Y = c_v^X \neq c_v^Y = c_X$ , where  $c_X = c_{v_{\ell-1}}^X$  and  $c_Y = c_{v_{\ell-1}}^Y$  if  $\ell > 1$ , and  $c_X = X_{v_0}$  and  $c_Y = Y_{v_0}$  if  $\ell = 1$ .*

**Proof.** We first argue that  $v$ ’s proposals must be flipped and accepted in both chains. Trivially, acceptance of a consistent proposal in both chains or rejection in both chains leads to  $X'_v = Y'_v$ . Moreover, observe that flipped proposals are, by construction, either accepted in both or rejected in both chains, as flipping changes the role of red and blue, but not the overall behavior. Indeed, suppose, without loss of generality, that  $c_v^X = c \in \{r, b\}$  is rejected by  $X$ . Thus, in particular,  $v$  has a neighbor  $u$  with current color or proposal  $c$  in  $X$ . As we are restricting our attention to the set  $S$  which does not have any adjacent node with current color red or blue, except for  $v_0$ , either  $u = v_0$  or  $u$  proposes  $c$ . So  $u$  either must have different current colors (if  $u = v_0$ ) or have mirrored proposals (if  $v \in F^d$ , then  $u \in M^{d'}$  for

some  $d' \leq d + 1$ , because at the latest  $v$ 's flipped proposal leads to  $u$  being added to the subsequent layer, by how we assign the proposals in breadth-first manner) and hence flipped proposals. Thus,  $v$ 's proposal  $\bar{c}$  in  $Y$  will be rejected by  $Y$ , since either  $u = v_0 \in N(v)$  has color  $\bar{c}$  or  $u \in N(v)$  proposes  $\bar{c}$ .

It thus remains to rule out the case of consistent proposals that are accepted in one and rejected in the other chain. Towards a contradiction, suppose that  $v$  proposes the same color  $c_v$  in both chains, and that it is accepted in one and rejected in the other. Since  $X_v = Y_v$  and  $c_v^X = c_v^Y$ , this can happen only if  $v$  is adjacent either to  $v_0$  or to at least one node with flipped proposals, as otherwise all proposals and all current colors in  $v$ 's inclusive neighborhood would be the same, leading to the same behavior in both chains. In both cases,  $v \in M^d$  for some  $d \geq 1$ , which means that its proposals are sampled mirroredly. Hence,  $c_v \notin \{r, b\}$ , as otherwise the proposals would be flipped. Now, since neither  $v$ 's current color nor  $v$ 's proposals is red or blue, and neighbors of  $v$  can differ in their colors or proposals only if red or blue is involved, the proposals are either accepted or rejected in both chains. It follows that indeed only nodes in  $S$  with flipped proposals that are accepted in both chains can have different colors in  $X'$  and  $Y'$ .

By construction of the layers, and since  $v \in F^\ell$  for some  $\ell \geq 1$ , there must exist a sequence of nodes  $v_1 \in F^1, \dots, v_{\ell-1} \in F^{\ell-1}$  connecting  $v_0$  to  $v$  in  $G$ : a flip path of length  $\ell$ . Moreover, the proposal is accepted in a chain only if the proposed color is the opposite of the color (red or blue) that is seen on the path (either as proposal if  $\ell > 1$ , or as current color of  $v_0$  if  $\ell = 1$ ). ◀

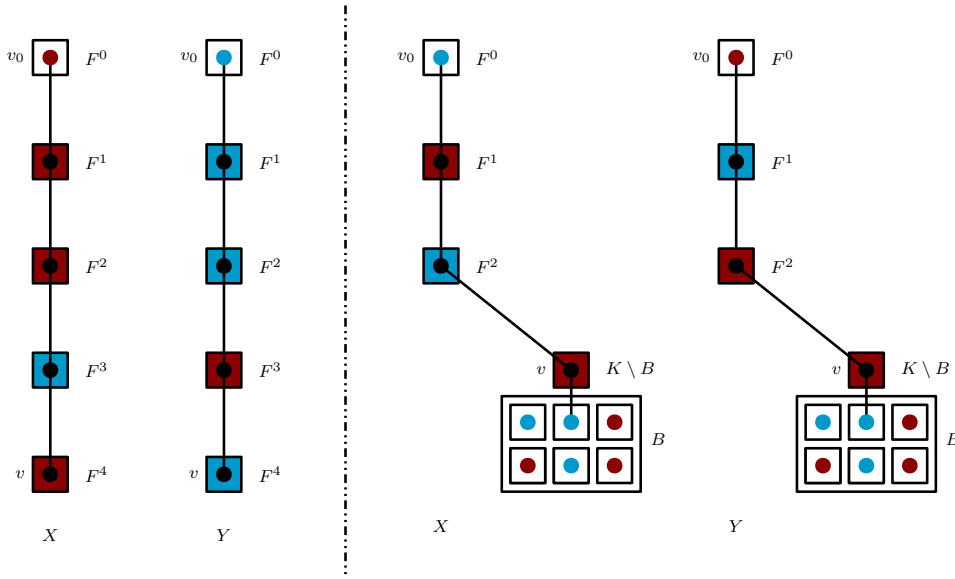
► **Lemma 4.** *If  $X'$  and  $Y'$  differ at  $v \neq v_0 \in K$ , there is a path  $(v_0, \dots, v_\ell = v) \in F^0 \times \dots \times F^{\ell-1} \times K$  of length  $\ell \geq 1$  in  $G$ , called an almost flip path, with the additional property that the proposal of  $v$  is either red or blue, that is,  $c_v = c_v^X = c_v^Y \in \{r, b\}$ .*

**Proof.** Since, by definition of the coupling,  $v \in K$  samples its proposals consistently,  $X'$  and  $Y'$  can only differ at  $v \neq v_0$  if the proposal is accepted in one and rejected in the other chain. This can happen only if  $v$  is adjacent to either  $v_0$  or to at least one node with flipped proposals. Otherwise, all proposals and all current colors in  $v$ 's inclusive neighborhood would be the same, leading to the same behavior. Hence,  $v$  is adjacent to some  $u \in F^d$  for some  $d \geq 0$ . By construction of the layers, there must exist a sequence of nodes  $v_1 \in F^1, \dots, v_{\ell-1} = u \in F^{\ell-1}$  connecting  $v_0$  to  $v$  in  $G$ : an almost flip path of length  $\ell = d + 1$ . Note that, in particular, because neighbors of nodes in  $B$  are by definition sampled consistently (as they are in  $K$ ), and a node at the end of an almost flip path has a neighbor with flipped proposals, this last node on an almost flip path must be in  $K \setminus B$ .

The proposal  $c_v$  is accepted in one and rejected in the other chain only if  $c_v \in \{r, b\}$ . In that case, the chain with the same color on the end of the path will reject, the other will (possibly) accept. ◀

### 2.3 Bounding the Expected Number of Differing Nodes

We show that  $\mathbb{E}[\phi(X', Y') \mid X, Y] \leq 1 - \delta$  for some  $0 < \delta < 1$ , by bounding the expectations  $\mathbb{E}[\sum_{v \neq v_0 \in V} 1(X'_v \neq Y'_v) \mid X, Y]$  and  $\mathbb{E}[1(X'_{v_0} \neq Y'_{v_0}) \mid X, Y]$  separately. We will see that, as  $\delta \rightarrow 0$ , both terms can be bounded by  $\approx \frac{1}{\alpha}$ , leading to an expected number of roughly  $\frac{2}{\alpha}$ , which is strictly less than 1 for  $\alpha > 2$ .



■ **Figure 2** A flip path on the left:  $v$ 's flipped proposals are accepted in both chains, yielding  $X'_v = r$  and  $Y'_v = b$ .

An almost flip path on the right:  $v \in K \setminus B$  samples its proposals consistently. In chain  $X$ , the proposal  $r$  will be accepted, in chain  $Y$ , it will be rejected, leading to  $X'_v = r \neq Y'_v = b$ . The disk color corresponds to the node's current color, where black means any color except red and blue. The color of the box around a node indicates this node's proposed color, where white means any color (also red and blue, but consistent).

**Nodes  $v \neq v_0$ .** Section 2.2, or more precisely, Lemmas 3 and 4, show that the number of nodes (different from  $v_0$ ) that have different colors in  $X'$  and  $Y'$  can be bounded by the number of (almost) flip paths with an additional property. We will next see that the expected number of such (almost) flip paths can be expressed as a geometric series summing over the depths of the layers.

There are at most  $\Delta^\ell$  paths  $(v_0, \dots, v_\ell)$  of length  $\ell$  in  $G$ . Moreover, each such path has probability  $(2\gamma/q)^{\ell-1} \gamma/q$  of being a flip or almost flip path with the mentioned additional property, since all intermediate nodes  $v_1, \dots, v_{\ell-1}$  need to mark themselves and to propose one arbitrary color in  $\{r, b\}$ , and  $v_\ell$  needs to mark itself and to propose the one color in  $\{r, b\}$  as specified in Lemmas 3 and 4, respectively. Note that a path in  $G$  can either be a flip path or an almost flip path, but never both. Moreover, observe that node  $v_0$  does not need to be marked. We get

$$\mathbb{E} \left[ \sum_{v \neq v_0 \in V} 1(X'_v \neq Y'_v) \mid X, Y \right] \leq \sum_{\ell=1}^{\infty} \Delta^\ell \cdot \left(\frac{2\gamma}{q}\right)^{\ell-1} \cdot \frac{\gamma}{q} = \frac{1}{2} \sum_{\ell=1}^{\infty} \left(\frac{2\gamma\Delta}{q}\right)^\ell \leq \frac{\frac{\gamma\Delta}{q}}{1 - \frac{2\gamma\Delta}{q}}. \quad (1)$$

**Node  $v_0$ .** Chains  $X'$  and  $Y'$  can agree at node  $v_0$  only if at least one the proposals is accepted. For that,  $v_0$  needs to be marked and its proposal  $c_{v_0} = c_{v_0}^X = c_{v_0}^Y$  needs to be different from all the at most  $\Delta$  current colors of its neighbors, that is,  $c_v \notin \bigcup_{v \in N(v_0)} \{X_v\}$ , which happens with probability at least  $\gamma(1 - \Delta/q)$ . Moreover, the proposals of  $v_0$ 's neighbors (if marked) need to avoid at most three colors in  $\{c_{v_0}, r, b\}$ , possibly less, which happens with probability at least  $1 - 3\gamma/q$ . We thus get

$$\mathbb{E} [1(X'_{v_0} \neq Y'_{v_0})] \leq 1 - \gamma \left(1 - \frac{\Delta}{q}\right) \left(1 - \frac{3\gamma}{q}\right)^\Delta. \quad (2)$$

**Wrap-Up.** Overall, combining Equations (1) and (2), we get

$$\mathbb{E}[\phi(X', Y') \mid X, Y] \leq 1 - \gamma \left(1 - \frac{1}{\alpha}\right) e^{-\frac{6\gamma}{\alpha}} + \frac{\frac{\gamma}{\alpha}}{1 - \frac{2\gamma}{\alpha}} = 1 - \gamma e^{-\frac{6\gamma}{\alpha}} \left(1 - \frac{1}{\alpha} \left(1 + \frac{e^{\frac{6\gamma}{\alpha}}}{1 - \frac{2\gamma}{\alpha}}\right)\right).$$

For  $\alpha > 2$  and  $\gamma := \gamma(\alpha)$  small enough, this is strictly bounded away from 1 from above, where the hidden constant depends on  $\alpha$  (but not on  $\Delta$  or  $n$ ).

---

## References

- 1 Christophe Andrieu, Nando De Freitas, Arnaud Doucet, and Michael I. Jordan. An Introduction to MCMC for Machine Learning. *Machine Learning*, 50(1-2):5–43, 2003.
- 2 Russ Bubley and Martin Dyer. Path Coupling: A Technique for Proving Rapid Mixing in Markov Chains. In *the Proceedings of the Symposium on Foundations of Computer Science (FOCS)*, pages 223–231, 1997.
- 3 Sitan Chen and Ankur Moitra. Linear Programming Bounds for Randomly Sampling Colorings. *arXiv preprint arXiv:1804.03156*, 2018.
- 4 Michelle Delcourt, Guillem Perarnau, and Luke Postle. Rapid Mixing of Glauber Dynamics for Colorings Below Vigoda’s 11/6 Threshold. *arXiv preprint arXiv:1804.04025*, 2018.
- 5 Roland L. Dobruschin. The Description of a Random Field by Means of Conditional Probabilities and Conditions of its Regularity. *Theory of Probability & Its Applications*, 13(2):197–224, 1968.
- 6 Weiming Feng, Thomas P. Hayes, and Yitong Yin. Distributed Symmetry Breaking in Sampling (Optimal Distributed Randomly Coloring with Fewer Colors). *arXiv preprint arXiv:1802.06953*, 2018.
- 7 Weiming Feng, Yuxin Sun, and Yitong Yin. What Can Be Sampled Locally? In *Proceedings of the International Symposium on Principles of Distributed Computing (PODC)*, pages 121–130, 2017.
- 8 Alan Frieze and Eric Vigoda. A Survey on the Use of Markov Chains to Randomly Sample Colourings. *Oxford Lecture Series in Mathematics and its Applications*, 34:53, 2007.
- 9 Joseph Gonzalez, Yucheng Low, Arthur Gretton, and Carlos Guestrin. Parallel Gibbs Sampling: From Colored Fields to Thin Junction Trees. In *the Proceedings of the International Conference on Artificial Intelligence and Statistics*, pages 324–332, 2011.
- 10 Heng Guo, Mark Jerrum, and Jingcheng Liu. Uniform Sampling Through the Lovász Local Lemma. *Proceedings of the Symposium on Theory of Computing (STOC)*, pages 342–355, 2017.
- 11 J. M. Hammersley and P. Clifford. Markov Fields on Finite Graphs and Lattices. Unpublished, available at <http://www.statslab.cam.ac.uk/~grg/books/hammfest/hamm-cliff.pdf>, 1971.
- 12 Mark Jerrum. A Very Simple Algorithm for Estimating the Number of k-Colorings of a Low-Degree Graph. *Random Structures & Algorithms*, 7(2):157–165, 1995.
- 13 Mark R. Jerrum, Leslie G. Valiant, and Vijay V. Vazirani. Random Generation of Combinatorial Structures from a Uniform Distribution. *Theoretical Computer Science*, 43:169–188, 1986.
- 14 Nathan Linial. Distributive Graph Algorithms - Global Solutions From Local Data. In *the Proceedings of the Symposium on Foundations of Computer Science (FOCS)*, pages 331–335. IEEE, 1987.
- 15 Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of State Calculations by Fast Computing Machines. *The Journal of Chemical Physics*, 21(6):1087–1092, 1953.



- 16 Surendra Nahar, Sartaj Sahni, and Eugene Shragowitz. Simulated Annealing and Combinatorial Optimization. In *Proceedings of the Design Automation Conference*, pages 293–299. IEEE Press, 1986.
- 17 Moni Naor and Larry Stockmeyer. What Can Be Computed Locally? *SIAM Journal on Computing*, 24(6):1259–1277, 1995.
- 18 David Newman, Padhraic Smyth, Max Welling, and Arthur U. Asuncion. Distributed Inference for Latent Dirichlet Allocation. In *Advances in Neural Information Processing Systems*, pages 1081–1088, 2008.
- 19 Jesús Salas and Alan D. Sokal. Absence of Phase Transition for Antiferromagnetic Potts Models via the Dobrushin Uniqueness Theorem. *Journal of Statistical Physics*, 86(3-4):551–579, 1997.
- 20 Eric Vigoda. Improved Bounds for Sampling Colorings. *Journal of Mathematical Physics*, 41(3):1555–1569, 2000.
- 21 Feng Yan, Ningyi Xu, and Yuan Qi. Parallel Inference for Latent Dirichlet Allocation on Graphics Processing Units. In *Advances in Neural Information Processing Systems*, pages 2134–2142, 2009.



# Fast Multidimensional Asymptotic and Approximate Consensus

**Matthias Függer**

CNRS, LSV, ENS Paris-Saclay, Université Paris-Saclay, and Inria, France  
mfuegger@lsv.fr

**Thomas Nowak**

Université Paris-Sud, France  
thomas.nowak@lri.fr

---

## Abstract

We study the problems of asymptotic and approximate consensus in which agents have to get their values arbitrarily close to each others' inside the convex hull of initial values, either without or with an explicit decision by the agents. In particular, we are concerned with the case of multidimensional data, i.e., the agents' values are  $d$ -dimensional vectors. We introduce two new algorithms for dynamic networks, subsuming classical failure models like asynchronous message passing systems with Byzantine agents. The algorithms are the first to have a contraction rate and time complexity independent of the dimension  $d$ . In particular, we improve the time complexity from the previously fastest approximate consensus algorithm in asynchronous message passing systems with Byzantine faults by Mendes et al. [Distrib. Comput. 28] from  $\Omega(d \log \frac{d\Delta}{\varepsilon})$  to  $O(\log \frac{\Delta}{\varepsilon})$ , where  $\Delta$  is the initial and  $\varepsilon$  is the terminal diameter of the set of vectors of correct agents.

**2012 ACM Subject Classification** Theory of computation → Distributed algorithms

**Keywords and phrases** asymptotic consensus, approximate consensus, multidimensional data, dynamic networks, Byzantine processes

**Digital Object Identifier** 10.4230/LIPIcs.DISC.2018.27

**Funding** This research was partially supported by the CNRS project PEPS DEMO and the Institut Farman.

## 1 Introduction

The problem of one-dimensional asymptotic consensus requires a system of agents, starting from potentially different initial real values, to repeatedly set their local output variables such that all outputs converge to a common value within the convex hull of the inputs. This problem has been studied in distributed control theory both from a theoretical perspective [10, 19, 5, 2] and in the context of robot gathering on a line [3] and clock synchronization [20, 16]. Extensions of the problem to multidimensional values naturally arise in the context of robot gathering on a plane or three-dimensional space [11], as subroutines in formation forming [10], and distributed optimization [4], among others.

The related problem of approximate consensus, also called approximate agreement, requires the agents to eventually decide, i.e., to only set their output variables once. Additionally all output variables must be within a predefined  $\varepsilon > 0$  distance of each other and lie within the convex hull of the inputs. There is a large body of work on approximate consensus in distributed computing devoted to solvability and optimality of time complexity [13, 14] and applications in clock synchronization; see e.g. [24, 23].



© Matthias Függer and Thomas Nowak;  
licensed under Creative Commons License CC-BY  
32nd International Symposium on Distributed Computing (DISC 2018).  
Editors: Ulrich Schmid and Josef Widder; Article No. 27; pp. 27:1–27:16  
Leibniz International Proceedings in Informatics



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

Both problems were studied under different assumptions on the underlying communication between agents and their computational strength, including fully connected asynchronous message passing with Byzantine agents [24, 13] and communication in rounds by message passing in dynamic communication networks [19, 10]. In [6, 7] Charron-Bost et al. analyzed solvability of asymptotic consensus and approximate consensus in dynamic networks with round-wise message passing defined by *network models*: a network model is a set of directed communication graphs, each of which specifies successful reception of broadcast messages; see Section 2.1 for a formal definition. Solving asymptotic consensus in such a model requires to fulfill the specification of asymptotic consensus in *any* sequence of communication graphs from the model. Charron-Bost et al. showed that in these highly dynamic networks, asymptotic consensus and approximate consensus are solvable in a network model if and only if each of its graphs contains a spanning rooted tree. An interesting class of network models are those that contain only *non-split* communication graphs, i.e., communication graphs where each pair of nodes has a common incoming neighbor. Several classical fault-models were shown to be instances of non-split models [6], among them asynchronous message passing systems with omissions.

Recently the multidimensional version of approximate consensus received attention. Mendes et al. [18] were the first to present algorithms that solve approximate consensus in Byzantine message passing systems for  $d$ -dimensional real vectors. Their algorithms, Mendes–Herlihy and Vaidya–Garg, are based on the repeated construction of so called safe areas of received vectors to constraint influence of values sent by Byzantine agents, followed by an update step, ensuring that the new output values are in the safe area. They showed that the diameter of output values contracts by at least  $1/2$  in each dimension every  $d$  rounds in the Mendes–Herlihy algorithm, and the diameter of the output values contracts by at least  $1 - 1/n$  every round in the Vaidya–Garg algorithm, where  $n$  is the number of agents. The latter bound assumes  $f = 0$  Byzantine failures and slightly worsens for  $f > 0$ . In terms of contraction rates as introduced in [15] (see Section 2.3 for a definition) of the respective non-terminating algorithms for asymptotic consensus, they thus obtain upper bounds of  $\sqrt[d]{1/2}$  and  $1 - 1/n$ . Note that the Mendes–Herlihy algorithm has a contraction rate depending only on  $d$  but requires an a priori common coordinate system, and the algorithm’s outcome depends on the choice of this coordinate system. By contrast the Vaidya–Garg algorithm is coordinate-free, i.e., its outcome is invariant under coordinate transformations such as translation and rotation, but it has a contraction rate depending on  $n$ .

Charron-Bost et al. [8] analyzed convergence of the Centroid algorithm where agents repeatedly update their position to the centroid of the convex hull of received vectors. The algorithm is coordinate-free and has a contraction rate of  $d/(d + 1)$ , independent of  $n$ . Local time complexity of determining the centroid was shown to be #P-complete [21] while polynomial in  $n$  for fixed  $d$ .

The contraction rate of the Centroid algorithm is always smaller or equal to that of the Mendes–Herlihy algorithm, though both contraction rates converge to 1 at the same speed with the dimension  $d$  going to infinity. More precisely,

$$\lim_{d \rightarrow \infty} \frac{\left| 1 - \sqrt[d]{\frac{1}{2}} \right|}{\left| 1 - \frac{d}{d+1} \right|} = \log 2 \quad ,$$

which implies  $\left| 1 - \sqrt[d]{\frac{1}{2}} \right| = \Theta \left( \left| 1 - \frac{d}{d+1} \right| \right)$ .

■ **Table 1** Comparison of local time complexity and contraction rates in non-split network models. Entries marked with an \* are new results in this paper.

	MidExtremes	ApproachExtreme	Centroid	MH	VG
contraction rate	$\sqrt{\frac{7}{8}}^*$	$\sqrt{\frac{31}{32}}^*$	$\frac{d}{d+1}$	$\sqrt{\frac{1}{2}}$	$1 - \frac{1}{n}$
local TIME	$O(n^2d)$	$O(nd)$	#P-hard	$O(nd)$	$O(nd)$
coordinate-free	yes	yes	yes	no	yes

## 1.1 Contribution

In this work we present two new algorithms that are coordinate-free: the MidExtremes and the ApproachExtreme algorithm, and study their behavior in dynamic networks. Both algorithms are coordinate-free, operate in rounds, and are shown to solve asymptotic agreement in non-split network models. Terminating variants of them are shown to solve approximate agreement in non-split network models.

As a main result we prove that their contraction rate is independent of network size  $n$  and dimension  $d$  of the initial values. For MidExtremes we obtain an upper bound on the contraction rate of  $\sqrt{7/8}$  and for ApproachExtreme of  $\sqrt{31/32}$ .

Due to the fact that classical failure models like asynchronous message passing with Byzantine agents possess corresponding network models, our results directly yield improved algorithms for the latter failure models: In particular, we improve the time complexity from the previously fastest approximate consensus algorithm in asynchronous message passing systems with Byzantine faults, the Mendes–Herlihy algorithm, from  $\Omega(d \log \frac{d\Delta}{\varepsilon})$  to  $O(\log \frac{\Delta}{\varepsilon})$ , where  $\Delta$  is the initial and  $\varepsilon$  is the terminal diameter of the set of vectors of correct agents. Note that our algorithms share the benefit of being coordinate-free with the Vaidya–Garg algorithm presented in the same work.

Table 1 summarizes our results and the algorithms discussed above for asymptotic and approximate consensus. The table compares the new algorithms MidExtremes and ApproachExtreme to the Centroid, Mendes–Herlihy (MH), and Vaidya–Garg (VG) algorithms with respect to their local time complexity per agent and round and an upper bound on their contraction rate in non-split network models. A lower bound of  $1/2$  on the contraction rate is due to Függer et al. [15].

The Mendes–Herlihy algorithm has a smaller contraction rate than the MidExtremes algorithm whenever  $d \leq 10$ ; the Centroid algorithm whenever  $d \leq 14$ . The Centroid algorithm is hence the currently fastest known algorithm for dimensions  $3 \leq d \leq 14$ . For dimensions  $d = 1$  and  $d = 2$ , the componentwise MidPoint algorithm has an optimal contraction rate of  $1/2$  [8]. Note that the MidExtremes algorithm is equivalent to the componentwise MidPoint algorithm for dimension  $d = 1$ . For  $d \geq 15$ , the MidExtremes algorithm is the currently fastest known algorithm.

We finally note that all our results hold for the class of inner product spaces and are not restricted to the finite-dimensional Euclidean spaces  $\mathbb{R}^d$ , in contrast to previous work. For example, this includes the set of square-integrable functions on a real interval. However, finite value representation and means to calculate the norm have to be guaranteed. Further, local TIME becomes  $n^2$ , respectively,  $n$  norm calculations.

## 2 Model and Problem

We fix some vector space  $V$  with an inner product  $\langle \cdot, \cdot \rangle : V \times V \rightarrow \mathbb{R}$  and the norm  $\|x\| = \sqrt{\langle x, x \rangle}$ . The prototypical finite-dimensional example is  $V = \mathbb{R}^d$  with the usual inner product and the Euclidean norm. The diameter of set  $A \subseteq V$  is denoted by  $\text{diam}(A) = \sup_{x, y \in A} \|x - y\|$ . For an  $n$ -tuple  $x = (x_1, \dots, x_n) \in V^n$  of vectors in  $V$ , we write  $\text{diam}(x)$  by slight abuse of notation to denote  $\text{diam}(\{x_1, \dots, x_n\})$ .

### 2.1 Dynamic Network Model

We consider a distributed system of  $n$  agents that communicate in rounds via message passing, like in the Heard-Of model [9]. In each round, each agent  $i$ , broadcasts a message based on its local state, receives some messages, and then updates its local state based on the received messages and its local state. Rounds are communication closed: agents only receive messages sent in the same round.

In each round  $t \geq 0$ , messages are delivered according to the *directed communication graph*  $G_t$  for round  $t$ : the message broadcast by  $i$  in round  $t$  is received by  $j$  if and only if the directed edge  $(i, j)$  is in  $G_t$ . Agents always receive their own messages, i.e.,  $(i, i) \in G_t$ . A *communication pattern* is an infinite sequence  $G_1, G_2, \dots$  of communication graphs. A (deterministic) *algorithm* specifies, for each agent  $i$ , the local state space of  $i$ , the set of initial states of  $i$ , the sending function for which message to broadcast, and the state transition function. For asymptotic consensus, each agent  $i$ 's local state necessarily contains a variable  $y_i \in V$ , which initially holds  $i$ 's input value and is then used as its output variable. We require that there is an initial state with initial value  $v$  for all vectors  $v \in V$ . A *configuration* is an  $n$ -tuple of local states. It is called initial if all local states are initial. The *execution* of an algorithm from initial configuration  $C_0$  induced by communication pattern  $G_1, G_2, \dots$  is the unique sequence  $C_0, G_1, C_1, G_2, C_2, \dots$  alternating between configurations and communication graphs where  $C_t$  is the configuration obtained by delivering messages in round  $t$  according to communication graph  $G_t$ , and applying the sending and local transition functions to the local states in  $C_{t-1}$  according to the algorithm. For a fixed execution and a local variable  $z$  of the algorithm, we denote by  $z_i(t)$  its value at  $i$  at the end of round  $t$ , i.e., in configuration  $C_t$ . In particular,  $y_i(t)$  is the value of  $y_i$  in  $C_t$ . We write  $y(t) = (y_1(t), \dots, y_n(t))$  for the collection of the  $y_i(t)$ .

A specific class of algorithms for asymptotic consensus are the so-called *convex combination*, or *averaging*, algorithm, which only ever update the value of  $y_i$  inside the convex hull of  $y_j$  it received from other agents  $j$  in the current round. Many algorithms in the literature belong to this class, as do ours.

Following [6], we study the behavior of algorithms for communication patterns from a *network model*, i.e., a non-empty set of communication patterns: a communication pattern is from network model  $\mathcal{N}$  if all its communication graphs are in  $\mathcal{N}$ . We will later on show that such an analysis also allows to prove new performance bounds for more classical fault-models like asynchronous message passing systems with Byzantine agents.

An interesting class of network models are so called *non-split* models, i.e., those that contain only non-split communication graphs: a communication graph is non-split if every pair of nodes has a common in-neighbor. Charron-Bost et al. [6] showed that asymptotic and approximate consensus is solvable efficiently in these network models in the case of one dimensional values. They further showed that: (i) In the weakest (i.e., largest) network model in which asymptotic and approximate consensus are solvable, the network model of

all communication graphs that contain a rooted spanning tree, one can simulate non-split communication graphs. (ii) Classical failure models like link failures as considered in [22] and asynchronous message passing systems with crash failures have non-split interpretations. Indeed we will make use of such a reduction from non-split network models to asynchronous message passing systems with Byzantine failures in Section 3.2.

## 2.2 Problem Formulation

An algorithm *solves the asymptotic consensus problem* in a network model  $\mathcal{N}$  if the following holds for every execution with a communication pattern from  $\mathcal{N}$ :

- *Convergence.* For every agent  $i$ , the sequence  $(y_i(t))_{t \geq 0}$  converges.
- *Agreement.* If  $y_i(t)$  and  $y_j(t)$  converge, then they have a common limit.
- *Validity.* If  $y_i(t)$  converges, then its limit is in the convex hull of the initial values  $y_1(0), \dots, y_n(0)$ .

For the deciding version, the *approximate consensus* problem (see, e.g., [17]), we augment the local state of  $i$  with a variable  $d_i$  initialized to  $\perp$ . Agent  $i$  is allowed to set  $d_i$  to some value  $v \neq \perp$  only once, in which case we say that  $i$  *decides*  $v$ . In addition to the initial values  $y_i(0)$ , agents initially receive the error tolerance  $\varepsilon$  and an upper bound  $\Delta$  on the maximum distance of initial values. An algorithm *solves approximate consensus* in  $\mathcal{N}$  if for all  $\varepsilon > 0$  and all  $\Delta$ , each execution with a communication pattern in  $\mathcal{N}$  with initial diameter at most  $\Delta$  satisfies:

- *Termination.* Each agent eventually decides.
- $\varepsilon$ -*Agreement.* If agents  $i$  and  $j$  decide  $d_i$  and  $d_j$ , respectively, then  $\|d_i - d_j\| \leq \varepsilon$ .
- *Validity.* If agent  $i$  decides  $d_i$ , then  $d_i$  is in the convex hull of initial values  $y_1(0), \dots, y_n(0)$ .

## 2.3 Performance Metrics

A direct natural performance metric to assess the speed of convergence of agent outputs  $y$  along an execution is the *round-by-round convergence rate*

$$c(t) = \frac{\text{diam}(y(t))}{\text{diam}(y(t-1))}$$

for a given round  $t \geq 1$  in the respective execution. The round-by-round convergence rate is the supremum over all executions and rounds. While a uniform upper bound of  $\beta < 1$  on the round-by-round convergence rate establishes convergence of the outputs, this measure fails in establishing convergence and comparing speeds of convergence for several algorithms considered in literature that set their output values every  $k > 1$  rounds, or that do not converge during an initial phase.

The *convergence rate*, defined by

$$\limsup_{t \rightarrow \infty} \sqrt[t]{\text{diam}(y(t))} ,$$

allows a comparison in this case by measuring eventual amortized convergence speed. For example, an algorithm that eventually contracts by a factor  $\beta < 1$  every  $k \geq 1$  rounds has a convergence rate of  $\sqrt[k]{\beta}$ .

As a performance measure for general asymptotic consensus algorithms, where agents do not necessarily set their outputs  $y$  to within the convex hull of previously received values, [15] considered the *contraction rate*, measuring contraction of reachable output limits rather



than output values: Following [15], the *valency* of a configuration  $C$ , denoted by  $Y^*(C)$ , is defined as the set of limits of the values  $y_i$  in executions that include configuration  $C$ . If the execution is clear from the context, we abbreviate  $Y^*(t) = Y^*(C_t)$ . The *contraction rate* of an execution is then defined as

$$\limsup_{t \rightarrow \infty} \sqrt[t]{\text{diam}(Y^*(t))} .$$

The contraction rate of an algorithm in a network model is the supremum of the contraction rates of its executions. For convex combination algorithms, the contraction rate is always upper-bounded by its convergence rate, that is,

$$\limsup_{t \rightarrow \infty} \sqrt[t]{\text{diam}(Y^*(t))} \leq \limsup_{t \rightarrow \infty} \sqrt[t]{\text{diam}(y(t))} ,$$

since the set of reachable limits  $Y^*(t)$  at round  $t$  is contained in the set of output values  $\{y_1(t), \dots, y_n(t)\}$  at round  $t$  for these algorithms.

Clearly, an algorithm that guarantees a round-by-round convergence rate of  $c(t) \leq \beta$  also guarantees a convergence rate of at most  $\beta$ . Since both of our algorithms are convex combination algorithms, all our upper bounds on the round-by-round convergence rates are also upper bounds for the contraction rates.

The *convergence time* of a given execution measures the time from which on all values are guaranteed to be in an  $\varepsilon$  of each other. Formally, it is the function defined as

$$T(\varepsilon) = \min \{t \geq 0 \mid \forall \tau \geq t: \text{diam}(y(\tau)) \leq \varepsilon\} .$$

In an execution that satisfies  $c(t) \leq \beta$  for all  $t \geq 1$ , we have the bound  $T(\varepsilon) \leq \left\lceil \log_{1/\beta} \frac{\Delta}{\varepsilon} \right\rceil$  on the convergence time, where  $\Delta = \text{diam}(y(0))$  is the diameter of the set of initial values.

### 3 Algorithms

In this section, we introduce two new algorithms for solving asymptotic and approximate consensus in arbitrary inner product spaces with constant contraction rates. We present our algorithms and prove their correctness and bounds on their performance in non-split networks models. While we believe that this framework is the one in which our arguments are clearest, our results can be extended to a number of other models whose underlying communication graphs turn out to be, in fact, non-split. The following is a selection of these models:

- **Rooted network models:** This is the largest class of network models in which asymptotic and approximate consensus are solvable [6]. A network model is rooted if all its communication graphs include a directed rooted spanning tree, though not necessarily the same in all graphs. Although not every such communication graph is non-split, Charron-Bost et al. [6] showed that the cumulative communication graph over  $n - 1$  rounds in a rooted network model is always non-split. In such network models, one can use amortized versions [7] of the algorithms, which operate in macro-rounds of  $n - 1$  rounds each. If an algorithm has a contraction rate  $\beta$  in non-split network models, then its amortized version has contraction rate  $\sqrt[n]{\beta}$  in rooted network models. The amortized versions of our algorithms thus have contraction rates independent of the dimension of the data.
- **Omission faults:** In the omission fault model studied by Santoro and Widmayer [22], the adversary can delete up to  $t$  messages from a fully connected communication graph each round. If  $t \leq n - 1$ , then all communication graphs are non-split. If  $t \leq 2n - 3$ , then all communication graphs are rooted [6]. Our algorithms are hence applicable in both these cases and have contraction rates independent of the dimension.

---

**Algorithm 1** Asymptotic consensus algorithm MidExtremes for agent  $i$ .

---

**Initialization:**

1:  $y_i$  is the initial value in  $V$

**In round  $t \geq 1$  do:**

2: broadcast  $y_i$

3:  $\text{Rcv}_i \leftarrow$  set of received values

4:  $(a, b) \leftarrow \arg \max_{(a,b) \in \text{Rcv}_i^2} \|a - b\|$

5:  $y_i \leftarrow \frac{a + b}{2}$

---

- Asynchronous message passing with crash faults: Building asynchronous rounds atop of asynchronous message passing by waiting for  $n - f$  messages in each round, the resulting communication graphs are non-split as long as the number  $f$  of possible crashes is strictly smaller than  $n/2$ . We hence get a constant contraction rate using our algorithms also in this model. For  $f \geq n/2$ , a partition argument shows that neither asymptotic nor approximate consensus are solvable.
- Asynchronous message passing with Byzantine faults: Mendes et al. [18] showed that approximate consensus is solvable in asynchronous message passing systems with  $f$  Byzantine faults if and only if  $n > (d+2)f$  where  $d$  is the dimension of the data. The algorithms they presented construct a round structure whose communication graphs turn out to be non-split. Since the construction is not straightforward, we postpone the discussion of our algorithms in this model to Section 3.2.

### 3.1 Non-split Network Models

We now present our two new algorithms, MidExtremes and ApproachExtreme. Both operate in the following simple round structure: broadcast the current value  $y_i$  and then update it to a new value depending on the set  $\text{Rcv}_i$  of values  $y_j$  received from agents  $j$  in the current round. Both of them only need to calculate distances between values and form the midpoint between two values. In particular, we do not need to make any assumption on the dimension of the space of possible values for implementing the algorithms. We only need a distance and an affine structure, for calculating the midpoint. Our correctness proofs, however, rely on the fact that the distance function is a norm induced by an inner product.

Note that, although we present algorithms for asymptotic consensus, combined with our upper bounds on the convergence time, one can easily deduce versions for approximate consensus by having the agents decide after the upper bound. Our upper bounds only depend on the precision parameter  $\varepsilon$  and (an upper bound on) the initial diameter  $\Delta$ . While upper bounds on the initial diameter cannot be deduced during execution in general non-split network models, it can be done in specific models, like asynchronous message passing with Byzantine faults [18]. Otherwise, we need to assume an a priori known bound on the initial diameter to solve approximate consensus.

The algorithm MidExtremes, which is shown in Algorithm 1, updates its value  $y_i$  to the midpoint of a pair of extremal points of  $\text{Rcv}_i$  that realizes its diameter. In the worst case, it thus has to compare the distances of  $\Theta(n^2)$  pairs of values. For the specific case of Euclidean spaces  $V = \mathbb{R}^d$  stored in a component-wise representation, this amounts to  $O(n^2 d)$  local scalar operations for each agent in each round.

It turns out that we can show a round-by-round convergence rate of the MidExtremes algorithm independent of the dimension or the number of agents, namely  $\sqrt{7/8}$ . For the

---

**Algorithm 2** Asymptotic consensus algorithm ApproachExtreme for agent  $i$ .

---

**Initialization:**

1:  $y_i$  is the initial value in  $V$

**In round  $t \geq 1$  do:**

2: broadcast  $y_i$

3:  $\text{Rcv}_i \leftarrow$  set of received values

4:  $b \leftarrow \arg \max_{b \in \text{Rcv}_i} \|y_i - b\|$

5:  $y_i \leftarrow \frac{y_i + b}{2}$

---

specific case of values from the real line  $V = \mathbb{R}$ , it reduces to the MidPoint algorithm [7], whose contraction rate of  $1/2$  is known to be optimal [15].

► **Theorem 1.** *In any non-split network model with values from any inner product space, the MidExtremes algorithm guarantees a round-by-round convergence rate of  $c(t) \leq \sqrt{7/8}$  for all rounds  $t \geq 1$ . Its convergence time is at most  $T(\varepsilon) = \left\lceil \log_{\sqrt{8/7}} \frac{\Delta}{\varepsilon} \right\rceil$  where  $\Delta$  is the diameter of the set of initial values.*

*In the particular case of values from the real line, it guarantees a round-by-round convergence rate of  $c(t) \leq 1/2$  and a convergence time of  $T(\varepsilon) = \left\lceil \log_2 \frac{\Delta}{\varepsilon} \right\rceil$ .*

The second algorithm we present is called ApproachExtreme and shown in Algorithm 2. It updates its value  $y_i$  to the midpoint of the current value of  $y_i$  and the value in  $\text{Rcv}_i$  that is the farthest from it. While having the benefit of only having to compare  $O(n)$  distances, and hence doing  $O(nd)$  local scalar operations for each agent in each round in the case of  $V = \mathbb{R}^d$  with component-wise representation, the ApproachExtreme algorithm also only has to measure distances from its current value to other agents' values; never the distance of two other agents' values. This can be helpful for agents embedded into the vector space  $V$  that can measure the distance from itself to another agent, but not necessarily the distance between two other agents.

The ApproachExtreme algorithm admits an upper bound of  $\sqrt{31/32}$  on its round-by-round convergence rate, which is worse than the  $\sqrt{7/8}$  of the MidExtremes algorithm. For the case of the real line  $V = \mathbb{R}$ , we can show a round-by-round convergence rate of  $3/4$ , however.

► **Theorem 2.** *In any non-split network model with values from any inner product space, the ApproachExtreme algorithm guarantees a round-by-round convergence rate of  $c(t) \leq \sqrt{\frac{31}{32}}$  for all rounds  $t \geq 1$ . Its convergence time is at most  $T(\varepsilon) = \left\lceil \log_{\sqrt{32/31}} \frac{\Delta}{\varepsilon} \right\rceil$  where  $\Delta$  is the diameter of the set of initial values.*

*In the particular case of values from the real line, it guarantees a round-by-round convergence rate of  $c(t) \leq 3/4$  and a convergence time of  $T(\varepsilon) = \left\lceil \log_{4/3} \frac{\Delta}{\varepsilon} \right\rceil$ .*

### 3.2 Asynchronous Byzantine Message Passing

We now show how to adapt algorithm MidExtremes to the case of asynchronous message passing systems with at most  $f$  Byzantine agents. The algorithm proceeds in the same asynchronous round structure and safe area calculation used by Mendes et al. [18] whenever approximate consensus is solvable, i.e., when  $n > (d + 2)f$ . Plugging in the MidExtremes

algorithm, we achieve a round-by-round convergence rate and round complexity independent of the dimension  $d$ .

More specifically, our algorithm has a round complexity of  $O(\log \frac{\Delta}{\varepsilon})$ , which leads to a message complexity of  $O(n^2 \log \frac{\Delta}{\varepsilon})$  where  $\Delta$  is the maximum Euclidean distance of initial vectors of correct agents. In contrast, the Mendes–Herlihy algorithm has a worst-case round complexity of  $\Omega(d \log \frac{d\Delta}{\varepsilon})$  and a worst-case message complexity of  $\Omega(n^2 d \log \frac{d\Delta}{\varepsilon})$ . We are thus able to get rid of all terms depending on the dimension  $d$ .

After an initial round estimating the initial diameter of the system, the Mendes–Herlihy algorithm has each agent  $i$  repeat the following steps in each coordinate  $k \in \{1, 2, \dots, d\}$  for  $\Theta(\log \frac{d\Delta}{\varepsilon})$  rounds:

1. Collect a multiset  $V_i$  of agents' vectors such that every intersection  $V_i \cap V_j$  has at least  $n - f$  elements via reliable broadcast and the witness technique [1].
2. Calculate the safe area  $S_i$  as the intersection of the convex hulls of all sub-multisets of  $V_i$  of size  $|V_i| - f$ . The safe area is guaranteed to be a subset of the convex hull of vectors of correct agents. Helly's theorem [12] can be used to show that every intersection  $S_i \cap S_j$  of safe areas is nonempty.
3. Update the vector  $y_i$  to be in the safe area  $S_i$  and have its  $k^{\text{th}}$  coordinate equal to the midpoint of the set of  $k^{\text{th}}$  coordinates in  $S_i$ .

The fact that safe areas have nonempty pairwise intersections guarantees that the diameter in the  $k^{\text{th}}$  coordinate

$$\delta_k(t) = \max_{i,j \text{ correct}} |y_i^{(k)}(t) - y_j^{(k)}(t)|$$

at the end of round  $t$  fulfills  $\delta_k(t) \leq \delta_k(t-1)/2$  if round  $t$  considers coordinate  $k$ . The choice of the number of rounds for each coordinate guarantees that we have  $\delta_k(t) \leq \varepsilon/\sqrt{d}$  after the last round for coordinate  $k$ . This in turn makes sure that the Euclidean diameter of the set of vectors of correct agents after all of the  $\Theta(d \log \frac{d\Delta}{\varepsilon})$  rounds is at most  $\varepsilon$ .

The article of Mendes et al. [18] describes a second algorithm, the Vaidya–Garg algorithm, which replaces steps 2 and 3 by updating  $y_i$  to the non-weighted average of arbitrarily chosen points in the safe areas of all sub-multisets of  $V_i$  of size  $n - f$ . Another difference to the Mendes–Herlihy algorithm is that it repeats the steps not several times for every dimension, but for  $\Theta(n^{f+1} \log \frac{d\Delta}{\varepsilon})$  rounds in total. The Vaidya–Garg algorithm comes with the advantage of not having to do the calculations to find a midpoint for the  $k^{\text{th}}$  coordinate while remaining inside the safe area, but also comes with the cost of a convergence rate and a round complexity that depends on the number of agents.

The algorithm we propose has the same structure as the Mendes–Herlihy algorithm, with the following differences: (i) like the Vaidya–Garg algorithm it is missing the loop over all coordinates one-by-one, and (ii) we replace step 3 by updating vector  $y_i$  to the midpoint of two points that realize the Euclidean diameter of the safe area  $S_i$ . According to our results in Section 4.1, the Euclidean diameter

$$\delta(t) = \max_{i,j \text{ correct}} \|y_i(t) - y_j(t)\|$$

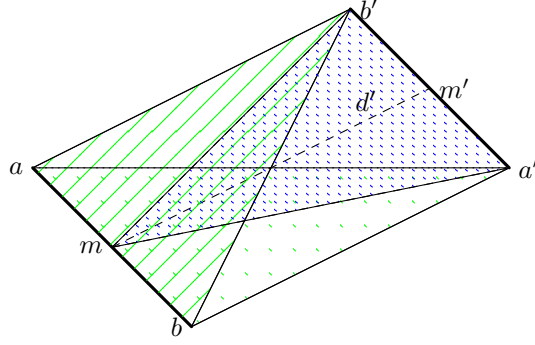
of the set of vectors of correct agents at the end of round  $t$  satisfies

$$\delta(t) \leq \sqrt{\frac{7}{8}} \delta(t-1) .$$

This means that we have  $\delta(T) \leq \varepsilon$  after

$$T(\varepsilon) = \left\lceil \log_{\sqrt{8/7}} \frac{\Delta}{\varepsilon} \right\rceil$$

rounds.



■ **Figure 1** Tetrahedron formed by extreme points  $a$  and  $b$  of agent  $i$  and extreme points  $a'$  and  $b'$  of agent  $j$ . The distance between the new agent positions  $m$  and  $m'$  is  $d'$ .

## 4 Performance Bounds

We next show upper bounds on the round-by-round convergence rate for algorithms MidExtremes (Theorem 1) and ApproachExtreme (Theorem 2) in non-split network models.

### 4.1 Bounds for MidExtremes

For dimension 1, MidExtremes is equivalent to the MidPoint Algorithm. We hence already know that  $c(t) \leq \frac{1}{2}$  from [7], proving the case of the real line in Theorem 1.

For the case of higher dimensions we will show that  $c(t) \leq \sqrt{\frac{7}{8}}$  holds. The proof idea is as follows: For a round  $t \geq 1$ , we consider two agents  $i, j$  whose distance realizes  $\text{diam}(y(t))$ . By the algorithm we know that both agents set their  $y_i(t)$  and  $y_j(t)$  according to  $y_i(t) = m = (a + b)/2$  and  $y_j(t) = m' = (a' + b')/2$ , where  $a, b$  are the extreme points received by agents  $i$  in round  $t$  and  $a', b'$  are the extreme points received by agents  $j$  in the same round. All four points must lie within a common subspace of dimension 3, and form the vertices of a tetrahedron as depicted in Figure 1.

Further, any three points among  $a, b, a', b'$  must lie within a 2 dimensional subspace, forming a triangle. Lemma 3 states the distance from the midpoint of two of its vertices to the opposite vertex, say  $c$ , and an upper bound in case the two edges incident to  $c$  are upper bounded in length.

► **Lemma 3.** *Let  $\gamma \geq 0$  and  $a, b, c \in V$ . Setting  $m = (a + b)/2$ , we have*

$$\|m - c\|^2 = \frac{1}{2}\|a - c\|^2 + \frac{1}{2}\|b - c\|^2 - \frac{1}{4}\|a - b\|^2 .$$

*In particular, if  $\|a - c\| \leq \gamma$  and  $\|b - c\| \leq \gamma$ , then*

$$\|m - c\|^2 \leq \gamma^2 - \frac{1}{4}\|a - b\|^2 .$$

**Proof.** We begin by calculating

$$\|a - c\|^2 = \|(a - m) + (m - c)\|^2 = \|a - m\|^2 + \|m - c\|^2 + 2\langle a - m, m - c \rangle \quad (1)$$

and

$$\|b - c\|^2 = \|(b - m) + (m - c)\|^2 = \|b - m\|^2 + \|m - c\|^2 + 2\langle b - m, m - c \rangle . \quad (2)$$

Adding (1) and (2), while noting  $\|a - m\|^2 = \|b - m\|^2 = \frac{1}{4}\|a - b\|^2$  and  $a - m = (a - b)/2 = -(b - m)$ , gives

$$\|a - c\|^2 + \|b - c\|^2 = \frac{1}{2}\|a - b\|^2 + 2\|m - c\|^2 .$$

Rearranging the terms in the last equation concludes the proof. ◀

We are now in the position to prove Lemma 4 that is central for Theorem 1. The lemma provides an upper bound on the distance  $d'$  between  $m$  and  $m'$  for the tetrahedron in Figure 1 given that all its sides are upper bounded by some  $\gamma \geq 0$  and the sum of the lengths of edge  $a, b$  and  $a', b'$ , i.e.,  $\|a - b\| + \|a' - b'\|$ , is lower bounded by  $\gamma$ . At the heart of the proof of Lemma 4 is an application of Lemma 3 for the three hatched triangles in Figure 1.

► **Lemma 4.** *Let  $a, b, a', b' \in V$  and  $\gamma \geq 0$  such that*

$$\text{diam}(\{a, b, a', b'\}) \leq \gamma \leq \|a - b\| + \|a' - b'\| . \quad (3)$$

*Then, setting  $m = (a + b)/2$  and  $m' = (a' + b')/2$ , we have*

$$\|m - m'\| \leq \sqrt{\frac{7}{8}}\gamma .$$

**Proof.** Applying Lemma 3 with the points  $a, b, a'$  yields

$$\|m - a'\|^2 \leq \gamma^2 - \frac{1}{4}\|a - b\|^2 . \quad (4)$$

Another invocation with the points  $a, b, b'$  gives

$$\|m - b'\|^2 \leq \gamma^2 - \frac{1}{4}\|a - b\|^2 . \quad (5)$$

Now, again using Lemma 3 with the points  $a', b', m$  and the bounds of (4) and (5), we get

$$\|m - m'\|^2 \leq \gamma^2 - \frac{1}{4}(\|a - b\|^2 + \|a' - b'\|^2) .$$

Using the second inequality in (3) then shows

$$\|m - m'\|^2 \leq \gamma^2 - \frac{1}{4}(\|a - b\|^2 + (\gamma - \|a - b\|)^2) . \quad (6)$$

Setting  $\xi = \|a - b\|$ , we get

$$\|m - m'\|^2 \leq \max_{0 \leq \xi \leq \gamma} \gamma^2 - \frac{1}{4}(\xi^2 + (\gamma - \xi)^2) .$$

Differentiating the function  $f(\xi) = \gamma^2 - \frac{1}{4}(\xi^2 + (\gamma - \xi)^2)$  reveals that its maximum is attained for  $-(2\xi - \gamma) = 0$ , i.e.,  $\xi = \gamma/2$ , which gives

$$\|m - m'\|^2 \leq \gamma^2 - \frac{\gamma^2}{8} = \frac{7}{8}\gamma^2 .$$

Taking the square root now concludes the proof. ◀

We can now prove Theorem 1. For the proof we consider the tetrahedron with vertices  $a, b, a', b'$  as discussed before; see Figure 1. Recalling that the vertices  $a, b$  are vectors received by an agent  $i$  and  $a', b'$  vectors received by an agent  $j$  in the same round, we may infer from the non-split property that all communication graphs must fulfill that both  $i$  and  $j$  must have received a common vector from an agent. Together with the algorithm's rule of picking  $a, b$  and  $a', b'$  as extreme points, we obtain the constraints required by Lemma 4. Invoking this lemma we finally obtain an upper bound on the distance  $d'$  between  $m$  and  $m'$ , and by this an upper bound on the round-by-round convergence rate of the MidExtremes algorithm.

**Proof of Theorem 1.** Let  $i$  and  $j$  be two agents. Let  $a, b \in \text{Rcv}_i(t)$  such that  $y_i(t) = (a+b)/2$  and  $a', b' \in \text{Rcv}_j(t)$  such that  $y_j(t) = (a' + b')/2$ . Define  $\gamma_{ij} = \text{diam}(\{a, b, a', b'\})$ . Since  $a, b, a', b'$  are the vectors of some agents in round  $t - 1$ , we have  $\gamma_{ij} \leq \text{diam}(y(t - 1))$ .

Further, from the non-split property, there is an agent  $k$  whose vector  $c = y_k(t - 1)$  has been received by both  $i$  and  $j$ , i.e.,  $c \in \text{Rcv}_i(t) \cap \text{Rcv}_j(t)$ . By the choice of the extreme points  $a, b$  by agent  $i$ , we must have  $\|a - c\| \leq \|a - b\|$ ; otherwise  $a, b$  would not realize the diameter of  $\text{Rcv}_i(t)$ . Analogously, by the choice of the extreme points  $a', b'$  by agent  $j$ , it must hold that  $\|a' - c\| \leq \|a' - b'\|$ .

From the triangular inequality, we then obtain

$$\|a - a'\| \leq \|a - c\| + \|c - a'\| \leq \|a - b\| + \|a' - b'\| .$$

Analogous arguments for the other pairs of points in  $\{a, b, a', b'\}$  yield

$$\text{diam}(\{a, b, a', b'\}) = \gamma_{ij} \leq \|a - b\| + \|a' - b'\| .$$

We can hence apply Lemma 4 to obtain

$$\|y_i(t) - y_j(t)\| \leq \sqrt{\frac{7}{8}} \gamma_{ij} \leq \sqrt{\frac{7}{8}} \text{diam}(y(t - 1)) .$$

Taking the maximum over all pairs of agents  $i$  and  $j$  now shows

$$\text{diam}(y(t)) \leq \sqrt{7/8} \cdot \text{diam}(y(t - 1)) ,$$

which concludes the proof. ◀

## 4.2 Bounds for ApproachExtreme

We start by showing the one-dimensional case of Theorem 2, i.e.,  $V = \mathbb{R}$ , in Section 4.2.1. Section 4.2.2 then covers the multidimensional case.

### 4.2.1 One-dimensional Case

For the proof we use the notion of  $\varrho$ -safety as introduced by Charron-Bost et al. [7]. A convex combination algorithm is  $\varrho$ -safe if

$$\varrho M_i(t) + (1 - \varrho)m_i(t) \leq y_i(t) \leq (1 - \varrho)M_i(t) + \varrho m_i(t) \tag{7}$$

where  $M_i(t) = \max(\text{Rcv}_i(t))$  and  $m_i(t) = \min(\text{Rcv}_i(t))$ .

It was shown [7, Theorem 4] that any  $\varrho$ -safe convex combination algorithm guarantees a round-by-round convergence rate of  $c(t) \leq 1 - \varrho$  in any non-split network model. In the sequel, we will show that ApproachExtreme is  $\frac{1}{4}$ -safe when applied in  $V = \mathbb{R}$ .



**Proof of Theorem 2, one-dimensional case.** Let  $i$  be an agent and  $t \geq 1$  a round in some execution of ApproachExtreme in  $V = \mathbb{R}$ . We distinguish the two cases  $y_i(t) \leq y_i(t-1)$  and  $y_i(t) > y_i(t-1)$ .

In the first case, we have  $b \leq y_i(t-1)$  for the vector  $b$  that agent  $i$  calculates in code line 4 in round  $t$ . But then necessarily  $b = y_i(t)$  since this is the most distant point to  $y_i(t-1)$  in  $\text{Rcv}_i(t)$  to the left of  $y_i(t-1)$ . Also,  $y_i(t-1) \geq (M_i(t) + m_i(t))/2$  since otherwise  $M_i(t)$  would be farther from  $y_i(t-1)$  than  $m_i(t)$ . But this means that

$$y_i(t) = \frac{y_i(t-1) + m_i(t)}{2} \geq \frac{1}{4}M_i(t) + \frac{1}{4}m_i(t) + \frac{1}{2}m_i(t) = \frac{1}{4}M_i(t) + \frac{3}{4}m_i(t) ,$$

which shows the first inequality of  $\rho$ -safety (7) with  $\rho = \frac{1}{4}$ . The second inequality of (7) follows from  $y_i(t-1) \leq M_i(t)$  since

$$y_i(t) = \frac{y_i(t-1) + m_i(t)}{2} \leq \frac{1}{2}M_i(t) + \frac{1}{2}m_i(t) \leq \frac{3}{4}M_i(t) + \frac{1}{4}m_i(t) .$$

In the second case, (7) is proved analogously to the first case. ◀

### 4.2.2 Multidimensional Case

For the proof of Theorem 2 with higher dimensional values, we consider two agents  $i, j$  whose distance realizes  $\text{diam}(y(t))$ . From the ApproachExtreme  $y_i(t) = m = (a + y_i(t-1))/2$  and  $y_j(t) = m' = (a' + y_j(t-1))/2$  where  $a$  and  $a'$  maximize the distance to  $y_i(t-1)$  and  $y_j(t-1)$ , respectively, among the received values.

To show an upper bound on the distance  $d'$  between the new agent positions  $m$  and  $m'$  in the multidimensional case, we need the following variant of Lemma 4 in which we relax the upper bound on  $\gamma$  by a factor of two, but thereby weaken the bound on  $d'$ .

Analogous to the proof of Theorem 1, the proof is by applying Lemma 5 to the three hatched triangles in Figure 1.

► **Lemma 5.** *Let  $a, b, a', b' \in V$  and  $\gamma \geq 0$  such that*

$$\text{diam}(\{a, b, a', b'\}) \leq \gamma \leq 2\|a - b\| + 2\|a' - b'\| .$$

*Then, setting  $m = (a + b)/2$  and  $m' = (a' + b')/2$ , we have*

$$\|m - m'\| \leq \sqrt{\frac{31}{32}}\gamma .$$

**Proof.** The proof of the lemma is essentially the same as that of Lemma 4, with the following differences: Equation (6) is replaced by

$$\|m - m'\|^2 \leq \gamma^2 - \frac{1}{4} \left( \|a - b\|^2 + \left( \frac{\gamma}{2} - \|a - b\| \right)^2 \right) ,$$

which changes the function  $f$  to  $f(\xi) = \gamma^2 - \frac{1}{4}(\xi^2 + (\frac{\gamma}{2} - \xi)^2)$ . The maximum of this function  $f$  is achieved for  $\xi = \gamma/4$ , which means that

$$\|m - m'\|^2 \leq f(\gamma/4) = \gamma^2 - \frac{\gamma^2}{32} = \frac{31}{32}\gamma^2 . \quad \blacktriangleleft$$

We are now in the position to prove Theorem 2.

**Proof of Theorem 2, multidimensional case.** Let  $i$  and  $j$  be two agents. Let  $a = y_i(t-1)$  and  $a' = y_j(t-1)$ . Further, let  $b \in \text{Rcv}_i(t)$  such that  $y_i(t) = (a+b)/2$  and  $b' \in \text{Rcv}_j(t)$  such that  $y_j(t) = (a'+b')/2$ . Define  $\gamma_{ij} = \text{diam}(\{a, b, a', b'\})$ . Since  $a, b, a', b'$  are the vectors of some agents in round  $t-1$ , we have  $\gamma_{ij} \leq \text{diam}(y(t-1))$ .

From the non-split property, there is an agent  $k$  whose vector  $c = y_k(t-1)$  has been received by both  $i$  and  $j$ , i.e.,  $c \in \text{Rcv}_i(t) \cap \text{Rcv}_j(t)$ . By the choice of the extreme point  $b$  by agent  $i$ , we must have  $\|a-c\| \leq \|a-b\|$ ; otherwise  $b$  would not maximize the distance to  $a$ . Analogously, by the choice of the extreme points  $b'$  by agent  $j$ , it must hold that  $\|a'-c\| \leq \|a'-b'\|$ . Note, however, that the roles of  $a$  and  $b$  are not symmetric and that, contrary to the proof of Theorem 1, we can have  $\|b-c\| > \|a-b\|$  or  $\|b'-c\| > \|a'-b'\|$ .

From the triangular inequality and the two established inequalities, we then obtain

$$\begin{aligned} \|a-a'\| &\leq \|a-c\| + \|a'-c\| \leq \|a-b\| + \|a'-b'\|, \\ \|a-b'\| &\leq \|a-c\| + \|c-a'\| + \|a'-b'\| \leq \|a-b\| + 2\|a'-b'\|, \end{aligned}$$

and

$$\|b-b'\| \leq \|b-a\| + \|a-c\| + \|c-a'\| + \|a'-b'\| \leq 2\|a-b\| + 2\|a'-b'\|.$$

Analogously,  $\|a'-b\| \leq 2\|a-b\| + \|a'-b'\|$ . Together this implies

$$\text{diam}(\{a, b, a', b'\}) = \gamma_{ij} \leq 2\|a-b\| + 2\|a'-b'\|.$$

We can hence apply Lemma 5 to obtain

$$\|y_i(t) - y_j(t)\| \leq \sqrt{\frac{31}{32}} \gamma_{ij} \leq \sqrt{\frac{31}{32}} \text{diam}(y(t-1)).$$

Taking the maximum over all pairs of agents  $i$  and  $j$  now shows  $\text{diam}(y(t)) \leq \sqrt{31/32} \cdot \text{diam}(y(t-1))$ , which concludes the proof.  $\blacktriangleleft$

## 5 Conclusion

We presented two new algorithms for asymptotic and approximate consensus with values in arbitrary inner product spaces. This includes not only the Euclidean spaces  $\mathbb{R}^d$ , but also spaces of infinite dimension. Our algorithms are the first to have constant contraction rates, independent of the dimension and the number of agents.

We have presented our algorithms in the framework of non-split network models and have then shown how to apply them in several other distributed computing models. In particular, we improved the round complexity of the algorithms by Mendes et al. [18] for asynchronous message passing with Byzantine faults from  $\Omega(d \log \frac{d\Delta}{\epsilon})$  to  $O(\log \frac{\Delta}{\epsilon})$ , eliminating all terms that depend on the dimension  $d$ .

The exact value of the optimal contraction rate for asymptotic and approximate consensus is known to be  $1/2$  in dimensions one and two [15, 8], but the question is still open for higher dimensions. Our results are a step towards the solution of the problem as they show the optimum in all dimensions to lie between  $1/2$  and  $\sqrt{7/8} \approx 0.9354 \dots$

---

## References

- 1 Ittai Abraham, Yonatan Amit, and Danny Dolev. Optimal resilience asynchronous approximate agreement. In Teruo Higashino, editor, *8th International Conference on Principles of Distributed Systems (OPODIS 2004)*, volume 3544 of *Lecture Notes in Computer Science*, pages 229–239. Springer, Heidelberg, 2005.

- 2 David Angeli and Pierre-Alexandre Bliman. Stability of leaderless discrete-time multi-agent systems. *MCSS*, 18(4):293–322, 2006.
- 3 Zohir Bouzid, Maria Gradinariu Potop-Butucaru, and Sébastien Tixeuil. Optimal Byzantine-resilient convergence in uni-dimensional robot networks. *Theoretical Computer Science*, 411(34-36):3154–3168, 2010.
- 4 Stephen Boyd and Lieven Vandenbergh. *Convex optimization*. Cambridge University Press, 2004.
- 5 Ming Cao, A. Stephen Morse, and Brian D. O. Anderson. Reaching a consensus in a dynamically changing environment: convergence rates, measurement delays, and asynchronous events. *SIAM J. Control Optim.*, 47(2):601–623, 2008.
- 6 Bernadette Charron-Bost, Matthias Függer, and Thomas Nowak. Approximate consensus in highly dynamic networks: The role of averaging algorithms. In *Proceedings of the 42nd International Colloquium on Automata, Languages, and Programming, ICALP15*, pages 528–539, 2015.
- 7 Bernadette Charron-Bost, Matthias Függer, and Thomas Nowak. Fast, robust, quantizable approximate consensus. In *Proceedings of the 43rd International Colloquium on Automata, Languages, and Programming, ICALP16*, pages 137:1–137:14, 2016.
- 8 Bernadette Charron-Bost, Matthias Függer, and Thomas Nowak. Multidimensional asymptotic consensus in dynamic networks. *CoRR*, abs/1611.02496, 2016. URL: <http://arxiv.org/abs/1611.02496>.
- 9 Bernadette Charron-Bost and André Schiper. The Heard-Of model: computing in distributed systems with benign faults. *Distrib. Comput.*, 22(1):49–71, 2009.
- 10 Bernard Chazelle. The total  $s$ -energy of a multiagent system. *SIAM Journal on Control and Optimization*, 49(4):1680–1706, 2011.
- 11 Mark Cieliebak, Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro. Solving the robots gathering problem. In *International Colloquium on Automata, Languages, and Programming*, pages 1181–1196. Springer, 2003.
- 12 Ludwig Danzer, Branko Grünbaum, and Victor Klee. Helly’s theorem and its relatives. In Victor Klee, editor, *Convexity*, volume 7 of *Proceedings of Symposia in Pure Mathematics*, pages 101–180. AMS, Providence, 1963.
- 13 Danny Dolev, Nancy A. Lynch, Shlomit S. Pinter, Eugene W. Stark, and William E. Weihl. Reaching approximate agreement in the presence of faults. *jacm*, 33(2):499–516, 1986.
- 14 Alan D. Fekete. Asymptotically optimal algorithms for approximate agreement. *Distrib. Comput.*, 4(1):9–29, 1990.
- 15 Matthias Függer, Thomas Nowak, and Manfred Schwarz. Tight bounds for asymptotic and approximate consensus. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing, PODC ’18*, pages 325–334, 2018.
- 16 Qun Li and Daniela Rus. Global clock synchronization in sensor networks. *IEEE Transactions on Computers*, 55(2):214–226, 2006.
- 17 Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, San Francisco, CA, 1996.
- 18 Hammurabi Mendes, Maurice Herlihy, Nitin Vaidya, and Vijay K. Garg. Multidimensional agreement in Byzantine systems. *Distributed Computing*, 28:423–441, 2015.
- 19 Luc Moreau. Stability of multiagent systems with time-dependent communication links. *IEEE Transactions on Automatic Control*, 50(2):169–182, 2005.
- 20 Reza Olfati-Saber and Richard M Murray. Consensus problems in networks of agents with switching topology and time-delays. *IEEE Transactions on automatic control*, 49(9):1520–1533, 2004.
- 21 Luis A. Rademacher. Approximating the centroid is hard. In *Proceedings of the Twenty-third Annual Symposium on Computational Geometry*, pages 302–305. ACM, 2007.

## 27:16 Fast Multidimensional Asymptotic and Approximate Consensus

- 22 Nicola Santoro and Peter Widmayer. Time is not a healer. In B. Monien and R. Cori, editors, *6th Symposium on Theoretical Aspects of Computer Science*, volume 349 of *LNCS*, pages 304–313. Springer, Heidelberg, 1989.
- 23 Fred B Schneider. Understanding protocols for Byzantine clock synchronization. Technical report, Cornell University, 1987.
- 24 Jennifer Lundelius Welch and Nancy Lynch. A new fault-tolerant algorithm for clock synchronization. *Information and computation*, 77(1):1–36, 1988.

# Local Queuing Under Contention

**Paweł Garncarek**

Institute of Computer Science, University of Wrocław, Poland  
pgarn@cs.uni.wroc.pl

**Tomasz Jurdziński**

Institute of Computer Science, University of Wrocław, Poland  
tju@cs.uni.wroc.pl

**Dariusz R. Kowalski**

Computer Science Department, University of Liverpool, Liverpool, UK  
D.Kowalski@liverpool.ac.uk

---

## Abstract

We study stability of local packet scheduling policies in a distributed system of  $n$  nodes. The local policies at nodes may only access their local queues, and have no other feedback from the underlying distributed system. The packets arrive at queues according to arrival patterns controlled by an adversary restricted only by injection rate  $\rho$  and burstiness  $b$ . In this work, we assume that the underlying distributed system is a shared channel, in which in order to get rid of a packet from the queue, a node needs to schedule it for transmission on the channel and no other packet is scheduled for transmission at the same time. We show that there is a local *adaptive* scheduling policy with relatively small memory, which is universally stable on a shared channel, that is, it has bounded queues for any  $\rho < 1$  and  $b \geq 0$ . On the other hand, without memory the maximal stable injection rate is  $O(1/\log n)$ . We show a local memoryless (*non-adaptive*) scheduling policy based on novel idea of ultra strong selectors which is stable for slightly smaller injection  $c/\log^2 n$ , for some constant  $c > 0$ .

**2012 ACM Subject Classification** Theory of computation → Distributed algorithms

**Keywords and phrases** Distributed algorithms, local queuing, shared channel, multiple-access channel, adversarial packet arrivals, stability, deterministic algorithms

**Digital Object Identifier** 10.4230/LIPIcs.DISC.2018.28

**Funding** Supported by Polish National Science Center grant 2017/25/B/ST6/02010.

## 1 Introduction

Queuing processes have been in the heart of computing and communication for decades. Queues are governed by scheduling algorithms, and the desired property is stability – meaning that there is an upper bound on the numbers of packets queued at devices at any time. Recently, due to rapidly growing number of devices and popularity of distributed protocols, the impact of congestion on stability of queuing processes has become a vibrant research topic. In this work, we study stability of local packet scheduling policies in the process of dynamic broadcasting on a shared channel. A shared channel, also called a multiple access channel, is a broadcast network with instantaneous delivery of transmitted messages to every device (also called a node or a station) in the system and a possibility of conflict for access to the transmitting medium. A message sent via a channel by a station is received successfully by all the stations when its transmission has not overlapped with transmissions by other stations.



© Paweł Garncarek, Tomasz Jurdziński, and Dariusz R. Kowalski;  
licensed under Creative Commons License CC-BY

32nd International Symposium on Distributed Computing (DISC 2018).

Editors: Ulrich Schmid and Josef Widder; Article No. 28; pp. 28:1–28:18

Leibniz International Proceedings in Informatics



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

The traditional approach to modeling dynamic broadcasting and the corresponding queuing process on a shared channel has assumed continuous packet injection subject to stochastic constraints (typically, Poisson arrival rates). Recent papers, inspired by adversarial queuing theory for store-and-forward packet networks, studied stability of deterministic broadcasting on a shared channel in adversarial settings. An adversary is determined by two parameters: injection rate  $\rho$ , which is the average number of injected packets, and burstiness  $b$ , which is the maximum number of packets that may be injected in a round. To the best of our knowledge, all previous work assumed that scheduling policies receive some feedback from the channel that help them in achieving stability. Our approach is the first that considers local schedulers without any feedback from the channel.

## 1.1 Our results

This paper investigates stability and other properties of deterministic local packet schedulers which are used for distributed broadcasting on a shared channel. The stability is studied in adversarial setting, defined in terms of global injection rate  $\rho$  and burstiness  $b$ . We consider two classes of local schedulers: adaptive and non adaptive. The former allows stations to monitor and store some digest of the local queue history, especially its size, while the latter allows the policy only to check whether the current local queue is empty or not. We show that there is a local adaptive scheduling policy with relatively small memory, which is universally stable on a shared channel, that is, it has bounded queues for any  $\rho < 1$  and  $b \geq 0$ . On the other hand, we prove that without memory the maximal stable injection rate is  $O(1/\log n)$ , more precisely, than any local non-adaptive scheduler can be stable only for injection rates  $O(1/\log n)$ . We also show a local non-adaptive policy based on novel idea of ultra strong selectors, which is stable for slightly smaller injection rate  $c/\log^2 n$ , for some constant  $c > 0$ .

## 1.2 Previous and related work

There is a rich history of research on broadcasting dynamically generated packets on multiple access channels. Early work in this direction included developing and studying properties of protocols like Aloha [1] and binary exponential backoff [23]. Most prior research on this topic has concentrated on scenarios when packets were injected subject to statistical constraints. Stability has been the basic quality criterion to achieve, understood in the sense that the input and output rates were equal. See the paper by Gallager [12] for an overview of early research; recent work includes the papers of Goldberg et al. [14], Goldberg et al. [15], Håstad et al. [17], and Raghavan and Upfal [24].

Adversarial queuing was introduced by Borodin et al. [8] as a framework to study stability of routing protocols in (point-to-point connected) networks under worst-case traffic scenarios modeled by adversaries. Andrews et al. [5] defined a greedy protocol to be universally stable when it was stable in all networks for any injection rate  $\rho < 1$ .

Bender et al. [6] studied stability of randomized backoff on a shared channel in adversarial settings in the *queue-free model* in which each packet is handled independently as if by a dedicated station. Stability was defined to mean that output rate was as large as injection rate. Some aspects of dynamic selection on multiple access channels by deterministic protocols were considered by Kowalski [21]. Both above approaches differ from the one in this paper, as they did not consider individual local queues.

Chlebus et al. [10] were the first who studied adversarial queuing on a shared channel. They however, similarly to all the follow up work (cf., [4, 3, 9]), assumed that schedulers are embedded into the channel, in the sense that they can receive channel feedback or

even attach and read additional information bits. Two of their results, however, could be transferred to the model of local scheduler assumed in this paper. Mainly, they showed that no acknowledgment-based protocol can be stable for injection rates larger than  $\frac{3}{1+\lg n}$ , and that there is a stable acknowledgment-based protocol for rates at most  $\frac{1}{cn \lg^2 n}$ , for a sufficiently large  $c > 0$ . This class of acknowledgment-based protocols is however different from the classes considered in this work, in the sense that the non-adaptive protocol restarts each time a new packet is considered by a scheduler – as such, it is difficult to compare their results with the ones in our work, and the only note we could make here is the huge exponential gap between the lower and upper bounds on stability rates.

Broadcasting on a shared channel with static input (i.e., when some  $k$  stations hold packets at the beginning of an execution) was also widely studied. The goal is to transmit either at least one of them, which is the *selection problem*, or all of them in the *k-broadcast problem*, in both cases as quickly as possible. Kushilevitz and Mansour [22] proved the lower bound  $\Omega(\log n)$  for the selection problem on the channel with  $n$  stations if collision detection was not available. Willard [25] developed protocols solving the selection problem in the expected time  $\mathcal{O}(\log \log n)$  in the channel with collision detection. This demonstrates that there is an exponential gap between models with collision detection and without it, with respect to the selection problem. The *k-broadcast problem* was studied by Greenberg and Winograd [16], Komlós and Greenberg [20], and Kowalski [21]. A related leader election problem was studied by Jurdziński et al. [18] for channels without collision detection. The related problem of wake-up, where stations become activated at possibly different times and have to transmit or exchange information, was also considered in the literature. It was introduced by Gąsieniec et al. [13]. They showed that if stations had access to a global clock then wakeup could be completed in the expected time  $\mathcal{O}(\log n)$  by a randomized protocol; this was later strengthened by Jurdziński and Stachowiak [19] who showed that the assumption about the global clock could be omitted. Czyżowicz et al. [11] studied deterministic solutions for the mutual exclusion and consensus problems on multiple-access channels when the adversary wakes up stations in arbitrary rounds. They considered various model settings, determined by the availability of three independent features: collision detection, global clock and knowledge of the number  $n$  of stations. In particular, they showed that if none of the three features is available then the problems are not solvable, while even a single of these features makes these problems feasible, with complexity of solutions depending on which combination of features is available. Bieńkowski et al. [7] analyzed randomized protocols for mutual exclusion under the model settings of [11].

## 2 Model

There are  $n$  stations attached to a shared channel. The stations have distinct names in the interval  $[0, n - 1]$ . Each station  $v$  knows its name  $v$  and also the number of all the stations  $n$ , in the sense that these parameters can be used in code of protocols.

### 2.1 Shared channel

Shared channel, also called a multiple access channel, is a communication medium with some special properties. We assume that the channel operates synchronously. Every device connected to it, called a node or a station, has its clock and the clock cycles are all of exactly the same length and synchronized. An execution of a protocol is partitioned into rounds – it takes precisely one round to transmit a message. We assume that stations have access to a global clock, meaning that all the local clocks at stations give the same round numbers.

Every station occasionally receives packets to broadcast. Packets are stored in a local queue. Stations use local scheduling algorithms to decide whether to schedule a packet from the queue for transmission in the current round or not. When exactly one station schedules a packet for transmission in a round, then the message containing this packet is successfully delivered and the packet disappears from the queue. Unlike in the previous work, we assume that local schedulers do not receive any feedback from the channel – they could only make their decisions based on examining local queue. When at least two stations transmit simultaneously in a round then *conflict for access* or *collision* occurs in the round and none of the transmitted packet is successful (i.e., all remain in their local queues).

## 2.2 Adversaries injecting packets

Packets are injected into stations in a dynamic fashion in the course of an execution of a broadcasting protocol. Packet injection is modeled by an adversary. Adversaries are specified by constraints on the maximum rate of injection  $\rho$  and by the burstiness of traffic  $b$ . An adversary generates a number of packets in each round and for each packet assigns a station to inject the packet in this round. The number of packets an adversary can inject into stations in one round is called the *burstiness* of the adversary. The adversary of type  $(\rho, b)$  can inject at most  $\rho t + b$  packets to stations, in total, during any time interval of  $t$  rounds. This type of adversary is typically called *leaky bucket*.

## 2.3 Local schedulers

A broadcast algorithm is determined by the work of local scheduling algorithms at each station. A local scheduler is a deterministic and distributed algorithm, which has only access to the current local queue, global clock, and potentially some additional internal memory (different from the local queue), but *does not receive* any feedback from the channel. It decides whether to schedule a packet for transmission in the current round or not. A packet scheduled for transmission is removed from the queue automatically if the transmission was successful (i.e., no other packet was scheduled for transmission in the same round), and the scheduler is aware of it.

W.l.o.g. we assume that the queue at a station operates in the first-in-first-out (FIFO) fashion, as we are only interested in stability (i.e., bounded queue sizes).

We focus on two classes of local schedulers. First class, *adaptive schedulers*, allows the algorithm to access the current queue (in particular, knows its size) and can use additional local memory (different from the local queue) to store some digest of queue history. The second class, *non-adaptive schedulers*, requires that the scheduler knows only if the current queue is empty or not and does not use local memory to record *any* digest of the history of local computation.

## 2.4 Quality of service

We say that a local scheduler is *stable* for injection rate  $\rho$  if in any execution of the scheduler against a  $(\rho, b)$ -adversary, for any  $b$ , queues are bounded at all times. *Universal stability* means stability for any injection rate  $\rho < 1$ .



### 3 Universally stable algorithm with (small) memory

In this section we build a universally stable algorithm under the assumption that nodes can collect information about history of an execution in their local memory.

We construct the algorithm gradually: first, we describe the basic window-base brute-force idea for known injection rate  $0 < \rho < 1$  and burstiness  $b \geq 0$  and under assumption that a node has an additional feedback from the channel; second, we modify it to get rid of the above mentioned assumptions.

#### 3.1 Window-based algorithm under additional assumptions

As mentioned earlier, the first algorithm will be designed for known injection rate  $\rho$  and burstiness  $b$ . Additionally, in this part we only consider the beeping channel feedback model, where a node can distinguish between no transmitting nodes and at least one transmitting node on the channel; in other words, a node can use so called “beeps” (i.e., at least one transmission on the channel) to encode some control information to other nodes.

The algorithm partitions time into windows of length  $L$ . Each window consists of a *gossip stage* and a *transmission stage*. In the gossip stage, the nodes learn the queue sizes of other nodes. Then they use this knowledge to compute locally an optimal schedule – as we will argue later, it will be the same at each node. In the transmission phase, the nodes use the schedule they computed in the preceding gossip phase to transmit all packets that were injected in the previous window. Below we specify the two stages in details.

##### Gossip phase

Let  $f(L) = \log L$  denote the maximal number of bits that any node must exchange in a gossiping algorithm in order to share its queue size taken from the beginning of the window (i.e., the size just at the beginning of the window); if the queue size is greater than  $L$ , the node only announces it has  $L$  packets. A node may require  $f(L)$  packets just to transmit the information in the gossip phase. Therefore each node actively participates in communication within a window only if it has at least  $f(L)$  packets at the beginning of the window. Round robin algorithm is used for this purpose, repeated  $f(L)$  times – a node transmits a packet only if it actively participates in the current window *and* it is its turn on the list of stations used by the round robin algorithm *and* it has 1 on the  $i$ -th position of the binary representation of its queue size from the beginning of the window, where  $i$  is the current number of iterations of round robin. In order to perform this algorithm, memory size  $f(L)$  is required at each node.

##### Transmission phase

Each node has full knowledge about the queue states of all active nodes in the system (subject to restrictions that for queues larger than  $L$  the value  $L$  is known) at the beginning of the transmission phase. Note that memory size  $n \cdot f(L)$  is needed for it. Therefore all active nodes can compute the same transmission schedule to be used in the remainder of the window. The nodes use a brute force algorithm to find the shortest schedule that transmits all packets injected before the end of the previous window and not yet transmitted, and then follow this schedule. There are at most  $\rho L + b$  packets injected into the system in the previous window, therefore a window of length  $T(\rho, b) + G(n, L)$  is long enough, where  $T(\rho, b) = \rho L + b$  is the length of the transmission phase, while  $G(n, L)$  is the length of the gossiping phase (in the gossiping phase the propagated queue sizes are capped by  $L$ , hence  $\log L$  bits are enough in

---

**Algorithm 1** Local Scheduling with Memory for a station  $v$ .
 

---

```

1:  $L_v \leftarrow \text{min\_length}$ 
2: loop
3:    $\text{old}_v \leftarrow$  number of packets in the queue
4:    $q_v \leftarrow$  number of packets injected in the previous window
5:   if  $\text{old}_v > f(n, L_v)$  then
6:      $(\text{schedule}, \text{increase?}) \leftarrow \text{Gossip}(L_v)$ 
7:     if  $\text{increase?}$  then
8:        $L_v \leftarrow 2L_v$ 
9:       Idle until round  $t$  such that  $t$  is divisible by the new  $L_v$ 
10:    else
11:       $\text{Transmit}(\text{schedule})$ 

```

---

each such transmission). Therefore, if we use window length  $L \geq (b + G(n, L))/(1 - \rho)$ , there is enough time to transmit all packets from the previous window. Indeed, for such values of  $L$  we have the desired property  $L \geq T(\rho, b) + G(n, L)$ .

Since we iterate round robin algorithm  $f(L)$  times for gossiping, we get  $G(n, L) = n \log L$ . Therefore, there exists a sufficiently large window length  $L$  such that  $L \geq (b + n \log L)/(1 - \rho)$  and consequently the above algorithm transmits all packets injected in a window during the next window from all active nodes. This proves (recursively) that all packets in queues at the beginning of a window have been injected during the previous window (except for at most  $f(n, L)$  packets in each station, which could be injected earlier without activating the station), automatically implying stability.

### 3.2 Modified algorithm

The above brute-force window-based algorithm has a number of weaknesses:

1.  $\rho$  and  $b$  are known
2. users need to listen to the channel feedback (as opposed to only detecting own collisions)

Next we modify the above algorithm to get rid of these weaknesses. A high-level description of algorithm executed by a node is presented below, as Algorithm 1., with subprocedures Gossip (Procedure 2), Exchange (Procedure 3) and Transmit (described only in words).

Each node  $v$  knows its previous window size  $L_v$  (different nodes may have different window lengths). At the start of the algorithm, each node starts with a window of length  $\text{min\_length} = n(n - 1) \cdot 8$ . To minimize desynchronization that occurs, a node  $v$  only starts a window of length  $L_v$  if the current round  $t$  is divisible by  $L_v$ .

Only nodes that started their window with at least  $n(n - 1) \cdot \log L_v$  packets in their queues participate in the window – these are called *active nodes*. Each inactive node  $u$  idles until it gets enough packets to become active (and then it idles until such a round  $t$  that is divisible by  $L_u$ ).

In a window all active nodes learn each other's *fresh packets* – packets injected in the previous window – and decide on a common transmission schedule that transmits all fresh packets. If the computed schedule is too long to fit by the end of their window, each active node  $v$  doubles its window size (and idles until the next window starts). Otherwise they transmit according to this schedule. If an active node  $v$  learns that some other active node  $u$  uses a shorter window than  $L_v$ , then  $v$  informs  $u$  about this and  $u$  doubles its window length (and idles until the next window starts).

---

**Algorithm 2** Procedure *Gossip*( $L_v$ ).
 

---

```

1:  $ignore \leftarrow Exchange(\vec{1})$ 
2:  $ignore \leftarrow Exchange(\vec{1})$ 
3:  $die? \leftarrow Exchange(\vec{0})$ 
4: if  $die? \neq \vec{0}$  then
5:   Idle for  $3(\lceil \log W_v \rceil + 1)$  phases
6:   return ( $null, true$ )
7:  $q_1, q_2, \dots, q_{\lceil \log W_v \rceil} \leftarrow$  bits of  $\min(q_v, 2^{\lceil \log W_v \rceil})$ 
8: for  $i \leftarrow 1; i \leq \lceil \log W_v \rceil; ++ i$  do
9:   if  $q_i = 1$  then
10:     $\vec{x} \leftarrow Exchange(\vec{1})$ 
11:     $\vec{y} \leftarrow Exchange(\vec{0})$ 
12:   else
13:     $\vec{x} \leftarrow Exchange(\vec{0})$ 
14:     $\vec{y} \leftarrow Exchange(\vec{1})$ 
15:   for  $j \leftarrow 1; j \leq n; ++ j$  do
16:     if  $x_j = y_j$  then
17:        $z_j \leftarrow 1$ 
18:     else
19:        $z_j \leftarrow 0$ 
20:    $die? \leftarrow Exchange(\vec{z})$ 
21:   if  $die? \neq \vec{0}$  then
22:     Idle for  $3(\lceil \log W_v \rceil + 1 - j)$  phases
23:     return ( $null, true$ )
24:  $ignore \leftarrow Exchange(\vec{0})$ 
25:  $ignore \leftarrow Exchange(\vec{0})$ 
26:  $die? \leftarrow Exchange(\vec{0})$ 
27: if  $die? \neq \vec{0}$  then
28:   return ( $null, true$ )
29:  $schedule \leftarrow compute(exchange\_results)$ 
30: return ( $schedule, false$ )

```

---

**Exchange subprocedure.** Exchange subprocedure's goal is to transmit a single bit of information to and from each active node (it can transfer different bits to each node) and therefore implements *beeps* from the previous section.

It takes  $n(n - 1)$  rounds. This timeframe is divided into  $n$  intervals of  $n - 1$  rounds each. In the  $i$ -th interval a  $i$ -th node has a chance to transmit a bit to each of the  $n - 1$  other nodes.  $i$ -th node transmitting a packet in  $j$ -th round of  $i$ -th interval means that  $i$ -th node tries to transmit a binary signal 1 to the  $j$ -th node (to the  $j + 1$ -th node if  $j \geq i$ ). No transmission in  $j$ -th round of interval  $i$  from node  $i$  means that either  $i$ -th node transmits a binary 0 to the  $j$ -th node (or  $j + 1$ -th node) or it is inactive.

Furthermore, the active nodes need to be able to receive these bits of information. Each active node  $j$  transmits a packet in  $j$ -th round of every interval  $i$  for  $i < j$  and of every interval  $i + 1$  for  $i \geq j$ . If a collision occurred in  $j$ -th round of interval  $i$ , then either (for  $j < i$ )  $j$ -th node interprets this as  $i$ -th node transmitting a bit 1 or (for  $j \geq i$ )  $j$ -th node interprets this as  $i + 1$ -th node transmitting a bit 1.

---

**Algorithm 3** Procedure  $Exchange(\vec{x})$  for a node  $v$ .

---

```

1: for  $block \leftarrow 1; block \leq n; ++ block$  do
2:   if  $block = v$  then
3:     for  $j \leftarrow 1; j \leq n - 1; ++ j$  do
4:       if  $x_j = 1$  then
5:         Send a packet
6:       else
7:         Idle for 1 round
8:   else
9:     for  $j \leftarrow 1; j \leq n - 1; ++ j$  do
10:      if  $(v < block \text{ and } v = j)$  or  $(v > block \text{ and } v = j + 1)$  then
11:        Send a packet
12:      else
13:        Idle for 1 round

```

---

Note that the *exchange* procedure has no guarantees about successful transmissions. It is used solely for communication.

**Gossip stage.** Gossip stage is used to share queue sizes of all active nodes, so that they can agree on a common schedule. Furthermore it is used to find out whether some nodes should increase their window sizes.

Gossip stage is divided into  $\log L_v + 2$  phases. Each phase consists of 3 *exchange* routines. The first phase and the last phase are used to inform other gossiping nodes about  $v$ 's start and end of gossiping stage. The middle  $\log L_v$  phases are used to transmit  $v$ 's queue size, each phase for one bit.

The first phase of gossip of node  $v$  is always exchanging 110 to each other node. The last phase of gossip is always exchanging 000 to each other node. In the middle phases, a transmission of a bit 1 is encoded as exchanging 10? to each other node, while a transmission of a bit 0 is encoded as exchanging 01? to each other node, where the third exchange of these phases will be described later.

This kind of coding is used to ensure that nodes with different window lengths, and therefore desynchronized gossiping phases, find out about each other and learn which one has shorter window length. Indeed, if another node  $u$  with a shorter window length starts a gossip stage at the same time as  $v$ ,  $v$  learns that  $u$  has a shorter gossip (and therefore a shorter window  $L_u$ ) after 2 exchanges of the last  $u$ 's phase (00 are enough to identify the finishing phase). If  $u$  starts a gossip during  $v$ 's gossip stage, within 2 exchanges (11 at the start of  $u$ 's gossip)  $v$  learns about it.

For every node  $u$  that starts or finishes a gossip during  $v$ 's gossip,  $v$  exchanges *kill* to node  $u$  – a bit 1 on the third exchange of the phase when  $v$  found out that  $u$  is desynchronized (this was denoted earlier by a '?'). A node that hears a kill from any other node immediately stops its transmissions, doubles its window size and waits until the next start of a window.

If the nodes after a gossiping phase find out that they are not able to transmit all packets from the previous window (i.e., the computed schedule would be too long for the allocated length of transmitting stage), then they double their window lengths.

**Transmission stage.** Transmission stage is used to transmit all fresh packets, according to a schedule found during a gossip stage. Note that gossip stage may successfully transmit some fresh packets or packets injected in earlier windows. Therefore a node may have not enough fresh packets to use the schedule, but that means that in this window it successfully transmitted all fresh packets.

Note that the schedule guarantees no collisions between nodes that started their windows at the same time. However, a collision may occur with a node which started its window at a different time, so with a different window length. If a node  $v$  detects a collision during its transmission stage, it stops following its schedule, and instead transmits kill messages to all other nodes until it runs out of packets or the window ends. These kill messages are transmitted in all rounds during intervals  $I = [t + 2n(n - 1), t + 3n(n - 1))$ , for every  $t$  divisible by  $3n(n - 1)$  (these are the rounds reserved for kill messages in gossiping nodes).

These transmissions will cause collisions with other active nodes  $u$  in their transmission stage or in their gossip stage. So  $u$  will either start transmitting kill messages (in case it happened during the transmission stage), or it will interpret this collision as a kill message (if it happened during the gossip stage).

### 3.3 Analysis of the modified algorithm

► **Theorem 1.** *There exists an adaptive scheduler against any adversary with injection rate  $\rho < 1$  and burstiness  $b$ .*

We show that the algorithm described above is a scheduler satisfying requirements of the theorem. We need to prove 2 lemmas:

► **Lemma 2.** *For any  $\rho$  and  $b$  there exists a window length  $L_{max}$ , such that it is never exceeded.*

► **Lemma 3.** *In every window of length  $L_{max}$  either all nodes transmit all packets injected in the previous window of length  $L_{max}$  or some node decides to increase its window length.*

If both lemmas hold, then there is a finite number of windows where packets are processed slower than they are injected. Therefore the queues are bounded and the algorithm is universally stable. It remains to show that both above lemmas hold.

**Proof of Lemma 2.** For every  $n, \rho, b$  there exists a window length  $L$  such that if every node achieves it, all the nodes transmit all the packets injected in the previous window, without collisions during transmission stages (see the explanation in the base algorithm, Section 3). Observe that a node  $v$  increases its window length in two cases:

- Case 1: During *Gossip* stage it heard a *kill* message from another node  $u$ .
- Case 2: During *Gossip* phase it found out that it has too many packets to transmit in a window of current length.

Note that the 1st case can only occur if  $u$  has a longer window than  $v$ , therefore  $L_v < L$ , so  $2L_v \leq L$ .

The 2nd case allows a node with the longest window in the entire network to increase its window length. This however happens only if the current window length is insufficient and therefore shorter than  $L$ .

Therefore the lemma holds for  $L_{max} = L$ . ◀

**Proof of Lemma 3.** If a node is unable to transmit all fresh packets, then in the gossip phase it doubles its window length. If a node  $v$  was unable to successfully transmit all fresh packets due to collisions during  $v$ 's transmissions stage with a node  $u$  that has a different window length, one of the cases below occurred:

1.  $L_u < L_v$  and  $u$  started its window before (or in the same phase)  $v$  finished its gossip stage –  $u$  receives a kill from  $v$  at the start of  $u$ 's gossip.
2.  $L_u < L_v$  and  $u$  started its window after  $v$  finished its gossip stage –  $v$  detects a collision during its transmission stage and starts transmitting kill messages instead of following its schedule. Either  $u$  heard a kill message from  $v$  or  $v$  ran out of packets (transmitted all fresh packets).
3.  $L_u > L_v$  and  $u$  started its window at the same time that  $v$  –  $v$  receives a kill at the end of its gossip stage.
4.  $L_u > L_v$  and  $u$  started its window before  $v$  and  $u$  finished its gossip stage after (or in the same phase)  $v$  started its gossip stage –  $v$  receives a kill at the start of its gossip
5.  $L_u > L_v$  and  $u$  started its window before  $v$  and  $u$  finished its gossip stage before  $v$  started its gossip stage –  $u$  detects a collision during its transmission stage and starts transmitting kill messages instead of following its schedule. Either  $v$  heard a kill message from  $u$  or  $u$  ran out of packets (transmitted all fresh packets).

Therefore any active node  $v$  during its window either:

- transmits all its fresh packets,
- finds out its window length is too short to transmit its packets,
- receives a kill message,
- transmits a kill message that is received by another node.

So either all active nodes transmit all their fresh packets in their every window during interval of length  $L_{max}$  or there exists a node that increased its window length. ◀

**Memory usage.** Note that a node remembers queue sizes of all other nodes, where queue sizes are bounded by  $L_{max}$  (a value dependent on  $b$ ,  $\rho$  and  $n$ ). So each node uses at most  $n \cdot \log L_{max}$  bits of memory.

## 4 Unknown queue size

In this section we consider a class of protocols where stations do not know how many packets they have themselves. Each station decides whether to send a packet or not based only on the current round number. If such station has no packets to send, it simply fails.

### 4.1 Impossibility result

In this subsection we will show that no oblivious algorithm can be stable against an adversary with injection rate  $\rho = \omega(1/\log n)$ . Firstly, we introduce some auxiliary terminology and provide a high level description of the proof of the result. Then the formal proof follows.

#### 4.1.1 Auxiliary terminology and high level description of the impossibility result

For a given set  $X$ , let  $2^X$  denote the set of all subsets of  $X$ .

► **Definition 4.** A *scenario* is a class of packet arrival patterns where only a specific subset of stations have packets injected into them.

According to the above definition, each scenario can be identified with a subset of  $[n]$  corresponding to nodes to which packets can be injected. And therefore, there are  $2^n$  possible scenarios.

Consider a fixed oblivious algorithm  $A$  for a system which consists of  $n > 0$  nodes. Then, for a fixed set of scenarios  $S \subseteq 2^{[n]}$  (i.e., subsets of  $U = [n]$ ), let us build a bipartite graph  $G(U \cup S, E)$ , where  $(u, s) \in E$  iff  $u \in s$ . That is,  $(u, s)$  denotes the fact that an adversary can inject packets to the node  $u$  in the scenario  $s$ . [Observe that each bipartite graphs with partition of nodes  $U \cup S$  and  $U = [n]$  uniquely describes a set of scenarios and each set of scenarios  $S$  uniquely determines the graph  $G(U \cup S, E)$ .]

We say that an execution of a system is *saturated* with respect to a scenario  $s \subseteq [n]$  in some period of time when each queue  $i \in s$  is nonempty at each round of the given period.

For a given schedule  $A$ , let us restrict to its saturated executions on some set of scenarios. Then, one can characterize successful rounds of schedule  $A$  (i.e., such that a packet is transmitted) in saturated executions for various scenarios, using the following terminology.

We say that a set  $H \subseteq [n]$  *hits* a scenario  $s \subseteq [n]$  iff  $|H \cap s| = 1$ . Now, for a given round  $t$ , let  $A_t \subseteq [n]$  be the set of nodes which transmit a message in round  $t$  according to the schedule  $A$  (provided their queues are not empty). We also say that the scenario  $s$  is *successful* in round  $t$  of  $A$  if a packet is successfully transmitted in round  $t$  of a saturated execution of  $A$  in the scenario  $s$ . One can easily observe that  $s$  is successful in round  $t$  of  $A$  iff  $A_t$  hits  $s$ , i.e.,  $|A_t \cap s| = 1$ . Thus, the number of (saturated) executions from a given set  $S$  in which a packet is successfully transmitted in round  $t$  is equal to the number of elements of  $S$  which are hit by  $A_t$ .

Interestingly, the above characterization of successful transmissions in saturated executions in the scenarios from the set  $S$  is closely related to broadcasting in radio networks. Let us consider a restricted broadcasting problem, where all nodes from  $U$  know a message  $\mathcal{M}$  and the goal is to deliver  $\mathcal{M}$  to all nodes from  $S$  in a radio network described by  $G(U \cup S, E)$ . The radio network [2] is the model of distributed networks, where node  $u$  of a graph  $G$  receives a message from  $v$  in a round  $t$  iff  $v$  is the only (in)neighbor of  $u$  in  $G$  transmitting a message in  $r$ . One can observe that, for the set of transmitters  $A_t$ ,

- the number of successful scenarios from  $S$  in round  $t$ ,
- the number of elements of  $S$  which receive a message (from nodes in  $U$ ) in the radio network  $G(U \cup S, E)$  in a round in which the set of transmitters is equal to  $A_t$ , and
- the number of elements of  $S$  hit by  $A_t$

coincide. Using this relationship, we will be able to use the lower bound for broadcasting in radio networks from [2], which in particular implies that the above restricted broadcasting problem requires  $\Omega(\log^2 n)$  rounds.

In the nutshell, the proof of our impossibility goes as follows. Firstly, we focus merely on saturated executions. For a given set of scenarios  $S$ , we say that a round  $t$  is *c-sparse* if the number of successful scenarios from  $S$  in  $t$  wrt saturated executions is at most  $O(|S|/(c \log |S|))$ . (Recall that the set of successful scenarios from  $S$  in round  $t$  is equal to the set of those elements of  $S$  which are hit by the set of transmitters  $A_t$ .) A set of scenarios  $S$  is *universally c-sparse* if it is *c-sparse* for each possible round of each schedule (i.e., for each possible set of nodes transmitting in a round).

The main technical step towards our result is phrased in the lemma (Lemma 8) which says that, for some constant  $c_2 \geq 1$  and for some arbitrarily large  $n$ , there exists a universally  $c_2$ -sparse set of scenarios of size polynomial wrt  $n$ . Intuitively, this technical claim says that only a fraction of  $\frac{1}{c_2 \log |S|}$  scenarios from  $S$  are successful in each round of each oblivious schedule  $A$ . We prove this particular lemma by contradiction: assuming that it is false, we build a schedule for the restricted broadcasting problem in radio networks of length  $O(\log^2 n)$ , which contradicts the lower bound for this problem from [2].

The above described technical result implies that, on average, a scenario from a certain set  $S$  is successful once in each  $c_2 \log |S|$  rounds. This in turn implies that, for each oblivious algorithm  $A$  and each round  $t$ , there exists a scenario  $v \in S$  and time period  $[t, t + s)$  ( $s \geq n$ ) such that at most  $s\rho'$  packets are successfully transmitted during a saturated execution of  $A$  in the period  $[t, t + s)$  for the scenario  $v$  and some  $\rho' = \frac{1}{c_2 \log |S|} = O(1/\log n)$ .

However, the above reasoning holds only for saturated executions. In order to generalize this idea for arbitrary executions, we need to show that an adversary is able to inject packets in such a way that an actual execution works as a saturated one, for arbitrary long period of time. We deal with this challenge in Lemma 6, proved by contradiction.

#### 4.1.2 Formal proof of the impossibility result

Let  $n$  be the number of stations,  $\rho = \delta/\log n$  for some constant  $\delta$  be the injection rate of an adversary  $ADV$  and  $b > n$  be the burstiness of the adversary  $ADV$ . Let us fix an algorithm  $ALG$ . Let  $f(x)$  be an arbitrary function such that  $f(x) \rightarrow \infty$  if  $x \rightarrow \infty$  and  $f(x) = o(x)$ .

► **Theorem 5.** *There exist arbitrarily large numbers of stations  $n$  such that no non-adaptive scheduler can have bounded queues against an adversary with injection rate  $\rho = 1/\log n$  and burstiness  $b > n$ .*

To prove Theorem 5, we will need additional definitions and Lemma 6.

Let  $Alg(v, I)$  for some scenario  $v$  and time interval  $I$  be the number of packets successfully transmitted by  $ALG$  during interval  $I$  under a saturated execution.

An interval  $I = [t_b, t_e]$  is a  $\rho$ -bounded interval if there exists a scenario  $v \in S$  and a constant  $s^* = b - n \geq 1$  such that for every time  $t \in I$ :

$$Alg(v, [t_b, t]) - \rho' \cdot (t - t_b) < s^*.$$

► **Lemma 6.** *Consider  $\rho' = c/\log n < \rho$ . There exist arbitrarily large numbers of stations  $n$  and an infinite sequence of time prefixes  $\mathcal{T}_i$  such that for each prefix  $\mathcal{T}_i$  there exists a  $\rho'$ -bounded interval  $I \subseteq \mathcal{T}_i$  of length  $f(\tau_i)$ , where  $\tau_i = |\mathcal{T}_i|$ .*

**Proof of Theorem 5.** Consider an adversary  $ADV'$  with injection rate  $\rho' = O(1/\log n)$  and burstiness  $b = n + s^*$ . Let  $\mathcal{T}$  be one of the prefixes  $\mathcal{T}_i$  whose existence is postulated in Lemma 6 and  $\tau = |\mathcal{T}|$ . Let  $v$  be the scenario and  $I = [t_b; t_e]$  – the interval of length  $f(\tau)$  that correspond to  $\mathcal{T}$ .  $ADV'$  will wait without injecting any packets until  $t_b$ . Then in one round it will inject 1 packet to each station in scenario  $v$  (using at most  $n$  from its burstiness). After that (until  $t_e$ ), whenever  $ALG$  successfully transmits a packet in the scenario  $v$ ,  $ADV'$  injects a new packet to the station that just transmitted. According to Lemma 6  $ADV'$  can do this using at most additional  $s^*$  of its burstiness.

Therefore an adversary  $ADV$  with injection rate  $\rho = \omega(1/\log n)$  and burstiness  $b = n + s^*$  can inject an additional  $(\rho - \rho') \cdot f(\tau)$  packets into the system that will not be transmitted by time  $t_e$ . So as we pick longer prefixes  $\mathcal{T}$  (that exist according to Lemma 6), we get  $f(\tau) \rightarrow \infty$  and thus  $(\rho - \rho') \cdot f(\tau) \rightarrow \infty$ . So the queues are not bounded. ◀

Before we prove Lemma 6, we introduce a new structure and more auxiliary lemmas.

► **Definition 7.** A connected graph  $G$  is  $(n, c_2)$ -good if:

1.  $G$  is bipartite, with partition  $U$  and  $S$ ,
2.  $|U| = n$ ,
3.  $|S| = s = \text{poly}(n) \leq n^{c_2}$ ,
4.  $|S| \geq |U|$ ,
5. each node in  $S$  has at least one neighbour in  $U$ .



The set  $U$  represents original stations. Each element  $a \in S$  represents a scenario, where only the nodes adjacent to  $a$  in graph  $G$  receive packets.

► **Lemma 8.** *There exist arbitrarily large numbers  $n$ , such that there exists a  $(n, c_2)$ -good graph  $G = (U \cup S, E)$  such that in one round  $U$  can transmit a packet to at most  $\frac{s}{c_2 \log s}$  nodes in  $S$ , where  $s = |S|$  and  $c_2 \geq 1$  is some constant.*

We will prove the above lemma later (but we use it now).

**Proof of Lemma 6.** Proof by contradiction.

Consider an adversary  $ADV'$  with injection rate  $\rho' < \rho$ . According to Lemma 8 there exists arbitrarily large  $n$  and a  $(n, c_2)$ -good graph  $G$  such that in one round  $U$  can transmit a packet to at most  $\frac{s}{c_2 \log s}$  nodes in  $S$ .

Contrary, assume that for all scenarios  $v \in S$ , all intervals  $I = [t_b, t_e] \subseteq \mathcal{T}$  such that  $t_e - t_b = f(\tau)$ , there exists  $t \in I$  such that

$$Alg(v, [t_b, t]) - \rho' \cdot (t - t_b) \geq s^*.$$

Then we can split almost entire prefix  $\mathcal{T}' = \mathcal{T} \setminus (\tau - f(\tau), \tau]$  into small intervals  $J_1, J_2, \dots, J_k$  for some  $k$ , where  $J_i = [t_{i-1}, t_i)$  ends at time  $t_i$  for which the above inequality holds, where  $t_0 = 0$  is the beginning of prefix  $\mathcal{T}$  and  $t_k \in [\tau - f(\tau), \tau] \subseteq \mathcal{T}$ . So for all  $i$  we have  $|J_i| \leq f(\tau)$  and  $Alg(v, J_i) - \rho' \cdot |J_i| \geq s^*$ . For each  $J_i$  we have  $Alg(v, J_i) \geq \rho' \cdot |J_i|$ . We can sum this over all  $J_i$  to get

$$Alg(v, \mathcal{T}') \geq \sum_i \rho' \cdot |J_i| \geq \rho' \cdot (\tau - f(\tau) + 1).$$

Let  $B$  be the sum of all successful transmissions by the algorithm  $ALG$  across all considered scenarios  $S$ . So

$$B \geq \sum_{v \in S} ALG(v, \mathcal{T}') \geq s \cdot \rho' \cdot (\tau - f(\tau) + 1)$$

On the other hand, since the set of scenarios  $S$  is represented by a  $(n, c_2)$ -good graph  $G$ , in at most  $\frac{s}{c_2 \log s}$  scenarios a packet can be sent in one round of the algorithm (we picked our  $n$  and  $G$  according to Lemma 8). Therefore in  $\tau$  rounds there are at most  $\tau \cdot \frac{s}{c_2 \log s}$  packets sent across all scenarios. So

$$B \leq \tau \cdot \frac{s}{c_2 \log s}$$

Then we must have  $\tau \cdot \frac{s}{c_2 \log s} \geq s \cdot \rho' \cdot (\tau - f(\tau) + 1)$ , and thus  $\rho' \leq \frac{\tau}{(\tau - f(\tau) + 1) \cdot c_2 \log s}$ . Since  $f(\tau) = o(\tau)$ , for large enough  $\tau$  we have  $\frac{\tau}{(\tau - f(\tau) + 1)} < 2$  and so

$$\rho' \leq \frac{1}{c_2 \log s} = \frac{1}{c_2 \log n^{c_2}} = \frac{1}{c_2 c_2 \log n}$$

However  $\rho' = c/\log n$  and for  $c > 1/(c_2)^2$  we have a contradiction! ◀

**Proof of Lemma 8.** Proof by contradiction.

Suppose that for all  $s$ , for all  $(n, c_2)$ -good graphs  $G = (U \cup S, E)$  where  $|S| = n^{c_2} = s$ , there exists a subset of transmitters  $U$  that in one round can transmit a packet to more than  $\frac{s}{h(n) \log s}$  receivers in  $S$ , where  $h(n) = \omega(1)$  and  $h(n) = o(\log n)$ . We will build a

broadcasting algorithm for a graph  $H = (V, F)$  with source  $a$ , where  $V = U \cup S \cup \{a\}$  and  $F = E \cup \{(a, v) | v \in U\}$  that requires  $o(\log^2 n)$  rounds, which contradicts the result of Alon et al. [2].

**The broadcasting algorithm.** According to our assumption we can inform in one round  $\frac{s}{h(n) \log s}$  nodes in  $S$ . Let us define new sets  $S'$  and  $U'$ :

- set  $S'$  consists of all uninformed nodes from  $S$  (so  $|S'| \leq s - \frac{s}{h(n) \log s}$ ),
- each node in  $S'$  has at least one neighbour in  $|U'|$  (for each node  $v \in S'$  we take  $v$ 's arbitrary neighbour and add it to  $U'$ ; so far  $|U'| \leq |S'|$ ),
- if  $|U'| \leq |S'|^{1/c_2}$  then we take arbitrary nodes  $v \in U$  and add them to  $U'$  until  $|U'| = |S'|^{1/c_2}$  (note that  $|S'|^{1/c_2} \leq |S'|$ ).

Let  $G'$  be a subgraph of  $G$  induced by nodes  $U' \cup S'$ . Let  $s' = |S'|$  and  $n' = |U'|$ . Then the graph  $G'$  is a  $(n', c_2)$ -good graph.

We repeat our algorithm recursively until the size of set  $S$  is constant. Let  $T(s)$  be the number of iterations required.  $T(s) = 1 + T(s - \frac{s}{h(n) \log s})$ . The value of  $s$  will be halved after every  $\frac{s/2 \cdot \log(s/2)}{h(s/2)}$  iterations. So  $T(s) \leq \frac{s/2 \cdot \log(s/2)}{h(s/2)} + T(s/2)$ . Thus  $T(s) = o(\log^2 s)$ . Each iteration takes only 1 round, so we only used  $o(\log^2 s) = o(\log^2 n)$  rounds. ◀

## 4.2 Algorithm

In this section we show that there exists an oblivious algorithm which is stable for injection rates  $\rho = O(\frac{1}{\log^2 n})$ . Recall that we consider such protocols that the transmission pattern of each node is determined by ID of the node and the number of nodes  $n$ . That is, the pattern for a node is a 0-1 sequence.

► **Theorem 9.** *There exists a stable algorithm for injection rates  $\rho = O(\frac{1}{\log^2 n})$ , such that transmission pattern of each node is determined by its ID.*

We describe the proof of Theorem 9 in the remaining part of this section. More precisely, we show that a stable algorithm exists for each  $n$ . In the proof, we use Probabilistic Method.

The schedule for each ID  $i \in [n]$  will be just a 0-1 sequence of length  $W = O(n^2)$  repeated infinite number of times. (The exact value of  $W$  will be determined later.)

Let  $l = \log n$  (more precisely,  $l = \lfloor \log n \rfloor$ ). For a given  $W \in \mathbb{N}$ , we build a 0-1 sequence  $X_i$  of length  $W$  for each ID  $i \in [n]$  such that  $\text{Prob}(X_i[t] = 1) = \frac{1}{2^{1+i \bmod l}}$  for each  $i \in [W]$  and all probabilistic choices are independent. Moreover, we split  $W$  into consecutive phases of length  $l$ . That is, the first  $l$  elements of  $W$  form the first phase, the next  $l$  elements of  $W$  form the second phase, and so on. And, the number of phases is equal to  $F = W/l = W/\log n$ . Consider an execution of a schedule of length  $W$  determined by the above defined random sequences under the assumption that the number of injected packets during an execution of a schedule is at most  $\rho W$ , where  $\rho = O(1/\log^2 n)$ . (The actual value of the constant hidden in the big-O notation will be determined later.) Thus, an adversary can inject packets in at most  $\rho W = O(\frac{1}{\log^2 n} F \log n) = O(F/\log n)$  phases. Thus, no packet is injected in at least  $F(1 - \frac{1}{\log n}) \geq \frac{1}{2}F$  phases. A phase in which

- no packet is injected, AND
  - no queue becomes empty
- is called a *clear phase*.

Now, let us consider a clear phase. Let  $m$  be the number of nodes with non-empty queues at the beginning of such a phase. Moreover, let  $k \leq l = \lceil \log n \rceil$  be such that  $2^k < m \leq 2^{k+1}$ . Consider two cases:

- (i) no packets are successfully transmitted in the first  $k - 1$  rounds of the phase,
- (ii) at least one packet is transmitted in the first  $k - 1$  rounds of the phase.

For (i), the probability that a packet is successfully transmitted in the  $k$ th round of the phase is  $m \cdot \frac{1}{2^{k+1}} \cdot \left(1 - \frac{1}{2^{k+1}}\right)^{m-1} \geq \frac{1}{4e} = c$ , where  $e$  is the base for natural logarithms. In the case (ii) we are guaranteed that at least one packet is transmitted in a phase. Thus, the probability that (at least one) packet is transmitted in a clear phase is at least  $c$ , where  $c$  is a constant.

As the number of clear phases is at least  $\frac{1}{2}F$  and the probability of a successful transmission in a clear phase is  $\geq c$ , the expected number of successful transmissions in  $W$  rounds (i.e.,  $F$  phases) is  $\geq \frac{F}{2c} = \frac{W}{2c \log n} \geq 4\rho W$ , provided  $\rho = O(1/\log n)$  is sufficiently small. Significantly, as all the random choices are independent, we can use Chernoff bounds to estimate the probability that the actual number of transmitted packets is close to its expectation. More precisely, let  $X$  denote the number of transmitted packets in the considered  $W$  rounds. As we have shown,  $X$  is bounded from below by the sum  $X$  of  $F/c$  independent 0-1 random variables such that  $E[X] \geq \frac{F}{2c}$ , where  $c$  is a fixed constant. Thus,

$$\text{Prob}(X < 4\rho W) < \text{Prob}\left(X < \frac{F}{4c}\right) < \text{Prob}\left(X < \left(1 - \frac{1}{2}\right)E[X]\right) < e^{-E[X]/8} = e^{-\frac{W}{16c \log n}}. \tag{1}$$

**Derandomization.** Now, our goal is to show that, with non-zero probability, a randomly chosen oblivious schedule of length  $W$  (i.e.,  $n$  random 0-1 sequences of length  $W$ ) guarantees  $\geq 4\rho W$  successful transmissions irrespective of the injection pattern, provided the number of packets in queues at the beginning of an execution of this schedule is at least  $\geq F/(4c) \geq 4\rho W$  and the number of injected packets is at most  $\rho W$ . By Probabilistic Method, this fact will imply that there exists an oblivious schedule which guarantees  $4\rho W$  successful transmissions in  $W$  rounds, provided there are at least  $4\rho W$  packets at the beginning and at most  $\rho W$  packets are injected during the considered period of  $W$  rounds.

Let a *injection event* denote the event that a packet is injected in some round to some node. Moreover, let a *deletion event* denote the fact that the queue of a particular node becomes empty at particular round. Observe that the actual behaviour of a fixed schedule in a particular period  $\mathcal{T}$  of  $W$  rounds can be determined by the following factors:

- the set of nodes with non-empty queues at the beginning of  $\mathcal{T}$ ,
- injection events during  $\mathcal{T}$ , where each injection event is described by: the ID  $v$  of the node, the number of the round in which a packet is injected to  $v$ ,
- deletion events during  $\mathcal{T}$ , where each deletion event is described by: the ID  $v$  of the node and the number of the round in which the queue of  $v$  becomes empty.

According to our assumptions, the number of injection events is at most  $\rho W$ . Each deletion event corresponds to a unique successful transmission of a packet. Thus, if the number of deletion events is larger than  $4\rho W$ , then the number of transmitted packets is also  $\geq 4\rho W$ . Therefore, it is sufficient to consider the case that the number of deletion events is  $\leq 4\rho W$  and therefore the number of all events is at most  $5\rho W$ .

According to the above description of events, each event can be described in

$$\lceil \log n \rceil + \lceil \log W \rceil + 1 < 2(\log W + \log n) \leq 6 \log n$$

## 28:16 Local Queuing Under Contention

bits. This bound follows from the fact that  $\lceil \log n \rceil$  bits are sufficient to encode ID,  $\lceil \log W \rceil$  bits are sufficient to encode the round number in the schedule of length  $W$  and one bit can encode the type of the event, either an injection event or a deletion event. (Let us stress here that we do not need the actual information about the sizes of queues at the beginning, since we assume that deletion events can be determined by an adversary. This assumption generalizes the scenario, where the queue becomes empty only after transmitting all packets in it.) Thus, finally, the size of the set of all possible scenarios is at most

$$2^n \cdot 2^{5\rho W \cdot 6 \log n} < 2^{31\rho W \log n},$$

for large enough  $n$ , where

- $2^n$  is the number of possible sets of non-empty queues at the beginning;
- $2^{5\rho W \cdot 6 \log n}$  is the upper bound on the size of the set of possible events,
- the inequality follows from the fact that  $W = n^2$  and  $\rho = \Theta(1/\log^2 n)$ .

As the probability that the number of transmitted packets is smaller than  $4\rho W$  is at most  $e^{-\frac{W}{16c \log n}}$  for a fixed scenario (see (1)), the probability that a random schedule does not guarantee at least  $\frac{F}{4c}$  successful transmissions is at most

$$2^{31\rho W \log n} \cdot e^{-\frac{W}{16c \log n}} < 2^{31\rho W \log n - \frac{W}{16c \log n}} < 1,$$

provided  $\rho = O(1/\log^2 n)$  is small enough. Thus, by Probabilistic Method, there exists a schedule  $\mathcal{S}$  of length  $W$  which guarantees that at least  $\rho W$  packets are successfully transmitted provided that the number of packets at the beginning is at least  $\rho W$  and the number of injected packets is at most  $\rho W$ .

In the above reasoning we assumed that at most  $\rho W$  packets can be injected in  $W$  rounds. This is very restrictive, since the adversary can actually inject  $\rho W + b$  packets in  $W$  rounds, where  $b$  is the burstiness. Significantly,  $b$  is unknown and therefore a schedule is independent of  $b$ . In order to take this circumstance into account, let us consider any fixed  $b \in \mathbb{N}$ . Let  $\mathcal{S}$  be a schedule of length  $W$  which guarantees that  $4\rho W$  packets are transmitted during  $\mathcal{S}$ , provided there are at least  $4\rho W$  packets in queues at the beginning. We will analyze the schedule  $\mathcal{S}$  in consecutive *stages* of length  $T = 2bW$ . Thus, the schedule  $\mathcal{S}$  is repeated  $2b$  times in a stage. We will show that the number of packets in queues is smaller than  $2T$  at the beginning of each stage which in turn implies that the number of packets in queues is smaller than  $3T$  in each round. Consider two cases:

- (a) The number of packets in queues at the beginning of a stage is at least  $T$  and at most  $2T$ .
- (b) The number of packets in queues at the beginning of a stage is smaller than  $T$ .

For (b) observe that the adversary can inject at most  $\rho T + b < T$  packets in a stage and therefore the number of packets at the beginning of the next stage is at most  $2T$ . Thus, it remains to consider (a). We say that an execution of  $\mathcal{S}$  is *safe* if the number of injected packets during that execution is at most  $\rho W$ . As an adversary can inject at most  $\rho T + b = 2b \cdot \rho W + b$  packets in the stage, at least  $b$  out of  $2b$  executions of  $\mathcal{S}$  are safe. As shown above, at least  $4\rho W$  packets are transmitted during a safe execution of  $\mathcal{S}$ . Eventually, the number of packets transmitted in a stage of length  $T$  is at least  $b \cdot 4\rho W \geq 2\rho T > \rho T + b$ . Thus, the number of successfully transmitted packets in the stage is larger than the number of injected packets and therefore the number of packets in all queues at the beginning of the next stage is at most  $2T$ .

## 5 Conclusions

In this work we investigated how stability of local schedulers run on a shared channel depends on their adaptivity. A natural research direction includes studying of other types of schedulers, as well as delivering more quantitative measurements of schedulers' quality (such as latency, queue sizes, local memory size, etc.). Schedulers could also be studied in the context of other related models with contention, such as SINR or dependency-graph models.

---

### References

- 1 Norman M. Abramson. Development of the ALOHANET. *IEEE Transactions on Information Theory*, 31(2):119–123, 1985.
- 2 Noga Alon, Amotz Bar-Noy, Nathan Linial, and David Peleg. A lower bound for radio broadcast. *Journal of Computer and System Sciences*, 43(2):290–298, 1991. doi:10.1016/0022-0000(91)90015-W.
- 3 L. Anantharamu, Bogdan S. Chlebus, and Mariusz A. Rokicki. Adversarial multiple access channel with individual injection rates. In *Proceedings of the 13th International Conference on Principles of Distributed Systems (OPODIS)*, LNCS 5923, pages 174–188. Springer-Verlag, 2009.
- 4 Lakshmi Anantharamu, Bogdan S. Chlebus, Dariusz R. Kowalski, and Mariusz A. Rokicki. Deterministic broadcast on multiple access channels. In *Proceedings of the 29th IEEE International Conference on Computer Communications (INFOCOM)*, pages 1–5, 2010.
- 5 Matthew Andrews, Baruch Awerbuch, Antonio Fernández, Frank Thomson Leighton, Zhiyong Liu, and Jon M. Kleinberg. Universal-stability results and performance bounds for greedy contention-resolution protocols. *Journal of the ACM*, 48(1):39–69, 2001.
- 6 Michael A. Bender, Martin Farach-Colton, Simai He, Bradley C. Kuszmaul, and Charles E. Leiserson. Adversarial contention resolution for simple channels. In *Proceedings of the 17th Annual ACM Symposium on Parallel Algorithms (SPAA)*, pages 325–332, 2005.
- 7 Marcin Bieńkowski, Marek Klonowski, Mirosław Korzeniowski, and Dariusz R. Kowalski. Dynamic sharing of a multiple access channel. In *Proceedings of the 27th International Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 83–94, 2010.
- 8 Allan Borodin, Jon M. Kleinberg, Prabhakar Raghavan, Madhu Sudan, and David P. Williamson. Adversarial queuing theory. *Journal of the ACM*, 48(1):13–38, 2001.
- 9 Bogdan S. Chlebus, Dariusz R. Kowalski, and Mariusz A. Rokicki. Maximum throughput of multiple access channels in adversarial environments. *Distributed Computing*, 22(2):93–116, 2009.
- 10 Bogdan S. Chlebus, Dariusz R. Kowalski, and Mariusz A. Rokicki. Adversarial queuing on the multiple access channel. *ACM Transactions on Algorithms*, 8(1):5:1–5:31, 2012.
- 11 Jurek Czyżowicz, Leszek Gąsieniec, Dariusz R. Kowalski, and Andrzej Pelc. Consensus and mutual exclusion in a multiple access channel. In *Proceedings of the 23rd International Symposium on Distributed Computing (DISC)*, LNCS 5805, pages 512–526. Springer-Verlag, 2009.
- 12 Robert G. Gallager. A perspective on multiaccess channels. *IEEE Transactions on Information Theory*, 31(2):124–142, 1985.
- 13 Leszek Gąsieniec, Andrzej Pelc, and David Peleg. The wakeup problem in synchronous broadcast systems. *SIAM Journal on Discrete Mathematics*, 14(2):207–222, 2001.
- 14 Leslie Ann Goldberg, Mark Jerrum, Sampath Kannan, and Mike Paterson. A bound on the capacity of backoff and acknowledgment-based protocols. *SIAM Journal on Computing*, 33(2):313–331, 2004.

- 15 Leslie Ann Goldberg, Philip D. MacKenzie, Mike Paterson, and Aravind Srinivasan. Contention resolution with constant expected delay. *Journal of the ACM*, 47(6):1048–1096, 2000.
- 16 Albert G. Greenberg and Shmuel Winograd. A lower bound on the time needed in the worst case to resolve conflicts deterministically in multiple access channels. *Journal of the ACM*, 32(3):589–596, 1985.
- 17 J. Håstad, F. T. Leighton, and B. Rogoff. Analysis of backoff protocols for multiple access channels. *SIAM Journal on Computing*, 25(4):740–774, 1996.
- 18 Tomasz Jurdziński, Mirosław Kutylowski, and Jan Zatośniański. Efficient algorithms for leader election in radio networks. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing (PODC)*, pages 51–57, 2002.
- 19 Tomasz Jurdziński and Grzegorz Stachowiak. Probabilistic algorithms for the wakeup problem in single-hop radio networks. In *Proceedings of the 13th International Symposium on Algorithms and Computation (ISAAC)*, LNCS 2518, pages 535–549. Springer-Verlag, 2002.
- 20 János Komlós and Albert G. Greenberg. An asymptotically fast nonadaptive algorithm for conflict resolution in multiple-access channels. *IEEE Transactions on Information Theory*, 31(2):302–306, 1985.
- 21 Dariusz R. Kowalski. On selection problem in radio networks. In *Proceedings of the 24th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 158–166, 2005.
- 22 Eyal Kushilevitz and Yishay Mansour. An  $\Omega(D \log(N/D))$  lower bound for broadcast in radio networks. *SIAM Journal on Computing*, 27(3):702–712, 1998.
- 23 Robert M. Metcalfe and David R. Boggs. Ethernet: Distributed packet switching for local computer networks. *Communications of the ACM*, 19(7):395–404, 1976.
- 24 Prabhakar Raghavan and Eli Upfal. Stochastic contention resolution with short delays. *SIAM Journal on Computing*, 28(2):709–719, 1998.
- 25 Dan E. Willard. Log-logarithmic selection resolution protocols in a multiple access channel. *SIAM Journal on Computing*, 15(2):468–477, 1986.

# Derandomizing Distributed Algorithms with Small Messages: Spanners and Dominating Set

**Mohsen Ghaffari**

ETH Zurich, Switzerland  
ghaffari@inf.ethz.ch

**Fabian Kuhn**

University of Freiburg, Germany  
kuhn@cs.uni-freiburg.de

---

## Abstract

This paper presents improved deterministic distributed algorithms, with  $O(\log n)$ -bit messages, for some basic graph problems. The common ingredient in our results is a deterministic distributed algorithm for computing a certain *hitting set*, which can replace the random part of a number of standard randomized distributed algorithms. This deterministic hitting set algorithm itself is derived using a simple method of conditional expectations. As one main end-result of this derandomized hitting set, we get a deterministic distributed algorithm with round complexity  $2^{O(\sqrt{\log n \cdot \log \log n})}$  for computing a  $(2k - 1)$ -spanner of size  $\tilde{O}(n^{1+1/k})$ . This improves considerably on a recent algorithm of Grossman and Parter [DISC'17] which needs  $O(n^{1/2-1/k} \cdot 2^k)$  rounds. We also get a  $2^{O(\sqrt{\log n \cdot \log \log n})}$ -round deterministic distributed algorithm for computing an  $O(\log^2 n)$ -approximation of minimum dominating set; all prior algorithms for this problem were either randomized or required large messages.

**2012 ACM Subject Classification** Theory of computation → Distributed algorithms

**Keywords and phrases** Distributed Algorithms, Derandomization, Spanners, Dominating Set

**Digital Object Identifier** 10.4230/LIPIcs.DISC.2018.29

**Related Version** A full version of the paper is available at [15], <http://tr.informatik.uni-freiburg.de/reports/report285/report00285.pdf>.

## 1 Introduction and Related Work

We present improved deterministic distributed algorithms in the CONGEST model for graph problems including spanners and dominating set. Let us first recall the model definition.

**The CONGEST model [28] of distributed computing.** The network is abstracted as a simple  $n$ -node undirected graph  $G = (V, E)$ . There is one processor on each graph node  $v \in V$ , with a unique  $\Theta(\log n)$ -bit identifier  $\text{ID}(v)$ , who initially knows only its neighbors in  $G$ . Communication happens in synchronous rounds. Per round, each node can send one, possibly different,  $O(\log n)$ -bit message to each of its neighbors. At the end, each node should know its own part of the output. For instance, when computing spanners, each node should know whether each of its edges is in the computed spanner (a computed subgraph of  $G$ , to be defined later) or not. We note that the variant of the model where we allow unbounded size messages is known as the LOCAL model [24, 28].



© Mohsen Ghaffari and Fabian Kuhn;  
licensed under Creative Commons License CC-BY  
32nd International Symposium on Distributed Computing (DISC 2018).  
Editors: Ulrich Schmid and Josef Widder; Article No. 29; pp. 29:1–29:17



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

## 1.1 Our Contributions

### 1.1.1 Spanners

Graph spanners are a fundamental graph concept with a wide range of applications in distributed computing [4, 29]. For a graph  $G = (V, E)$ , a subgraph  $H = (V, E')$  is an  $\alpha$ -stretch spanner if each pairwise distance in  $H$  is at most an  $\alpha$  factor larger than the same distance in  $G$ . Ideally, we want spanners with small stretch and small number of edges. It is known that any  $n$ -node graph admits a  $(2k - 1)$ -stretch spanner with  $O(n^{1+1/k})$  edges [4, 29], and this tradeoff is optimal conditioned on a widely-believed girth conjecture of Erdős [14].

Baswana and Sen [8] gave a randomized algorithm in the CONGEST model for computing a  $(2k - 1)$ -stretch spanner with  $O(kn^{1+1/k})$  edges in  $O(k^2)$  rounds. Notice that  $k \in [1, \log n]$ . Hence, this is a  $\text{poly}(\log n)$  round randomized algorithm with spanner size within a logarithmic factor of the optimal. There was a series of works that eventually led to a  $\text{poly}(\log n)$  or even just  $O(k)$  round deterministic algorithm with a similar spanner size [10–12] but all these algorithms use large messages. Currently, there are only three deterministic algorithms that work in the CONGEST model. One is the work of Barenboim, Elkin, and Gavoille [7], which runs in  $\text{poly}(\log n)$  rounds, but has a considerably weaker stretch-size tradeoff: it computes a spanner with stretch  $O(\log^{k-1} n)$  and size  $O(n^{1+1/k})$  in  $O(\log^{k-1} n)$  rounds. The other two results obtain a near-optimal stretch-size tradeoff but their round complexity is considerably higher. Derbel, Mosbah, and Zemhari [13] gave an algorithm with round complexity  $O(n^{1-1/k})$  for computing a  $(2k - 1)$ -stretch spanner with size  $O(kn^{1+1/k})$ . Finally, very recently, Grossman and Parter [18] gave an algorithm with round complexity  $O(2^k n^{1/2-1/k})$  for computing a  $(2k - 1)$ -stretch spanner with size  $O(kn^{1+1/k})$ .

Our first result considerably improves on this line of work, leading to a sub-polynomial round complexity for a nearly optimal stretch-size tradeoff:

► **Theorem 1.** *There is a distributed deterministic algorithm in the CONGEST model that computes a  $(2k - 1)$ -stretch spanner with size  $O(kn^{1+1/k} \log n)$  in  $2^{O(\sqrt{\log n \log \log n})}$  rounds.*

### 1.1.2 Minimum Dominating Set

Minimum Dominating Set is another problem that has been central in the study of distributed algorithms for local problems, see e.g. [22]. Given a graph  $G = (V, E)$ , a set  $S \subseteq V$  is a dominating set of  $G$  iff each node  $v \in V$  is either in  $S$  or has a neighbor in  $S$ . Jia et al. [19] gave a randomized  $O(\log \Delta)$ -approximation in  $O(\log n \log \Delta)$  rounds of CONGEST model. Kuhn and Wattenhofer [22] gave a randomized distributed algorithm that computes an  $O(\sqrt{k} \Delta^{1/\sqrt{k}} \log \Delta)$ -approximation in  $O(k)$  rounds, e.g., an  $O(\log^2 \Delta)$  approximation in  $O(\log^2 \Delta)$  randomized rounds of CONGEST model. Later, Kuhn et al. [21] gave an  $O(\log \Delta)$  randomized approximation in  $O(\log n)$  rounds. Lenzen and Wattenhofer [23] pointed out that obtaining efficient deterministic algorithms for approximating minimum dominating set remains open. The only known result afterward is an algorithm of Barenboim et al. [7], which computes an  $O(n^{1/k})$ -approximation in  $O(\log^{k-1} n)$  rounds; however this algorithm uses large messages. The complexity of deterministic CONGEST-model algorithms for approximating minimum dominating set remains open.

Our second result provides the first answer to this question, by providing a sub-polynomial round complexity for poly-logarithmic approximation.



► **Theorem 2.** *There exists a distributed deterministic algorithm in the CONGEST model that computes an  $O(\log^2 n)$  approximation of minimum dominating set in  $2^{O(\sqrt{\log n \log \log n})}$  rounds.*

We remark that while it might be possible to improve this round complexity to  $2^{O(\sqrt{\log n})}$ , improving it further and especially to  $\text{poly}(\log n)$  would imply a major breakthrough in distributed graph algorithms: A result of Ghaffari, Harris, and Kuhn [17, Theorem 7.6] shows that obtaining a  $\text{poly}(\log n)$  approximation of minimum dominating set within  $\text{poly}(\log n)$  rounds is conditionally hard (even if we allow unbounded messages), because it would lead to a  $\text{poly}(\log n)$ -round deterministic algorithm for all locally checkable problems that admit  $\text{poly}(\log n)$  round randomized algorithms. This includes problems such as Maximal Independent Set (MIS) and  $(O(\log n), O(\log n))$ -network decomposition. Getting a  $\text{poly}(\log n)$ -round deterministic algorithm for these would resolve several well-known open question of distributed graph algorithms including that of Linial from 1987 about polylogarithmic deterministic MIS [24], and many of the open problems in the book of Barenboim and Elkin [6].

### 1.1.3 Network Decompositions and Neighborhood Covers

Network decompositions, first introduced by Awerbuch et al. [3], have been a key tool in developing efficient (deterministic) distributed algorithms for a variety of distributed algorithms. Given an  $n$ -node graph  $G = (V, E)$ , a  $(d(n), c(n))$ -network decomposition of  $G$  partitions it into a  $c(n)$  vertex-disjoint subgraphs, known as *blocks* of the decomposition (and indicated via different colors), such that in the subgraph induced by each block, each connected component (which is known as a *cluster* of this block) has a diameter at most  $d(n)$ . See Section 2 for the more formal definition. Awerbuch et al. [3] gave a deterministic algorithm with round complexity  $2^{O(\sqrt{\log n \log \log n})}$  for computing a  $(d(n), c(n))$ -network decomposition with  $d(n) = c(n) = 2^{O(\sqrt{\log n \log \log n})}$ . This was later improved by Panconesi and Srinivasan [27] to a  $2^{O(\sqrt{\log n})}$ -round LOCAL algorithm for decomposition with  $d(n) = c(n) = 2^{O(\sqrt{\log n})}$ . While the algorithm of [3] works in the CONGEST model, that of [27] requires large messages. See also the work of Barenboim et al. [7, Corollary 5.4], where a generalized tradeoff of network decomposition in the CONGEST model is presented.

All of these decomposition algorithms [3, 7, 27] fail to work in the CONGEST model when we need a larger separation between the clusters of the same block, i.e., when their distance should be two or more hops. This is actually something that significantly limits the power of these network decompositions for CONGEST model algorithms (e.g., for the applications in spanners and dominating sets).

As our third contribution, we present a CONGEST model network decomposition algorithm that can be used to compute a decomposition such that clusters of the same block are at least  $k$  hops apart. The statement of the result is presented below as Theorem 3, the proof of which is deferred to the full version [15], due to space limitations.

► **Theorem 3.** *Let  $G = (V, E)$  be an  $n$ -node graph and let  $k \geq 1$  be an integer. There is a deterministic CONGEST-model algorithm that computes a strong diameter  $k$ -hop  $(k \cdot f(n), f(n))$ -decomposition of  $G$  in  $k \cdot f(n)$  rounds, where  $f(n) = 2^{O(\sqrt{\log n \cdot \log \log n})}$ .*

In the above theorem,  $k$ -hop indicates that the clusters of the same block are at least  $k$  hops apart. See Section 2 for the more formal definition. The above theorem leads to the first efficient deterministic CONGEST model algorithm for neighborhood covers, another basic and central graph structure, which was introduced by Awerbuch and Peleg [5]. We refer to Section 2 for the related technical definition.

► **Corollary 4.** *Assume that we are given a strong diameter  $2k$ -hop  $(d, c)$ -decomposition of a graph  $G$ . One can compute a  $c$ -sparse  $k$ -neighborhood cover of diameter  $d + k$  in  $O(c(d + k))$  rounds in the CONGEST model on  $G$ . Consequently, for every  $k \geq 1$ , one can deterministically compute a  $2^{O(\sqrt{\log n \cdot \log \log n})}$ -sparse  $k$ -neighborhood cover of diameter  $k \cdot 2^{O(\sqrt{\log n \cdot \log \log n})}$  of an  $n$ -node graph  $G$  in  $k \cdot 2^{O(\sqrt{\log n \cdot \log \log n})}$  rounds of CONGEST.*

## 1.2 Our Method in a Nutshell, and Comparison with Prior Methods

Our spanner and minimum dominating set algorithms are developed also via network decompositions. We depart from the standard methodology in two parts. To outline these changes, we first review the standard methodology of algorithms that use network decompositions. We then comment on its shortcomings and outline how we go around each issue.

**The standard method for (deterministic) algorithms via Network Decomposition.** A standard technique in developing (deterministic) distributed algorithms for local graph problems (formally including Locally Checkable Labelings [26] and any other problem that can be formulated similarly using local constraints) is via the concept of  $(d(n), c(n))$ -network decompositions. The generic way to use them is to process the blocks sequentially in  $c(n)$  phases. In the  $i^{\text{th}}$  phase, for each connected component of the  $i^{\text{th}}$  block, one gathers the whole topology of the that component (and perhaps some extra information about neighboring nodes) in an elected center of the component, make that node decide about all (local) decisions of the nodes of the component, and deliver this information back to the nodes. Since different components are disconnected from each other, their decisions do not influence each other and thus can be performed in parallel.

**Shortcomings of the Generic Method via Network Decompositions for CONGEST.** The method is perfect for the LOCAL model with large messages. However, when it comes to using small messages – i.e., in the CONGEST model – the method has two shortcomings:

**Issue 1 – decompositions on power graphs.** For many local problems, the constraints are not only about the direct neighbors of a node but a small neighborhood of a distance  $r \geq 2$ . For instance, as we will see, in the case of spanner computations this radius  $r$  can be as large as  $O(\log n)$ . In such cases, we need to ensure that connected components of each block are  $r$  hops away from each other, instead of just not being adjacent. This is (almost) the same as computing a network decomposition of  $G^r$ , which denotes the graph with an edge between two vertices if their distance is at most  $r$  hops<sup>1</sup>. The algorithm provided by Awerbuch et al. [3] for computing network decompositions does not seem to extend to computing a decomposition for  $G^r$ , because of the congestion that the algorithm creates<sup>2</sup>. We present a CONGEST-model algorithm for network decompositions of power graphs  $G^r$ ; the formal statement is Theorem 3.

<sup>1</sup> Almost! Technically, we need that the components of each block are also connected in the base graph  $G$  so that we can run CONGEST model algorithms in each component independently, i.e., each edge of  $G$  has to pass messages for one component.

<sup>2</sup> We note that this is not a small technicality: The CONGEST model complexity of problems can be significantly different between  $G$  and even  $G^2$ . For instance, the problem of each node knowing the its degree in  $G$  and  $G^2$  are very different. The former has a single-round CONGEST-algorithm, while there is no known  $o(n)$ -round algorithm for the latter.

**Issue 2 – gathering topology in each component.** The generic method of using network decompositions, each component is solved by gathering the whole topology of the component (and some neighborhood outside) and then solving the problem in a brute-force centralized manner. One can argue that this brute-force centralized computation is quite a stretch for the notion of having a *distributed* method of solving the problem.

The method we use to go around this issue is a derandomization of randomized distributed algorithms, which can typically solve the local problems that we are considering in  $\text{poly}(\log n)$  rounds. We outline the method here. Most parts are generic and applicable to various problems, except the last part, which is specific to the constraints of each problem. We observe that for many problems, including spanners and dominating set, the corresponding efficient randomized algorithm can be made to work with only  $\text{poly}(\log n)$  bits of randomness, using concepts such as  $k$ -wise independence. We refer to these bits as the *seed of randomness*. Then, derandomization is just a matter of determining a deterministic assignment to these  $\text{poly}(\log n)$  bits while preserving certain properties of the output of the randomized algorithm. For that purpose, following an approach of Luby [25], we use the method of conditional expectations to fix the bits one by one. The only remaining piece of the algorithm is to check whether a bit should be 0 or 1. This requires us to be able to learn, or estimate, the expected number of unsatisfied local constraints. This last part will be done using a method specific for each problem, depending on its constraints.

### Comparison with the methods of Censor-Hillel, Parter, and Schwartzman [9] and Grossman and Parter [18]

We note that this second part of our contribution as described in issue 2 above – namely, the method of conditional expectation applied on a random algorithm that uses only  $\text{poly}(\log n)$  bits of randomness overall – is inspired by the work of Luby [25] in parallel algorithms and the recent work of Censor-Hillel, Parter, and Schwartzman [9] in distributed CONGEST and CONGESTED-CLIQUE model algorithms. Let us explain how our approach differs from that of [9], thus allowing us to improve on the bounds of [18].

Censor-Hillel et al. [9] give an  $O(D \log^2 n)$ -round CONGEST algorithm for maximal independent set (MIS), by derandomizing the randomized MIS algorithm of [16], using a method of conditional expectation close to Luby [25]. The key difference there is that (1) the complexity depends on the global diameter  $D$ , (2) for MIS, each of the constraints in the method of conditional expectation spans only the neighbors of one node and therefore, computing an upper bound on the score/cost function is much easier. In our case, we want a complexity that is considerably sublinear in the diameter, which calls for network decompositions. Moreover, for spanner and dominating set (and presumably many other local problems), the constraints span  $k$ -hop neighborhoods for some  $k \geq 2$ , instead of direct neighborhood. This causes two challenges: (A) we need a network decomposition of the power graphs, which prior to our work was not known in the CONGEST model, as explained above in issue 1. (B) Even computing each part of the cost function now spans  $k$ -hop neighborhood for some  $k \geq 2$ , and evaluating it with CONGEST-model messages requires a different method.

Censor-Hillel et al. [9] also give a derandomized spanner algorithm in the CONGESTED-CLIQUE model, where all node-pairs can communicate with each other, exchanging  $O(\log n)$  bits per round. This also follows a derandomization method inspired by that of Luby [25]. However, again there two differences, which limit that result from extending to our setting: (A)

this derandomization does not need to work with network decompositions and especially power-graph network decompositions, because everything is within one hop in the CONGESTED-CLIQUE and one can share the seed of randomness to all nodes, (B) computing the score/cost function, which spans  $k$ -hop neighborhoods, is much easier in the CONGESTED-CLIQUE, because this model does not suffer from the locality constraint.

In both cases above, both of the issues appear quite non-trivial to us. Indeed, Censor-Hillel et al. [9] comment that the best deterministic CONGEST algorithm for spanners takes barely sublinear time,  $O(n^{1-1/k})$  rounds to be precise. That is much higher than the sub-polynomial time that we achieve. This  $O(n^{1-1/k})$  bound was improved to nearly  $O(\sqrt{n}) - O(n^{1/2-1/k} \cdot 2^k)$  rounds to be precise – in the simultaneous work of Grossman and Parter [18], using a special and well-crafted deterministic method for constructing spanners, and particularly without attempting a derandomization. We now show that the derandomization techniques can be extended and improved, along with the strengthened power-graph network decomposition, to achieve a round complexity  $2^{O(\sqrt{\log n \cdot \log \log n})}$  rounds.

**Some Other Related Work.** Ghaffari, Harris, and Kuhn [17] also use some variant of a method of conditional expectation to obtain derandomized distributed algorithms, but for all of their results, locality is the main topic, and their algorithms use large messages. Kawarabayashi and Schwartzman [20] present distributed derandomizations for some other problems, including max cut and max  $k$ -cut. These work by turning a sequential process to a distributed process by going through the colors of a certain (defective) graph coloring one by one. However, those methods cannot extend to the problems that we consider as there the score/cost functions are very local (spanning single neighborhoods), whereas in our case, the constraints span up to  $\log n$ -neighborhood, which means a suitable coloring would require even up to polynomial many colors.

## 2 Model and Definitions

**Mathematical Notation.** For a graph  $G = (V, E)$  and two nodes  $u, v \in V$ , we define  $d_G(u, v)$  to be the hop distance between  $u$  and  $v$ . For an integer  $k \geq 1$ , we define  $G^k = (V, E')$  to be the graph with an edge  $\{u, v\} \in E'$  whenever  $d_G(u, v) \leq k$ . Given a node  $v \in V$ , we use  $N_{G,k}(v) := \{u \in V : d_G(u, v) \leq k\}$  to denote the set of nodes within distance  $k$  of  $v$  in  $G$ . For a node set  $S \subseteq V$ , we use the shorthand notation  $N_{G,k}(S) := \bigcup_{v \in S} N_{G,k}(v)$  and we drop the subscript  $G$  if it is clear from the context. Throughout, we use  $\ln(\cdot)$  to refer to natural logarithm and  $\log(\cdot)$  to refer to logarithms to base 2. Moreover, for a graph  $G = (V, E)$ , integers  $a \geq 1$  and  $b \geq 0$  and a node set  $V' \subseteq V$ , a set of nodes  $S \subseteq V'$  is called a  $(a, b)$ -ruling set of  $G$  w.r.t.  $V'$  [3] if (A) for any two nodes  $u, v \in S$ , we have  $d_G(u, v) \geq a$ , and (B)  $\forall u \in V' \setminus S$ , there is a node  $v \in S$  such that  $d_G(u, v) \leq b$ . If  $V' = V$ ,  $S$  is simply called an  $(a, b)$ -ruling set of  $G$ .

**Network Decomposition.** A network decomposition of a graph  $G$  is given by a clustering of  $G$  and a coloring of the graph induced by contracting each cluster. We therefore first define the notion of a *cluster graph*.

► **Definition 5 (Cluster Graph).** Given a graph  $G = (V, E)$  and an integer parameter  $d \geq 1$ , an  $(N, d)$ -cluster graph  $\mathcal{G} = (\mathcal{V}, \mathcal{E})$  of  $G$  is a graph that is given by a set of  $N \geq 1$  clusters  $\mathcal{V} := \{C_1, \dots, C_N\} \in 2^V$  such that (a) the clusters  $C_1, \dots, C_N$  form a partition of  $V$ , (b)

each cluster  $C_i$  induces a connected subgraph  $G[C_i]$  of  $G$ , (c) each cluster  $C_i$  has a leader node  $\ell(C_i)$  that is known by all nodes of  $C_i$ , and (d) inside each cluster, there is a rooted spanning tree  $T(C_i)$  of  $G[C_i]$  that is rooted at  $\ell(C_i)$  and has diameter at most  $d$ . There is an edge  $\{C_i, C_j\}$  between two clusters  $C_i, C_j \in \mathcal{V}$  if there is edge in  $G$  connecting a node in  $C_i$  to a node in  $C_j$ . The identifier  $\text{ID}(C_i)$  of a cluster  $C_i$  is its leader's ID.

Given a cluster graph  $\mathcal{G} = (\mathcal{V}, \mathcal{E})$  of  $G$  and an integer  $k \geq 1$ , we say that two clusters  $C, C' \in \mathcal{V}$  are  $k$ -separated if for any two nodes  $u$  and  $v$  of  $G$  such that  $u \in C$  and  $v \in C'$ , we have  $d_G(u, v) > k$ . A strong-diameter  $k$ -hop network decomposition of a graph  $G$  is then defined as follows.

► **Definition 6 (Network Decomposition).** Let  $G = (V, E)$  be a graph and let  $k \geq 1$ ,  $d \geq 0$ , and  $c \geq 1$  be integer parameters. A *strong diameter  $k$ -hop  $(d, c)$ -decomposition* of  $G$  is a  $(N, d)$ -cluster graph  $\mathcal{G}$  of  $G$  for some integer  $N \geq 1$  together with a coloring of the clusters of  $\mathcal{G}$  with colors  $\{1, \dots, c\}$  such that any two clusters with the same color are  $k$ -separated.

**Sparse Neighborhood Covers.** The notion of sparse neighborhood covers as introduced by Awerbuch and Peleg [5] is closely related to network decompositions.

► **Definition 7 (Sparse Neighborhood Cover).** Let  $G = (V, E)$  be a graph and let  $k \geq 1$ ,  $d \geq 1$ , and  $s \geq 1$  be three integer parameters. A  *$s$ -sparse  $k$ -neighborhood cover of diameter  $d$*  is a collection of clusters  $C \subseteq V$  such that (a) for each cluster  $C$ , we have a rooted spanning tree of  $G[C]$  of diameter at most  $d$ , (b) each  $k$ -neighborhood of  $G$  is completely contained in some cluster, and (c) each node of  $G$  is in at most  $s$  clusters.

As we explain in the proof of Corollary 4, any  $2k$ -hop  $(d, c)$ -decomposition leads to a  $c$ -sparse  $k$  neighborhood cover of diameter  $d + k$ .

### 3 Hitting Set

In this section, we define an abstract problem, which we call the *hitting set* problem. This problem, which can be solved easily using randomized algorithms, captures a variety of the usual applications of randomness in distributed algorithms. In this section we provide a deterministic algorithm for solving this *hitting set* problem. In the later sections, we see how to use this deterministic subroutine to develop deterministic algorithms for other problems such as spanners and minimum dominating set, primarily by replacing their randomized parts with this deterministic hitting set subroutine.

Our main formulation of the hitting set problem (which is presented below in Definition 8 and solved in Lemma 9) is tailored to its usage in our spanner result. At the end of this section, in Lemma 10, we provide an alternative formulation and the corresponding deterministic algorithm, which are more suitable for our minimum dominating set result. The proofs are quite similar.

► **Definition 8 (The Hitting Set Problem).** Consider a graph  $G = (V, E)$  with two special sets of nodes  $L \subseteq V$  and  $R \subseteq V$  with the following properties: each node  $\ell \in L$  knows a set of vertices  $R(\ell) \subseteq R$ , where  $|R(\ell)| = \Theta(p \log n)$ , such that  $\text{dist}_G(\ell, r) \leq T$  for every  $r \in R(\ell)$ . Here,  $p$  and  $T$  are two given integer parameters in the problem. Moreover, there is a  $T$ -round CONGEST algorithm that can deliver one message from each node  $r \in R$  to all nodes  $\ell \in L$  for which  $r \in R(\ell)$ . We emphasize that the same message is delivered to all nodes  $\ell \in L$ .

Given this setting, the objective in the hitting set problem is to select a subset  $R^* \subseteq R$  such that (I)  $R^*$  dominates  $L$  – i.e., each node  $\ell \in L$  has at least one node  $r^* \in R^*$  such that  $\ell \in R(r^*)$  – and (II) we have  $|R^*| \leq |R|/p$ .

► **Lemma 9.** *Given a  $2T$ -hop  $(d, c)$ -decomposition of the graph  $G$  of the hitting set problem, there is a deterministic distributed algorithm that in  $\tilde{O}(c(d+T))$  rounds solves the hitting set problem.*

**Proof.** The trivial randomized algorithm includes each node of  $R$  in  $R^*$  with probability  $1/(2p)$ . It is easy to verify that this satisfies the requirements (I) and (II), with high probability. In this proof, we develop a deterministic algorithm for solving the hitting set problem, effectively by derandomizing this randomized process. This derandomization has four aspects, which we discuss one by one.

**Point 1 – Transforming the Requirements to One Cost Function.** We try to capture the requirements (I) and (II) with one cost function. In particular, we define a *cost function* for any fixed set  $R^* \subseteq R$  as follows. Consider the following indicator (random) variables: for each node  $\ell \in L$ , define  $x_\ell = 1$  iff  $R(\ell) \cap R^* = \emptyset$ . Moreover, for each node  $r \in R$ , define  $y_r = 1$  iff  $r \in R^*$ . Define the cost function as  $Z = \sum_{\ell \in L} x_\ell + \sum_{r \in R} y_r$ . Notice the value is clearly a function of the choice of  $R^* \subseteq R$ . Furthermore, it is easy to see that in the natural randomized algorithm that includes each node of  $R$  in  $R^*$  with probability  $1/(2p)$ , we have  $\mathbb{E}[Z] \leq |R|/(2p) + 1/n^2$ . This is because  $\mathbb{E}[\sum_{r \in R} y_r] = \sum_{r \in R} \mathbb{E}[y_r] = \sum_{r \in R} 1/(2p) = |R|/(2p)$ . Moreover, for each  $\ell \in L$ , we have  $\mathbb{E}[x_\ell] = \Pr[x_\ell = 1] = (1 - 1/(2p))^{\Theta(p \log n)} \leq 1/n^3$ , which implies  $\mathbb{E}[\sum_{\ell \in L} x_\ell] \leq 1/n^2$ .

During the next three points presented below, we will describe a deterministic process for selecting  $R^*$  such that the related cost is at most  $|R|/(2p) + 1/n$ . Notice that this still does not mean that  $R^*$  satisfies (I). To take care of that issue, we perform the following clean up step, which has round complexity  $T$ , at the end: Suppose we have already chosen a subset  $R^* \subseteq R$  such that the cost  $Z = \sum_{\ell \in L} x_\ell + \sum_{r \in R} y_r$  of this selected set  $R^*$  is at most  $|R|/(2p) + 1/n$ . The number of nodes  $\ell \in L$  for which  $R(\ell) \cap R^* = \emptyset$  is  $\sum_{\ell \in L} x_\ell$ . By definition, these are exactly the vertices for which requirement (I) is not satisfied. For each such node  $\ell$ , we mark one node  $r \in R(\ell)$  arbitrarily and add the marked nodes to  $R^*$ . This can be done in  $T$  rounds by reversing the communication from  $R$  to  $L$ , now delivering one bit to each node  $r \in R$  of whether any of the nodes  $\ell \in L$  for which  $r \in R(\ell)$  marked  $r$  or not. These marked nodes, which are added to  $R^*$ , increase the size of  $R^*$  by at most  $\sum_{\ell \in L} x_\ell$ . Thus, the total new size of  $R^*$  is at most  $\sum_{r \in R} y_r + \sum_{\ell \in L} x_\ell \leq |R|/(2p) + 1/n \leq |R|/p$ . Hence, now we have a set  $R^*$  that satisfies all the requirements (I) and (II).

**Point 2 – Limited Independence Suffices.** Next, we describe how we deterministically select a set  $R^*$  with cost at most  $Z \leq |R|/(2p) + 1/n$ . To be able to pick such a set  $R^*$  deterministically, it is helpful to have a randomized process that uses only a small number of random bits. For this reason, we first explain how to replace the fully random process of selecting  $R^*$  nodes with another random process that uses less randomness, in a certain sense to be formalized, but still provides the same guarantee on the expected cost. We will then derandomize this randomness-efficient random process.

Let us think of the decisions of whether a node  $r \in R$  is included in  $R^*$  or not as a function  $f : R \rightarrow \{0, 1, 2, \dots, 2p-1\}$  where  $f(r) = 0$  means  $r \in R^*$  and all other values mean  $r \notin R^*$ . Notice that if for each  $r \in R$ ,  $f(r)$  is chosen uniformly at random

from  $\{0, \dots, 2p - 1\}$ , then we have  $\Pr[r \in R^*] = 1/(2p)$ , as desired. Following standard terminology, we say that a family  $\mathcal{F}$  of functions  $f : R \rightarrow \{0, 1, 2, \dots, 2p - 1\}$  is  $k$ -wise independent if for any set  $S = \{s_1, s_2, \dots, s_k\} \subset R$  with  $|S| = k$  and any choice of values  $b_1, b_2, \dots, b_k \in \{0, 1, 2, \dots, 2p - 1\}$ , we have that

$$\Pr_{f \in \mathcal{F}}[f(s_1) = b_1 \& \dots \& f(s_k) = b_k] = (1/(2p))^k.$$

That is, upon selecting a function  $f$  uniformly at random from  $\mathcal{F}$ , the probability of the values of  $f$  over set  $S$  is exactly the same as in the fully random function. The advantage of  $k$ -wise independent functions is that the corresponding family is quite small and thus, we can choose one function in the family using considerably less randomness. This is made more clear in the next point. Moreover, they still provide many of the nice behaviors expected from truly random functions. In particular, using the extensions of standard Chernoff bound to functions with limited independence [30], we can see that if the selection function for choosing  $R^*$  out of  $R$  is  $k = \Theta(\log n)$ -wise independent (i.e., if it is chosen randomly from a  $k$ -wise independent family), then we still have a concentration within a constant factor what would be implied by the standard Chernoff bound. More concretely, we still have  $\Pr[x_\ell = 1] \leq 1/n^3$  for each  $\ell \in L$ . Hence, even with a  $k$ -wise independent selection function  $f$ , we have that the expected cost is small as desired, i.e.,  $\mathbb{E}[Z] \leq |R|/(2p) + 1/n^2$ .

**Point 3 – Defining a  $k$ -wise Independent Selection Process.** To define a  $k$ -wise independent selection function in a manner that is suitable for our network decomposition, we use an independent function for the vertices of each cluster  $C$  of the decomposition. Hence, we have full independence among different colors and even among clusters of the same color. However, inside each cluster  $C$ , the selections are made using one  $k$ -wise independent function  $g(C) : R \cap C \rightarrow \{1, 2, \dots, 2p\}$ . One can easily see that such a combination of independent random functions, each of which is  $k$ -wise independent, is also a  $k$ -wise independent function.

To select a  $k$ -wise independent selection function for cluster  $C$ , we rely on classic constructions of  $k$ -wise independent functions. It is known [1] that there is a family  $\mathcal{G}$  of  $n^{O(k)}$  deterministic functions such that if we pick one function from  $\mathcal{G}$  uniformly at random, we have a  $k$ -wise independent random function. This family can be known to all nodes of the cluster; they can all construct it by following the deterministic sequential construction of [1]. To randomly and uniformly sample one member of this family  $\mathcal{G}$ , which has  $n^{O(k)}$  members, merely  $O(k \log n)$  bits of randomness suffice. Hence, by using a random function defined via  $O(k \log n) = O(\log^2 n)$  bits of randomness for each cluster, we can define a random selection function for vertices of  $R$  which ensures that  $\mathbb{E}[Z] \leq |R|/(2p) + 1/n^2$ .

**Point 4 – Fixing the Bits of Randomness.** We now fix the bits of randomness in the above random selection of  $R^*$ , in  $c$  phases. In the  $i^{\text{th}}$  phase, we decide about the vertices of  $R$  that are in the  $i^{\text{th}}$  color of network decomposition, whether to include each of them in  $R^*$  or not. This gradual process will be such that, at each point of time, the conditional expectation of the cost function, conditioned on the already decided vertices, is at most  $|R|/(2p) + 1/n^2$ . Hence, once we finish the process, a set  $R^*$  is selected with cost at most  $|R|/(2p) + 1/n^2$ .

Fix a color  $i$ . We fix the bits of randomness in each cluster of color  $i$ . Since clusters of this color are at least  $2T$  hops apart in  $G$ , each variable  $x_\ell$  or  $y_v$  in the cost function  $Z = \sum_{\ell \in L} x_\ell + \sum_{r \in R} y_r$  is influenced by the randomness fixing of at most one cluster. Hence, each cluster  $C$  can fix its own randomness independent of the other clusters.



Let us focus on one cluster  $C$  in color  $i$ . We have a family of  $\mathcal{G}$  of  $n^{O(k)}$  deterministic functions for the selection of the  $R^*$ -nodes among  $R \cap C$ . We pick one function from  $\mathcal{G}$  by fixing the corresponding bits of randomness one by one, in a manner that does not increase the conditional expectation of  $Z$ , given prior assignments. Imagine that all the functions in the family  $\mathcal{G}$  are indexed with numbers from 1 to  $n^{O(k)}$ , and suppose that these indices are written as binary numbers with  $O(k \log n)$  bits. Consider the process of fixing the first bit; the next bits are similar. Break the family  $\mathcal{G}$  of  $n^{O(k)}$  assignment functions into two subfamilies,  $\mathcal{G}_0$  which are those that their function index starts with bit 0, and  $\mathcal{G}_1$  which are those that their function index starts with bit 1. For each subfamily, we compute the conditional expectation of  $Z$  over the variables in  $N^T(C)$  – i.e., the  $T$ -hop neighborhood of cluster  $C$  – when the assignment function is chosen uniformly at random from this subfamily. We then fix the first bit of randomness according to whichever leads to a smaller expectation, i.e., that is, we zoom in to one of subfamilies  $\mathcal{G}_0$  and  $\mathcal{G}_1$ , in our search for a deterministic assignment function. We next explain why the expectation of  $Z$  over the variables in  $N^T(C)$  can be computed in  $O(d + T)$  time.

We first spend  $T$  rounds to deliver one message from each node  $r \in R$  to all nodes  $\ell \in L$  for which  $r \in R(\ell)$ . In this message, node  $r$  reports its color and cluster center ID, and whether node  $r$  has been put in  $R^*$  or not if the color of  $r$  was some  $j < i$ . Thus, each node  $\ell$  in  $N^T(C) \cap L$  can learn whether it is already hit or not, i.e., whether any of the nodes in  $R(\ell)$  in the previous color clusters has been fixed to be in  $R^*$  or not. If there is already some such node  $r \in R(\ell) \cap R^*$ , then  $x_\ell = 0$  and it will not change. If not, the expectation of  $x_\ell$  can change by the assignments in  $C$ . In this case, node  $\ell$  can exactly compute  $\mathbb{E}[x_\ell] = Pr[x_\ell = 1]$  because it knows all the nodes in  $R(\ell)$ , those of colors less than  $i$  that their decisions have been made in the previous phases, the identifiers of those that are being decided in this phase, the colors and cluster identifiers of those with colors greater than  $i$  which will be decided in the next phases, and also the subfamily  $\mathcal{G}_0$  or  $\mathcal{G}_1$  in consideration. Similarly, each node  $r \in R \cap C$  can compute  $\mathbb{E}[y_r]$  because that only depends on the identifier of the node  $r$  and the subfamily  $\mathcal{G}_0$  or  $\mathcal{G}_1$  in consideration. Then, we can spend  $d$  rounds to perform a convergecast on the tree of cluster  $C$  to gather the summation of these expectations at the root<sup>3</sup>.

Once these two expectations are gathered at the root of the cluster  $C$ , we go with the smaller one and zoom into the corresponding subfamily, among  $\mathcal{G}_0$  or  $\mathcal{G}_1$ . This fixes the first bit of randomness in  $C$  but does not increase the conditional expectation of the cost function compared to when the assignment function was chosen from  $\mathcal{G}$ . We then proceed to the next bit. After going through all the  $O(k \log n) = O(\log^2 n)$  bits, which takes  $O(d \log^2 n)$  rounds, we have fixed all the bits and thus we have chosen a deterministic assignment for the  $R$  vertices of cluster  $C$  in a manner that did not increase the conditional expectation of the cost function. This finishes the process for one color. We then proceed to the next color and perform a similar process. After going through all colors, which takes  $\tilde{O}(c(d + T))$  rounds, we have found a set  $R^* \subseteq R$  such that the cost  $Z = \sum_{\ell \in L} x_\ell + \sum_{r \in R} y_r$  of this selected set  $R^*$  is at most  $|R|/(2p) + 1/n$ . As described in point 1 above, this set  $R^*$  can be augmented to satisfy all the requirements of the hitting set problem, in  $T$  additional rounds. ◀

<sup>3</sup> We note that in the CONGEST model, we may not be able to convergecast the full precision of the expectation, but may need to truncate it to  $\Theta(\log n)$  bits of precision. This would increase the expectation by at most  $1/\text{poly}(n)$ . This is negligible even over all the at most  $n$  iterations that we perform such a convergecast and subfamily selection.



**A Modified Variant of Hitting Set.** We can use a similar method to solve a slightly modified variant of the hitting set problem, as stated in the following lemma. The proof is deferred to the full version. We use this variant in our dominating set algorithm.

► **Lemma 10** (An Alternative Hitting Set Lemma). *Let  $H = (L \cup R, E)$  be a bipartite graph and let  $p \geq 1$  be an integer parameter. Further assume that there is a spanning tree of diameter  $D$  that spans all nodes of  $H$  and that we can use the edges in  $E$  and the spanning tree edges for communication. There is a deterministic  $\tilde{O}(D)$ -time CONGEST-model algorithm that selects a subset  $R^* \subseteq R$  of the nodes in  $R$  such that the following conditions hold:*

- (a) *For all nodes  $u \in L$ , the number of neighbors in  $R^*$  is at most  $O(\deg(u)/p + \log n)$ .*
- (b) *For all nodes in  $u \in L$  with  $\deg(u) \geq cp \log n$  for a sufficiently large constant  $c > 0$ , at least one neighbor of  $u$  is in  $R^*$ .*

## 4 Spanners

Here, we present the proof of Theorem 1, i.e., we develop a deterministic distributed algorithm for computing spanners by derandomizing the algorithm of Baswana and Sen [8], using our hitting set. We first briefly recall the algorithm of Baswana and Sen.

**Baswana-Sen's Spanner Algorithm.** The algorithm has  $k$  levels, where we gradually build, and sometimes dissolve clusters. At level  $i$ , each cluster induces a tree of depth at most  $i - 1$  rooted at the corresponding cluster center. Initially, each node is one cluster. In the  $i^{\text{th}}$  level for  $i \in \{1, 2, \dots, k - 1\}$ , each cluster of the previous level is active with probability  $n^{-1/k}$  and inactive otherwise. This randomized decision is made by the corresponding cluster center. Then, inactive clusters get dissolved and their nodes either join other clusters or get dropped from the algorithm permanently. For each node  $v$  in an inactive cluster, if it has a neighbor in an active cluster, then  $v$  joins the cluster of one such neighbor  $u$ , and adds the edge  $\{v, u\}$  to the tree of that cluster. If  $v$  has no neighbor in an active cluster, then  $v$  gets dropped from the rest of the algorithm. But just before that, for each inactive cluster  $\mathcal{C}$  that contains a neighbor of  $v$ , node  $v$  adds to the spanner one edge to some neighbor in  $\mathcal{C}$ . Moreover, for each cluster of this level, we add the corresponding tree rooted in the cluster center to the spanner. This finishes level  $i$ , and we then proceed to the next level. In the very last level, all clusters are considered inactive and we act accordingly.

### Properties of the Spanner algorithm of Baswana and Sen.

- (1) **Round Complexity:** Clearly, the  $i^{\text{th}}$  level can be implemented in  $O(i)$  rounds of the CONGEST model and thus the whole algorithm takes  $O(k^2)$  rounds.
- (2) **Stretch:** Eventually, all clusters are dissolved. For each edge  $\{v, u\}$  in the graph, suppose without loss of generality that  $v$  gets dropped from the clustering no later than  $u$ . Then, an edge is added to the spanner from  $v$  to some node  $w$  in the cluster of  $u$ . If  $w = u$ , edge  $\{v, u\}$  is in the spanner. Otherwise, there is an alternate route to go from  $v$  to  $u$  in the spanner by going to  $w$  and then using the cluster tree of  $u$  at that level; potentially going from  $w$  to its cluster center and then coming back to  $u$ . Since the tree has depth at most  $i - 1 \leq k - 1$ , the whole path has length at most  $2k - 1$ . That is, edge  $\{v, u\}$  has stretch at most  $2k - 1$ .
- (3) **Spanner Size:** The total number of cluster tree edges, over all levels, is  $O(nk)$ . Each node gets dropped in some level, when it has no active neighboring cluster, and then adds one edge connecting it to each (inactive) neighboring cluster to the spanner. If the node has

more than  $\Theta(n^{1/k} \log n)$  neighboring clusters, w.h.p., it will have an active neighboring cluster. So the number of added edges per node is with high probability no more than  $\Theta(n^{1/k} \log n)$ . This is also true for the last level as there the total number of clusters is  $\Theta(n^{1/k})$ , w.h.p. Hence, the total number of edges in the spanner is  $O(kn^{1+1/k} \log n)$ , w.h.p.<sup>4</sup>.

**Derandomization – Abstracting the Properties of the Random Selection.** The only part of this algorithm that relies on randomness is the step of selecting active clusters. As can be seen in the analysis, it suffices that this (random) selection satisfies the following two properties, per level: (1) nodes that have more than  $d = \Theta(n^{1/k} \log n)$  neighboring clusters will have at least one active cluster, (2) if the number of clusters in this level is  $R \geq \Theta(n^{1/k} \log n)$ , the number of active clusters is at most  $R \cdot n^{-1/k}$ . The former ensures that the number of edges added per node in a level  $i \in \{1, 2, \dots, k-1\}$  is at most  $\Theta(n^{1/k} \log n)$ . The latter follows from Chernoff bound. Because of having this property in all levels, it follows that the total number of clusters at the last level is  $O(n^{1/k} \log n)$ . Hence, the number of added edges per node in that level is  $O(n^{1/k} \log n)$ .

**Derandomization via Deterministic Hitting Set Computations.** We can formulate the above two properties as a direct instance of the hitting set problem discussed in Definition 8, as follows: We set  $p = n^{1/k}$  and  $T = i + 1 \leq O(\log n)$ . Moreover, we make each node that has at least  $d = \Theta(n^{1/k} \log n)$  neighboring clusters be one node in  $L$  and each cluster center one node in  $R$ . Clearly, each node  $\ell \in L$  can know  $\Theta(p \log n)$  nodes of  $R$  that are within its  $i + 1 \leq k + 1 \leq O(\log n)$  hops, these are the vertices in  $R(\ell)$ . We can also deliver one message from each  $r \in R$  to all vertices  $\ell \in L$  for which  $r \in R(\ell)$  in  $T$  rounds. For that, we simply do a broadcast in the cluster centered at  $r$  and then pass it on to all neighboring nodes including  $\ell$ . These provide all that we need to set up the hitting set problem. Moreover, we also use a  $2T$ -hop  $(d, c)$ -decomposition of graph  $G$ , for  $d = c = 2^{O(\sqrt{\log n \cdot \log \log n})}$ , which can be computed using Theorem 3 in  $2^{O(\sqrt{\log n \cdot \log \log n})}$  rounds. We can now invoke the deterministic hitting set algorithm of Lemma 9, which runs in  $2^{O(\sqrt{\log n \cdot \log \log n})}$  rounds. That provides a subset  $R^* \subseteq R$  with size at most  $R/p = R \cdot n^{-1/k}$  such that each node  $\ell \in L$  has at least one node in  $R^* \cap R(\ell)$ . That is, each node that has more than  $\Theta(n^{1/k} \log n)$  neighboring clusters will have at least one active cluster. These satisfy the two properties abstracted above, thus providing us with a deterministic selection of active clusters in each iteration of Baswana-Sen, hence completing the proof of Theorem 1.

## 5 Minimum Set Cover and Dominating Set

Consider a set cover instance  $(X, \mathcal{S})$  consisting of a set  $X$  of elements and a set  $\mathcal{S} \subseteq 2^X$  of subsets of  $X$  such that  $\bigcup_{A \in \mathcal{S}} A = X$ . The objective of the minimum set cover problem is to select a subset  $\mathcal{C} \subseteq \mathcal{S}$  of the sets in  $\mathcal{S}$  such that  $\bigcup_{A \in \mathcal{C}} A = X$  and such that the cardinality of  $\mathcal{C}$  is minimized. As standard (see e.g., [2]), we model the set cover instance  $(X, \mathcal{S})$  as a distributed graph problem by defining a bipartite network graph that has a node  $u_x$  for each element  $x \in X$  and a node  $v_A$  for each set  $A \in \mathcal{S}$  and that contains an edge  $\{u_x, v_A\}$  whenever  $x \in A$ . We also note that one can solve the distributed minimum dominating set

<sup>4</sup> With slightly more care, one can show that this number is actually  $O(kn^{1+1/k})$ , with high probability.

**Algorithm 1:** Distributed Set Cover Algorithm.

---

```

 $\mathcal{C} := \emptyset$  // start with an empty set cover;
for stage  $i := 1, 2, \dots, \lceil \log n \rceil$  do
  for phase  $j := 1, 2, \dots, \lceil \log n \rceil$  do
    for step  $c := 1, 2, \dots, c(n)$  do
       $\mathcal{S}_{i,c} := \{A \in \mathcal{S} : \delta(A) \geq n/2^i \text{ and } A \text{ is in cluster of color } c\}$ ;
       $X_{i,j,c} := \{x \in X : s(x, c, n/2^i) \geq n/2^j\}$ ;
      Select  $\mathcal{S}' \subseteq \mathcal{S}_{i,c}$  such that;
        a)  $\forall x \in X_{i,j,c} : \exists A \in \mathcal{S}' : x \in A$ ;
        b)  $\forall x \in X : x \text{ uncovered} \implies |\{A \in \mathcal{S}' : x \in A\}| = O(\log n)$ ;
       $\mathcal{C} := \mathcal{C} \cup \mathcal{S}'$  // add  $\mathcal{S}'$  to the set cover

```

---

problem on a graph  $G = (V, E)$  by using a distributed set cover algorithm and applying it to the corresponding set cover instance (where each node  $v \in V$  represents an element and a set and where the set corresponding to a node  $u$  contains  $u$ , as well as all neighbors of  $u$  in  $G$ ). The network graph of the set cover instance for the dominating set problem on  $G$  is given by the bipartite cover of  $G$  and a CONGEST-model algorithm on the bipartite cover of  $G$  can be run on the CONGEST model on  $G$  in the same time.

In the following, we assume that we are given a set cover instance  $(X, \mathcal{S})$  and that  $G = (V_X \cup V_S, E)$  is the bipartite  $n$ -node graph corresponding to the given set cover instance. We further assume that for some  $d(n) \geq 1$  and  $c(n) \geq 1$ , a strong diameter 2-hop  $(d(n), c(n))$ -decomposition of  $G$  is given. Recall that for  $d(n) = c(n) = 2^{O(\sqrt{\log n \log \log n})}$ , such a decomposition can be computed in  $2^{O(\sqrt{\log n \log \log n})}$  rounds on  $G$  (cf. Theorem 3).

We first describe the algorithm to compute a small set cover of  $(X, \mathcal{S})$ . The algorithm can be seen as a distributed variant of the well-known sequential greedy algorithm. The algorithm starts with an empty set cover and it consists of a sequence of steps in which several sets of  $\mathcal{S}$  are added to the set cover in parallel. Throughout the algorithm, we trace some properties of the subproblem that still has to be solved. For every set  $A \in \mathcal{S}$ , we use  $\delta(A)$  to denote the number of uncovered elements of  $A$  (i.e., at the beginning of the algorithm, we have  $\delta(A) = |A|$  and at the end, we need to have  $\delta(A) = 0$ ). Further, for every element  $x \in X$ , for each of the colors  $c \in \{1, \dots, c(n)\}$  of the given 2-hop decomposition of  $G$ , and for some parameter  $d \geq 1$ , we define the degree- $d$ , color- $c$  support  $s(x, c, d)$  of  $x$  as follows. If  $x$  is already covered, we have  $s(x, c, d) = 0$ , otherwise,  $s(x, c, d)$  is defined to be the number of sets  $A \in \mathcal{S}$  such that  $x \in A$ ,  $A$  is in a cluster of color  $c$ , and  $\delta(A) \geq d$ . The algorithm consists of  $\lceil \log n \rceil$  stages  $i = 1, 2, \dots, \lceil \log n \rceil$  and each stage consists of  $\lceil \log n \rceil$  phases  $j = 1, \dots, \lceil \log n \rceil$ . The algorithm guarantees that throughout stage  $i \in \{1, \dots, \lceil \log n \rceil\}$ , for all sets  $A \in \mathcal{S}$ , it holds that  $\delta(A) < n/2^{i-1}$ , i.e., in each stage, the upper bound on the maximum remaining set size is halved. Further, for each stage  $i \in \{1, \dots, \lceil \log n \rceil\}$  and each phase  $j \in \{1, \dots, \lceil \log n \rceil\}$ , it holds that  $s(x, c, n/2^i) < n/2^{j-1}$  for all  $x \in X$  and all  $c \in \{1, \dots, c(n)\}$ . Further, each phase consists of  $c(n)$  steps. The pseudocode of the whole set cover algorithm is given by Algorithm 1.

► **Lemma 11.** For all  $i, j \in \{1, \dots, \lceil \log n \rceil\}$  and all cluster colors  $c \in \{1, \dots, c(n)\}$ , throughout stage  $i$  and phase  $j$  of Algorithm 1, it holds that

- (a) for every  $A \in \mathcal{S}$ , we have  $\delta(A) < n/2^{i-1}$ ,
- (b) for every  $x \in X$ , we have  $s(x, c, n/2^i) < n/2^{j-1}$ ,
- (c) at the end of step  $c$ , for every  $x \in X$ , we have  $s(x, c, n/2^i) < n/2^j$ .

**Proof.** We prove (a)–(c) by induction on  $i$ ,  $j$ , and  $c$ . First, note that (a) holds for  $i = 1$  because the bipartite graph  $G$  representing the set cover instance has  $n$  nodes. Because there needs to be at least one set and at least one element in every set cover instance, we thus have  $|\mathcal{S}| < n$  and  $|X| < n$ . Further note that if (a) is true for some stage  $i$ , then (b) holds for the given stage  $i$  and  $j = 1$  for the same reason. Also note that (b) and (c) always hold for all covered elements  $x$  because in this case we defined  $s(x, c, n/2^i)$  to be 0.

We next prove that (b) implies (c). Step  $c$  of stage  $i$  and phase  $j$  guarantees that for each element  $x \in X_{i,j,c}$  (i.e., for each element for which  $s(x, c, n/2^i) \geq n/2^j$ ), there is a set  $A \in \mathcal{S}'$  such that  $x \in A$ . Consequently  $x$  is covered after the step and thus  $s(x, c, n/2^i) = 0$ . By condition (c), after all the  $c(n)$  steps of stage  $i$  and phase  $j$ , we have  $s(x, c, n/2^i) < n/2^j$  and thus if  $j < \lceil \log n \rceil$ , condition (b) also holds for stage  $i$  and phase  $j + 1$ . To also prove the induction step for condition (a), consider the end of phase  $j = \lceil \log n \rceil$  of stage  $i$ . By (c), we have  $s(x, c, n/2^i) < n/2^{\lceil \log n \rceil} \leq 1$  and thus  $s(x, c, n/2^i) = 0$  for all  $x \in X$  and all  $c \in \{1, \dots, c(n)\}$ . This implies that there is no set  $A \in \mathcal{S}$  left with  $\delta(A) \geq n/2^i$ . ◀

► **Lemma 12.** *Given a strong diameter 2-hop  $(d(n), c(n))$ -decomposition of  $G$ , Algorithm 1 can be implemented deterministically in  $\tilde{O}(d(n) \cdot c(n))$  rounds in the CONGEST model on  $G$ .*

**Proof.** The algorithm consists of  $O(\log n)$  stages,  $O(\log n)$  phases per stage, and  $c(n)$  steps per phase. The total number of steps is therefore  $O(c(n) \log^2 n) = \tilde{O}(c(n))$ . To prove the claim of the lemma, we thus need to show that each step can be implemented in  $\tilde{O}(d(n))$  rounds in the CONGEST model on  $G$ . Consider some stage  $i$ , some phase  $j$ , and some step  $c$  in stage  $i$  and phase  $j$ . Recall that  $\mathcal{S}_{i,c} \subseteq \mathcal{S}$  contains all sets  $A \in \mathcal{S}$  that are in clusters of color  $c$  of the given network decomposition and for which  $\delta(A) \geq n/2^i$  at the beginning of step  $c$  of phase  $j$  of stage  $i$ . Let  $X_{i,c}$  be the set of uncovered elements of the sets in  $\mathcal{S}_{i,c}$ . Consider the subgraph  $G_{i,c}$  of the set cover graph  $G$  that is induced by nodes corresponding to the elements in  $X_{i,c}$  and the sets in  $\mathcal{S}_{i,c}$ . Note that for some element  $x \in X_{i,c}$ ,  $s(x, c, n/2^i)$  is the degree of the corresponding node in  $G_{i,c}$ . The algorithm needs to select a subset  $\mathcal{S}'$  of the sets in  $\mathcal{S}_{i,c}$  such that for each  $x \in X_{i,c}$ , the number of selected sets containing  $x$  is at most  $O(\log n)$  and for each  $x \in X_{i,j,c}$ , there is at least 1 set containing  $x$  selected. On the graph  $G_{i,c}$ , this translates into selecting a subset of the nodes  $v_A$  corresponding to the sets  $A \in \mathcal{S}_{i,c}$  such that for each  $x \in X_{i,c}$ , the corresponding node  $u_x$  has at most  $O(\log n)$  neighbors selected and if  $u_x$  has degree at least  $n/2^j$ , it has at least 1 neighbor selected. From Lemma 11, we further know that all nodes  $u_x$  in  $G_{i,c}$  have degree at most  $n/2^{j-1}$  and all nodes  $v_A$  have degree at most  $n/2^{i-1}$ . Selecting the subset of sets  $\mathcal{S}'$  therefore exactly corresponds to the solving the problem given by Lemma 10 on graph  $G_{i,c}$  with parameter  $p = n/(\gamma 2^j \log n)$  for an appropriate constant  $\gamma > 0$ . Further note that because we are given a 2-hop  $(d(n), c(n))$ -decomposition, the parts of the graph  $G_{i,c}$  corresponding to different clusters of color  $c$  are disjoint. We can therefore solve the problem of selecting nodes in  $\mathcal{S}_{i,c}$  separately for each cluster of color  $c$ . Because each such cluster has a spanning tree of diameter  $d(n)$ , Lemma 10 implies that each step can be implemented in  $\tilde{O}(d(n))$  rounds. ◀

► **Lemma 13.** *Algorithm 1 computes a solution for a given set cover instance that is with an  $O(\log^2 n)$ -factor of an optimal solution.*

**Proof.** The algorithm always computes a valid solution (i.e., a solution that covers all the elements): For  $i = j = \lceil \log n \rceil$ , condition (c) of Lemma 11 implies that  $s(x, c, 1) = 0$  for all  $x \in X$  and all  $c \in \{1, \dots, c(n)\}$ . This can only be true if all elements  $x \in X$  are covered.

To prove the bound on the approximation ratio, we use a standard *dual fitting* argument (see e.g. [31, Chapter 13]). In the step that covers an element  $x \in X$ , we assign a dual variable  $y_x > 0$  to  $x$  such that at the end of the algorithm  $\sum_{x \in X} y_x = |\mathcal{C}|$ . Consider some step  $c$  of stage  $i$  and phase  $j$  and assume that the sets in  $\mathcal{S}'$  are added to the set cover  $\mathcal{C}$ . Let  $X' \subseteq X$  be the set of elements that were uncovered before step  $c$  of stage  $i$  and phase  $j$  and which are covered by the sets in  $\mathcal{S}'$ . For all  $x \in X'$ , we set the dual variable  $y_x$  to  $y_x := |\mathcal{S}'|/|X'|$ . This clearly implies that at the end  $\sum_{x \in X'} y_x = |\mathcal{S}'|$  and thus at the end  $\sum_{x \in X} y_x = |\mathcal{C}|$ . Note that for all sets  $A \in \mathcal{S}'$ , we have  $\delta(A) \geq n/2^i$ . Because for each uncovered element  $x \in X$ , there are at most  $O(\log n)$  sets  $A \in \mathcal{S}'$  for which  $x \in A$ , we have  $|X| = \Omega(|\mathcal{S}'| \cdot n/(2^i \log n))$ . Because by condition (a) of Lemma 11 for all  $A \in \mathcal{S}$ , we have  $\delta(A) \leq n/2^{i-1}$ , for all  $x \in X' \cap A$ , we have  $y_x = O(\log n)/\delta(A)$ . At the end of the algorithm, we thus get that for every set  $A \in \mathcal{S}$ ,

$$\sum_{x \in A} y_x = O(\log n) \cdot \sum_{\ell=1}^{|A|} \frac{1}{\ell} = O(\log^2 n).$$

Dividing all  $y_x$ -variables by  $O(\log^2 n)$  gives a feasible solution to the dual LP of the standard set cover LP relaxation. By LP duality, the obtained set cover is within an  $O(\log^2 n)$  factor of the optimal solution. ◀

► **Theorem 14.** *A  $O(\log^2 n)$ -approximation for the distributed set cover problem can be computed deterministically in  $2^{O(\sqrt{\log n \cdot \log \log n})}$  rounds in the CONGEST model.*

**Proof.** We can compute a 2-hop  $(2^{O(\sqrt{\log n \cdot \log \log n})}, 2^{O(\sqrt{\log n \cdot \log \log n})})$ -decomposition deterministically in  $2^{O(\sqrt{\log n \cdot \log \log n})}$  rounds in the CONGEST model, using Theorem 3. Having this, the theorem then directly follows from Lemmas 13 and 12. ◀

---

## References

- 1 Noga Alon and Joel H Spencer. *The probabilistic method*. John Wiley & Sons, 2004.
- 2 Matti Åstrand and Jukka Suomela. Fast distributed approximation algorithms for vertex cover and set cover in anonymous networks. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, pages 294–302. ACM, 2010.
- 3 B. Awerbuch, AV Goldberg, M. Luby, and S. Plotkin. Network decomposition and locality in distributed computation. In *FOCS*, pages 364–369, 1989.
- 4 Baruch Awerbuch. Complexity of network synchronization. *Journal of the ACM (JACM)*, 32(4):804–823, 1985.
- 5 Baruch Awerbuch and David Peleg. Sparse partitions. In *Proc. IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 503–513, 1990.
- 6 Leonid Barenboim and Michael Elkin. Distributed graph coloring: Fundamentals and recent developments. *Synthesis Lectures on Distributed Computing Theory*, 4(1):1–171, 2013.
- 7 Leonid Barenboim, Michael Elkin, and Cyril Gavoille. A fast network-decomposition algorithm and its applications to constant-time distributed computation. *Theoretical Computer Science*, 2016.
- 8 Surender Baswana and Sandeep Sen. A simple and linear time randomized algorithm for computing sparse spanners in weighted graphs. *Random Structures & Algorithms*, 30(4):532–563, 2007.

- 9 Keren Censor-Hillel, Merav Parter, and Gregory Schwartzman. Derandomizing local distributed algorithms under bandwidth restrictions. In *31 International Symposium on Distributed Computing*, 2017.
- 10 Bilel Derbel and Cyril Gavoille. Fast deterministic distributed algorithms for sparse spanners. In *International Colloquium on Structural Information and Communication Complexity*, pages 100–114. Springer, 2006.
- 11 Bilel Derbel, Cyril Gavoille, and David Peleg. Deterministic distributed construction of linear stretch spanners in polylogarithmic time. In *International Symposium on Distributed Computing*, pages 179–192. Springer, 2007.
- 12 Bilel Derbel, Cyril Gavoille, David Peleg, and Laurent Viennot. On the locality of distributed sparse spanner construction. In *Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*, pages 273–282. ACM, 2008.
- 13 Bilel Derbel, Mohamed Mosbah, and Akka Zemmari. Sublinear fully distributed partition with applications. *Theory of Computing Systems*, 47(2):368–404, 2010.
- 14 Paul Erdős. Some problems in graph theory. In *STUDIA SIC MATH. HUNGAR.* Citeseer, 1966.
- 15 M. Ghaffari and F. Kuhn. Derandomizing distributed algorithms with small messages: Spanners and dominating set. Technical Report 285, U. of Freiburg, Dept. of Computer Science, 2018. URL: <http://tr.informatik.uni-freiburg.de/reports/report285/report00285.pdf>.
- 16 Mohsen Ghaffari. An improved distributed algorithm for maximal independent set. In *Pro. ACM-SIAM Symp. on Discrete Algorithms (SODA)*, 2016.
- 17 Mohsen Ghaffari, David G Harris, and Fabian Kuhn. On derandomizing local distributed algorithms. *arXiv preprint arXiv:1711.02194*, 2017.
- 18 Ofer Grossman and Merav Parter. Improved deterministic distributed construction of spanners. In *31 International Symposium on Distributed Computing*, 2017.
- 19 Lujun Jia, Rajmohan Rajaraman, and Torsten Suel. An efficient distributed algorithm for constructing small dominating sets. *Distributed Computing*, 15(4):193–205, 2002.
- 20 Ken-Ichi Kawarabayashi and Gregory Schwartzman. Adapting local sequential algorithms to the distributed setting. *arXiv preprint arXiv:1711.10155*, 2017.
- 21 Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. Local computation: Lower and upper bounds. *J. ACM*, 63(2):17:1–17:44, mar 2016.
- 22 Fabian Kuhn and Roger Wattenhofer. Constant-time distributed dominating set approximation. In *Proc. ACM Symp. on Principles of Distributed Computing (PODC)*, pages 25–32, 2003.
- 23 Christoph Lenzen and Roger Wattenhofer. Minimum dominating set approximation in graphs of bounded arboricity. In *International Symposium on Distributed Computing*, pages 510–524. Springer, 2010.
- 24 Nathan Linial. Distributive graph algorithms global solutions from local data. In *Proc. IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 331–335. IEEE, 1987.
- 25 Michael Luby. Removing randomness in parallel computation without a processor penalty. *Journal of Computer and System Sciences*, 47(2):250–286, 1993.
- 26 Moni Naor and Larry Stockmeyer. What can be computed locally? *SIAM Journal on Computing*, 24(6):1259–1277, 1995.
- 27 Alessandro Panconesi and Aravind Srinivasan. Improved distributed algorithms for coloring and network decomposition problems. In *Proc. ACM Symp. on Theory of Computing (STOC)*, pages 581–592. ACM, 1992.
- 28 David Peleg. *Distributed Computing: A Locality-sensitive Approach*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

- 29 David Peleg and Jeffrey D Ullman. An optimal synchronizer for the hypercube. *SIAM Journal on computing*, 18(4):740–747, 1989.
- 30 Jeanette P Schmidt, Alan Siegel, and Aravind Srinivasan. Chernoff-hoeffding bounds for applications with limited independence. *SIAM J. on Discrete Math.*, 8(2):223–250, 1995.
- 31 Vijay V Vazirani. *Approximation algorithms*. Springer Science & Business Media, 2013.





# Distributed MST and Broadcast with Fewer Messages, and Faster Gossiping

**Mohsen Ghaffari**

ETH Zurich, Switzerland  
ghaffari@inf.ethz.ch

**Fabian Kuhn**

University of Freiburg, Germany  
kuhn@cs.uni-freiburg.de

---

## Abstract

We present a distributed minimum spanning tree algorithm with near-optimal round complexity of  $\tilde{O}(D + \sqrt{n})$  and message complexity  $\tilde{O}(\min\{n^{3/2}, m\})$ . This is the first algorithm with sub-linear message complexity and near-optimal round complexity and it improves over the recent algorithms of Elkin [PODC'17] and Pandurangan et al. [STOC'17], which have the same round complexity but message complexity  $\tilde{O}(m)$ . Our method also gives the first broadcast algorithm with  $o(n)$  time complexity – when that is possible at all, i.e., when  $D = o(n)$  – and  $o(m)$  messages. Moreover, our method leads to an  $\tilde{O}(\sqrt{nD})$ -round GOSSIP algorithm with bounded-size messages. This is the first such algorithm with a sublinear round complexity.

**2012 ACM Subject Classification** Theory of computation → Distributed algorithms

**Keywords and phrases** Distributed Algorithms, Minimum Spanning Tree, Round Complexity, Message Complexity, Gossiping, Broadcast

**Digital Object Identifier** 10.4230/LIPIcs.DISC.2018.30

**Acknowledgements** We are grateful to Merav Parter for helpful discussions.

## 1 Introduction

This paper presents a distributed algorithm for computing a Minimum Spanning Tree (MST) with a nearly optimal round complexity and an improved message complexity. Our method also leads to improvements for two other basic problems, namely broadcast and gossiping. Let us start with briefly recalling the CONGEST model, which is the standard synchronous message-passing model of distributed computing with small messages:

**The CONGEST Model [37].** The network is abstracted as a weighted graph  $G = (V, E, w)$  where  $n = |V|$ ,  $m = |E|$ . Moreover, we use  $D$  to denote the diameter of the graph. Initially, each node has a unique  $\Theta(\log n)$ -bit identifier and knows its own edges – i.e., the identifier of the other endpoint of the edges – as well as the weight of these edges. At the end, each node should know its own part of the output, e.g., which of its edges are in the computed minimum spanning tree. Per round, each node can send one  $O(\log n)$ -bit message to each of its neighbors. The round complexity of an algorithm is the number of rounds until all nodes are done with their computation, and the message complexity of the algorithm is the total number of messages sent throughout the algorithm.



© Mohsen Ghaffari and Fabian Kuhn;

licensed under Creative Commons License CC-BY

32nd International Symposium on Distributed Computing (DISC 2018).

Editors: Ulrich Schmid and Josef Widder; Article No. 30; pp. 30:1–30:12

Leibniz International Proceedings in Informatics



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

**History and Significance of MST in Distributed Algorithms.** Minimum Spanning Tree (MST) is one of the central problems in the study of distributed graph algorithms, and it has been studied extensively since 1980s [12, 6, 11, 2, 13, 30, 39, 38, 8, 10, 26, 7, 28, 27, 16, 18, 36, 9, 32]. One can argue that much of the developments in the CONGEST model of distributed computing have been centered around (MST); the algorithmic or impossibility techniques developed for MST led to important results for other fundamental graph problems.

The work on distributed MST started with the algorithm of Gallager, Humblet, and Spira [12], which has time complexity  $O(n \log n)$ , and can be seen as a variant of the 1926 algorithm of Boruvka [35]. The time complexity was gradually improved [6, 11], eventually leading to the “optimal”  $O(n)$ -round algorithm of Awerbuch [2]. At the time, a round complexity of  $O(n)$  was regarded as being optimal, because there are graphs of diameter  $D = \Omega(n)$  on which one cannot do better (e.g., the  $n$ -node cycle).

A pioneering work of Garay, Kutten and Peleg [30, 13] shifted the are toward sublinear time algorithms in graphs of sublinear diameter, i.e., where this excuse of graphs with  $D = \Omega(n)$  is ruled out. In particular, Garay et al. [13] presented an  $O(D + n^{0.61})$ -round MST algorithm, which was subsequently improved by Kutten and Peleg [30] to  $O(D + \sqrt{n} \log^* n)$ . Shortly after, Rubinfeld and Peleg [38] proved a lower bound of  $\Omega(D + \sqrt{n})$  for the round complexity of any distributed MST algorithm. By now, we understand that a wide range of other fundamental graph problems can be solved or approximated in time (close to)  $\tilde{O}(D + \sqrt{n})$ : the list includes minimum cut [14, 34], single-source shortest path [31, 33, 24, 4], tree embedding [19], maximum  $s$ - $t$  flow [17], and minimum connected dominating set [15]. Moreover, this time complexity is optimal for essentially all of these problems [7]. Many of these results build on the methods and results developed initially for MST.

**Message Complexity Comes Back.** In the above line of work, the primary focus has been the round complexity. However, now that we understand the round complexity aspect of MST rather well, there has been a revived interest in getting algorithms with improved message complexity. The time-optimal  $\tilde{O}(D + \sqrt{n})$ -round algorithm of Kutten and Peleg has message complexity  $\tilde{O}(m + n^{3/2})$ . In a recent work, Pandurangan et al. [36] provided the first (randomized) MST algorithm with round complexity  $\tilde{O}(D + \sqrt{n})$  and message complexity  $\tilde{O}(m)$ . Shortly after, Elkin [9] presented a simpler deterministic algorithm with similar time and message complexities (in fact with improvements in the logarithmic factors).

By a result of Awerbuch et al. [3], this  $\tilde{O}(m)$  message complexity is known to be optimal for deterministic algorithms. Moreover, it is the best possible if either we restrict ourselves to randomized *comparison-based* algorithms [3] or to the variant of the distributed model where at the beginning, each node does not know its neighbors [3, 29] (this is sometimes called the KT0 variant of the model, as opposed to the KT1 version where at the beginning nodes know their neighbors). However, if at the beginning, each node knows all its neighbors and we are allowed to use general randomized algorithms, this  $\tilde{\Omega}(m)$  message complexity lower bound does not apply. In fact, a beautiful result of King, Kutten, and Thorup [27] presents an algorithm with message complexity  $\tilde{O}(n)$ , however at the expense of having time complexity  $\tilde{O}(n)$ .

**Open Question.** This state of the art exhibits one clear open question:

Can we find a time-optimal MST algorithm with message complexity  $o(m)$ ?

## 1.1 Our Results

**MST.** Our primary result in this paper is to provide a positive answer to the above open question. In particular, we prove that

► **Theorem 1.** *There is a distributed randomized MST algorithm with round complexity  $\tilde{O}(D + \sqrt{n})$  and message complexity  $\tilde{O}(\min\{n^{3/2}, m\})$ .*

**Broadcast.** Along the way to this MST algorithm, we find a single-message broadcast algorithm – which can deliver a message from any source to all nodes – with an improved message complexity. We find this to be important on its own, given the centrality of the broadcast problem throughout distributed computing.

► **Theorem 2.** *There is a distributed randomized algorithm that broadcasts one message to all nodes in  $\tilde{O}(D + \sqrt{n})$  using  $\tilde{O}(\min\{n^{3/2}, m\})$  messages.*

We note that prior to this result, all known broadcast algorithms needed to use at least  $\Omega(m)$  messages (in which case a simple flooding delivers the message to all in  $O(D)$  rounds), with only one exception: the only known broadcast algorithm with message complexity  $o(m)$  would need to use  $\Omega(n)$  rounds [27].

**Gossiping.** The method that we develop for a message-efficient MST algorithm has one more significant consequence: it leads to the first sublinear-time GOSSIP algorithm in general graphs with bounded size messages [21, 5, 22]. In this problem, initially one node knows an  $O(\log n)$ -bit message, which should be delivered to all nodes. Per round each node can either PUSH an  $O(\log n)$ -bit message to an arbitrary neighbor or PULL an  $O(\log n)$ -bit message from an arbitrary neighbor.

► **Theorem 3.** *There is a distributed randomized algorithm that broadcasts one message to all nodes in  $\tilde{O}(\sqrt{nD})$  rounds of the GOSSIP model.*

## 2 Preliminaries

In this section, we introduce two tools that will be used throughout our algorithms: one is an adaptation of the well-known 1926 algorithm of Boruvka for computing a minimum spanning tree [35], and the other is a *randomized linear sketching* tool.

**Boruvka’s algorithm.** The algorithm is made of  $\Theta(\log n)$  phases, where we gradually merge fragments (i.e., subtrees) of the MST with each other until we have exactly one fragment, i.e., the MST. Initially, each node is its own fragment. Per phase, we do as follows: Each fragment  $C$  picks the lightest edge that connects  $C$  to nodes outside  $C$ . It is well-known and easy to see that all these lightest edges, one per fragment, belong to the MST. We will add only some of these edges to the MST, in a way that ensures that the corresponding fragment merges have a small depth (in a sense that will become clear soon); this allows us to perform these merges efficiently. In particular, each fragment throws a Head/Tails fair coin. Then, for each edge  $e = (v, u)$  which is the lightest edge of the fragment  $C \ni v$  who connects to fragment  $C' \ni u$ , we *accept* edge  $e$  for a merge if and only if  $C$  has a Tail coin toss and  $C'$  has a Head coin toss. The accepted merge edges get added to the output MST. As a result, we perform a merge along each accepted merge edge by unifying the fragments connected via accepted merge edges, all at the same time. Notice that each new fragment is formed by merging a number of previous fragments in a star shape. That is, the new fragment is made

of a number Tail old fragments which merge with one central Head old fragment. Then, we proceed to the next phase. It is known that after  $\Theta(\log n)$  phases, with high probability, we have exactly one fragment which is the final MST. See e.g. [16] for the correctness proof of this method.

We also note that the algorithm can be extended directly to disconnected graphs, in which case it computes a minimum spanning tree in each connected component. Thus, overall, it provides a maximal forest with minimum weight.

**Linear Sketching.** The linear sketching tool that we will use was first used by Ahn et al. [1] in streaming algorithms. By now, variants of it have been used in various distributed algorithms [23, 27, 20, 25]. In a rough sense, this tool creates a “sketch” of a large set of elements (particularly, edges in our applications) which has only a few bits, and such that out of these few bits, we can still decode one element of the set. Since the sketch is linear (concretely, a bit-wise XOR), adding the sketches of two sets gives a linear sketch of their symmetric set difference. This last property will be important for our application. We can abstract this sketching as the following lemma.

► **Lemma 4** (Linear Sketching). *Consider a set  $E$  of elements, each with a unique  $\Theta(\log n)$  bit identifier. There is a family  $\mathcal{F}$  of encoder-decoder function pairs, where  $|F| = 2^{\Theta(\log^3 n)}$ , with the following properties: Here, in each pair in  $\mathcal{F}$ , the encoder is a function  $ENC : E \rightarrow \{0, 1\}^{\Theta(\log^3 n)}$  and the decoder is a function  $DEC : \{0, 1\}^{\Theta(\log^3 n)} \rightarrow E$ . The family  $\mathcal{F}$  is such that if we pick one pair of encoder function  $ENC$  and the corresponding decoder function  $DEC$  at random from  $\mathcal{F}$ , then for each set  $S \in E$ , we have  $DEC(\oplus_{e \in S} ENC(e)) \in S$ , with probability at least  $1 - 1/n^5$ .*

### 3 Minimum Spanning Tree

In this section, we prove the following result, which is a more detailed version of Theorem 1 and provides a time-optimal MST algorithm, up to logarithmic factors, with an improved message complexity.

► **Theorem 5** (Sublinear Messages & Near-Optimal Time). *There is a randomized distributed algorithm that in any  $n$ -node  $m$ -edge weighted graph  $G = (V, E, w)$  with diameter  $D$  computes an MST with round complexity  $\tilde{O}(D + \sqrt{n})$  and message complexity  $\tilde{O}(\min\{n^{3/2}, m\})$ , with high probability.*

**Roadmap.** The algorithm for computing this MST is made of two parts. We first compute a sparser subgraph which has at most  $\tilde{O}(n^{3/2})$  edges and still the diameter of it does not exceed  $\tilde{O}(D + \sqrt{n})$ . Moreover, this subgraph is such that we are able to compute it in  $\tilde{O}(D + \sqrt{n})$  rounds and using  $\tilde{O}(n^{3/2})$  messages. Then, we can easily compute a breadth first tree  $T$  of this sparse subgraph, which has depth at most  $\tilde{O}(D + \sqrt{n})$ , in  $\tilde{O}(D + \sqrt{n})$  rounds and using  $\tilde{O}(\min\{n^{3/2}, m\})$  messages. This part is captured by Lemma 6. This lemma itself has the direct corollary for the broadcast problem, it proves Theorem 2, showing that we can perform a global broadcast in  $\tilde{O}(D + \sqrt{n})$  rounds and using  $\tilde{O}(n^{3/2})$  messages. To finish the proof of Theorem 5, we then explain how to use the low-depth tree  $T$  constructed by Lemma 6 to compute the MST, in  $\tilde{O}(D + \sqrt{n})$  rounds and using  $\tilde{O}(n)$  messages. This second part is captured by Lemma 7. Putting Lemma 6 and Lemma 7 together proves Theorem 5.

Finally, we comment that one can generalize Lemma 7 and prove that for every  $\epsilon \in [0, 1/2]$ , there exists a randomized algorithm for constructing MST using  $\tilde{O}(D + n^{1-\epsilon})$  rounds and  $\tilde{O}(\min\{n^{1+\epsilon}, m\})$  messages.

► **Lemma 6** (Sparsification Lemma). *There is a distributed algorithm that in any  $n$ -node  $m$ -edge weighted graph  $G = (V, E, w)$  with diameter  $D$ , computes a spanning subgraph that has diameter  $\tilde{O}(D + \sqrt{n})$  and  $\tilde{O}(n^{3/2})$  edges, in  $\tilde{O}(D + \sqrt{n})$  rounds and using  $\tilde{O}(\min\{n^{3/2}, m\})$  messages. Then, we can also compute a spanning tree of diameter  $\tilde{O}(D + \sqrt{n})$ , in  $\tilde{O}(D + \sqrt{n})$  rounds and using  $\tilde{O}(\min\{n^{3/2}, m\})$  messages.*

**Proof.** Before describing the algorithm, we comment that the algorithm is described with focus on graphs that have  $\Omega(n^{3/2})$  edges; for these, we show that our algorithm will use  $\tilde{O}(n^{3/2})$  messages. The algorithm is such that it passes at most  $\text{poly}(\log n)$  messages through each edge and therefore, it automatically has message complexity at most  $\tilde{O}(m)$ . Hence, we can write the message complexity as  $\tilde{O}(\min\{n^{3/2}, m\})$ .

**Heavy and Light Nodes and the Heavy Subgraph  $G'$ .** Call a node *heavy* if its degree exceeds  $\sqrt{n}$  and *light* otherwise. Let  $G'$  be the subgraph of  $G$  induced by heavy vertices, i.e., the subgraph made of all edges that both of their endpoints are heavy. We call this the Heavy Subgraph. Notice that  $(G \setminus G')$  has at most  $n^{3/2}$  edges.

**Step 1 – Forming Stars.** Define  $S$  to be a random subset of all vertices where each node is included in  $S$  with probability  $\frac{10 \log n}{\sqrt{n}}$ . We add all these vertices and their edges to the so-called heavy subgraph  $G'$ . It is easy to see that, w.h.p.,  $|S| = O(\sqrt{n} \log n)$  and moreover, each heavy node has at least one neighbor in  $S$ . Make each node in  $S$  send a message to each of its neighbors. This uses  $O(n^{3/2} \log n)$  messages overall. Then, make each heavy node  $v \notin S$  that receives a message from a neighbor in  $S$  pick one such neighbor as its center. For each node  $s \in S$ , this defines a *star* subgraph centered at  $s$  along with some of its neighbors. These stars are disjoint for different centers and they satisfy the following guarantees: We have  $O(n^{3/2} \log n)$  stars and each heavy node is in exactly one star.

**Step 2 – Boruvka on the Heavy Subgraph, and Starting From the Stars.** Now, we run Boruvka's algorithm – as explained in Section 2 – for computing a certain maximal spanning forest  $F$  of  $G'$ , with the initial configuration that each star is one fragment of  $F$ . We remark that here we can ignore the edge weights as we are interested in simply one maximal spanning forest, with no regard for the weights. Hence, per phase, for each component, it suffices to find one outgoing edge for each fragment  $C$  of  $F$ . Notice that the naive way for detecting whether an edge  $e = u, v$  incident on a node  $v \in C$  is outgoing or not, i.e., whether  $u \in C$  or not, would require communicating through this edge  $e$ , which overall can lead to  $O(m)$  message complexity. To circumvent that, we follow a *linear sketching* method of King, Kutten, and Thorup [27]. The key aspect of this linear sketching is abstracted by Lemma 4.

We make the center of fragment  $C$  pick one sketch (i.e., a pair of encoder and decoder functions) from the family  $\mathcal{F}$  of Lemma 4 at random and we deliver the  $\Theta(\log^3 n)$  bits of the description of  $\mathcal{F}$  to all nodes of  $C$ , via a simple broadcast along the tree of fragment  $C$ . Then, each node  $v$  computes  $ENC(e)$  for each edge  $e$  incident on  $v$ . Then, we aggregate a bit-wise XOR of these values  $ENC(e)$ , through a convergecast, at the root of the fragment. Notice that for each internal edge  $e = u, v$  which has both  $u$  and  $v$  in fragment  $C$ , the related encoding  $ENC(e)$  gets added to the computed bit-wise XOR twice, once from each endpoint. Hence, it gets canceled out. On the other hand,  $ENC(e)$  for each outgoing edge  $e$  is added exactly once to the bit-wise XOR and therefore, it does not get canceled out. By Lemma 4, once the fragment center receives this bit-wise XOR, that is  $\bigoplus_{e \in O} ENC(e)$  where  $O$  denotes the set of edges going out of fragment  $C$ , it can decode it using the function  $DEC()$ . Thus,

the fragment center then obtains one outgoing edge  $e \in O$ . We then broadcast this outgoing edge to all vertices of the fragment. In particular the endpoint of  $e$  in  $C$  knows that its edge  $e$  is chosen as a potential edge for a merge.

The above process explains how we compute one outgoing edge for each fragment (unless the fragment already fully spans its component and no edge can be added to it, in which case the sketch shows an empty set of outgoing edges). Moreover, it uses  $\tilde{O}(n)$  messages overall because we are simply performing  $O(1)$  convergecasts and broadcasts in the tree of each fragment, each of them carrying  $\text{poly}(\log n)$  bit messages, and all the trees together have at most  $n - 1$  edges. Furthermore, this procedure has round complexity equal to the maximum fragment diameter, up to logarithmic factors. Since we have  $\tilde{O}(n^{1/2})$  stars and each edge of  $G'$  is incident on one of the stars, and because in each fragment all nodes of each star also have all of their star edges as a part of the fragment, the maximum fragment diameter is  $\tilde{O}(n^{1/2})$ . This completes the process of selecting outgoing edges.

Then, selecting the accepted outgoing edges for merge, using the related Head/Tail coins of the fragments, and performing the corresponding merges can be done similarly with simple broadcasts and convergecasts, same as described in the description of Boruvka's algorithm in Section 2. This again takes  $\tilde{O}(n^{1/2})$  rounds and  $\tilde{O}(n)$  messages. After going through all the  $\Theta(\log n)$  phases of Boruvka, we get a maximal spanning forest  $F$  of  $G'$ . throughout step 2, we have used  $\tilde{O}(n)$  messages and  $\tilde{O}(n^{1/2})$  rounds.

**Arguing that  $(G \setminus G') \cup F$  is our desired sparse low-diameter spanning subgraph.** Consider the spanning subgraph  $H$  with all edges of  $(G \setminus G') \cup F$ . This subgraph clearly has at most  $\tilde{O}(n^{3/2})$  edges. We next argue that it has diameter at most  $\tilde{O}(D + \sqrt{n})$ . Consider two arbitrary nodes  $v, u \in G$ . We prove that there is a path of length at most  $\tilde{O}(D + \sqrt{n})$  in  $H$  between  $u$  and  $v$ . Suppose that in  $G$ , we contract each connected component of  $F$  into a single node, and call the resulting graph  $H'$ . There must still be a path of length at most  $D$  between  $u$  and  $v$  in this contracted graph  $H'$ . Let  $P_{u,v}$  be the shortest path in  $H'$  between  $u$  and  $v$ . Now we can transform  $P_{u,v}$  to a path of length  $D + O(\sqrt{n} \log n)$  in  $H$  as follows: any contracted part of the path  $P_{u,v}$  can be “un-contracted” – essentially undoing the process of contraction – and replaced with the portion of the maximal forest  $F$  that spans that component of  $F$ . This step increases the length of the path by at most a constant factor of the number of stars in that connected component of  $F$ . Since the total number of stars is  $O(\sqrt{n} \log n)$ , over all the components, this “un-contracting” process increase the path length from  $D$  to at most  $D + O(\sqrt{n} \log n)$ . This path now exists in  $H = (G \setminus G') \cup F$ . Since this argument holds for any two arbitrary nodes  $v, u \in G$ , we get that the spanning subgraph  $H$  has diameter at most  $\tilde{O}(D + \sqrt{n})$ .

**Computing a Low-Diameter Spanning Tree.** Once we have  $H$ , we can also perform a breadth first search (BFS) in  $H$ , which finds a spanning tree (of  $G$ ) that has diameter  $\tilde{O}(D + \sqrt{n})$ . Since  $H$  has at most  $\tilde{O}(\min\{n^{3/2}, m\})$  edges, the construction of this BFS tree uses at most  $\tilde{O}(\min\{n^{3/2}, m\})$  edges, thus proving the second part of the lemma. ◀

We next describe our MST computation method which we shall use with the help of the computed low-diameter spanning tree provided by Lemma 6.

► **Lemma 7 (MST Computation).** *Suppose that we are given a spanning tree with depth  $d = \tilde{O}(D + \sqrt{n})$  of the graph. Then, we can compute the MST in  $\tilde{O}(D + \sqrt{n})$  rounds and using  $\tilde{O}(n)$  messages.*

**Proof Sketch.** We will use Boruvka’s MST algorithm, combined with the linear sketching method of King et al. [27] for finding the lightest outgoing edge of each fragment, as well as an idea of Elkin [9]. Roughly speaking, the latter will allow us to primarily work on low-depth fragments, except for switching to computation on the global spanning tree, at the end. The algorithm is made of two parts, which we explain next.

**Part 1 – MST Growth With Low-Diameter Fragments.** During the first part, we grow a partial forest  $T$  of the MST. This will be done such that at the end, each fragment of  $T$  has diameter in  $[d, 5d]$ . We always make sure not to have any fragment of diameter exceeding  $5d$ . Initially,  $T$  is the trivial forest made of each node as its own fragment. Then, throughout  $\Theta(\log n)$  phases, we merge some fragments with each other similar to Boruvka’s algorithm explained in Section 2, but with some modification: In each phase, we do as follows. Any fragment  $C$  of diameter at most  $d$  is called *active* and each other fragment is called *inactive*. Each inactive fragment  $C$  will not initiate a merge, that is, it will not compute its lightest outgoing edge to propose it to be added to MST. But  $C$  is still receptive to merges, i.e., if an active fragment  $C'$  proposes a merge with  $C$  (and  $C$  has a Head coin toss and  $C'$  has a Tail coin toss), we accept this merge.

Let us discuss how we compute the lightest outgoing edge for each active component. This is done via a binary search through the range of the weights. Each time if the active search range for fragment  $C$  is  $[L, U]$ , we sketch all edges incident on  $C$  with weight in  $[L, \frac{L+U}{2}]$  and deliver this sketch to the root of fragment  $C$  using a convergecast of the XOR of sketches (similar to proof of Lemma 6). If this sketch indicates an empty edge-set, then the binary search zooms into search range  $[\frac{L+U}{2}, U]$ . Otherwise, it zooms into  $[L, \frac{L+U}{2}]$  as the search range. In either case, the upper and lower bound of the search range are then broadcast to all nodes of the fragment  $C$ . After  $O(\log n)$  steps of the binary search, each of which takes  $\tilde{O}(d)$  rounds, we find the lightest outgoing edge of  $C$  and this edge is known to all nodes of  $C$ . Then, the merge operations are similar to Section 2, along edges that go from Tail coin active fragments to Head coin fragments.

Once these merges happen, the diameter of the resulting fragment might exceed  $5d$ . However, it still will be at most  $7d + 2 = O(d)$ . This is because the new fragment is made of a star merge centered at one fragment of diameter at most  $5d$  with a number of side fragments, each of diameter at most  $d$ . We then spend  $\Theta(d)$  rounds to truncated the fragment to diameter at most  $5d$ . In particular, for each fragment that has diameter exceeding  $5d$ , we select some edges of it to be discarded from the current forest such that each remaining fragment has diameter in  $[d, 5d]$ . This can be done in  $O(d)$  rounds as the next paragraph explains.

To cut this component tree into fragments with diameter in  $[d, 5d]$ , we do as follows: first, perform a broadcast on the fragment tree so that each node knows its distance from the root of the fragment. Then, for each node  $v$  with distance an integer multiple of  $(d + 1)$ , the edge of that node to its parent will be discarded from the forest. This ensures that each fragment of the remaining forest has diameter at most  $2d$ . However, an unfortunate side-effect is that there might be some nodes  $v$  whose edge to its parent gets discarded and such that the subtree rooted at  $v$  has a height smaller than  $d$ . We can detect such nodes using a simple convergecast from the leaves toward the root. For every such node  $v$ , we undo the step of discarding the edge connecting  $v$  to its parent and thus effectively we merge back its subtree with its parent fragment. This might increase the depth of its parent fragment but the corresponding diameter cannot exceed  $4d$ .



Now that the fragments are truncated to have diameter in  $[d, 5d]$ , we proceed to the next phase. After  $\Theta(\log n)$  phases, the first part of the algorithm is done. At that point, we have reached a setting where we have a forest  $F$  made of  $O(n/d)$  fragments, each with diameter in  $[d, 5d]$ . Moreover, all edges of this forest belong to the minimum spanning tree. Overall, this first part uses  $\tilde{O}(d)$  rounds and  $\tilde{O}(n)$  messages.

**Part 2 – MST Growth Beyond Low-Diameter Fragments.** Now, the second part of the algorithm starts. Here, we grow the eventual MST  $T$ , but using a different method. Initially, we set  $T = F$ . The forest  $T$  will grow over time while  $F$  is maintained as is, permanently. We again will have  $\Theta(\log n)$  Boruvka-style phases of growing the fragments of  $T$ . At each time, each fragment of  $T$  is made of a number of fragments of  $F$ .

To compute the lightest outgoing edge of each fragment  $t$  of  $T$ , we do the corresponding binary search differently. Suppose the current weight range of binary search for  $t$  is some range  $[L, U]$ , which is known to all nodes of  $t$ . We then make each of the fragments  $f \in F$  which are a part of  $t$  compute its own sketch of edges incident on  $f$  with weight in  $[L, \frac{L+U}{2}]$  using a convergecast inside  $f$ . This results in one  $\Theta(\log^3 n)$ -bit sketch for the whole fragment  $f$ . Then, we broadcast all these sketch, one for each fragment of  $F$ , to the root of the whole graph using the given spanning tree of depth  $d$ . Since each fragment of  $F$  sends one  $\Theta(\log^3 n)$ -bit message,  $F$  has  $O(n/d)$  fragments, and each message travels  $d$  hops to reach the root of the spanning tree, the overall message complexity is  $\tilde{O}(n/d \cdot d) = \tilde{O}(n)$ . Once the root receives all these sketches, it can compute one outgoing edge for each fragment  $t$  of  $T$  (if there was some edge in the corresponding search range and the sketch was not empty). Then, the root can broadcast these edges back to the nodes of  $t$  by simply putting the identifier and weight of that found edge in the message that arrived from each subfragment  $f \in F$  of  $t$  and reversing the schedule of the convergecast, now delivering messages from the root to all nodes. This completes one step of the binary search. After  $O(\log n)$  such steps of the binary search, the root knows the lightest outgoing edge of each fragment  $t$  of  $T$ . Thus, the root can perform one phase of Boruvka-style merges on  $T$ . It then updates  $T$  accordingly, by merging the related fragments, and reports the new fragment IDs to all nodes. This is again done by putting this  $T$ -fragment ID back in the message that came from the corresponding subfragment  $f \in F$  of  $t$  and reversing the schedule of the convergecast, after which a broadcast inside fragment  $f \in F$  delivers it to all nodes of that fragment. This finishes the description for one phase of Boruvka.

Repeating a similar process for  $\Theta(\log n)$  phases finishes the computation of the MST. The round complexity of the process is  $\tilde{O}(d)$ , since each phase is made of  $\text{poly}(\log n)$  many broadcasts and convergecasts, each it trees of diameter  $d$ . Moreover, the message complexity is  $\tilde{O}(n)$ , again because the convergecasts and broadcasts inside fragments of  $F$  clearly use at most  $\tilde{O}(n)$  messages, and the broadcast and convergecasts in the overall spanning tree use  $\tilde{O}(n/d \cdot d) = \tilde{O}(n)$  messages. ◀

## 4 Fast Gossiping with Bounded-Size Messages

In this section, we prove the following results, which is a more detailed version of Theorem 3.

► **Theorem 8.** *There is a distributed algorithm that spreads a rumor from one node to all in  $\tilde{O}(\sqrt{nD})$  rounds of the GOSSIP model, where per round each node initiates one PUSH or PULL contact with one of its neighbors, communicating an  $O(\log n)$  bit message.*



**Proof.** We first find a sparse spanning subgraph and then perform a gossiping broadcast on this subgraph. The sparsity of the subgraph allows us to have a fast algorithm in the GOSSIP model, as it enables each node to contact only a few other nodes. To build the sparse subgraph, the general method is mostly similar to the one used in Lemma 6. However, we now need to adapt the construction method to the GOSSIP model and analyze the resulting round complexity. Here, we explain the changes.

**Step 1 – Forming Stars.** Now, we call a node heavy if its degree exceeds  $\sqrt{n/D}$  and light otherwise. The subgraph  $G'$  made of all edges induced by heavy vertices is called the heavy subgraph. We pick  $10\sqrt{nD} \log n$  nodes at random in expectation to be the set  $S$  of centers of the stars – i.e., we include each node of the graph in  $S$  randomly with probability  $10\sqrt{D/n} \log n$ . To form the stars, each (heavy) node goes through its neighbors and pulls from them, one by one, until finding the first node who is chosen to be a star. Notice that each heavy node will find a star within  $O(\sqrt{n/D})$  pulls, with high probability. Moreover, each contacted node can always respond whether it is star center or not.

**Step 2 – Boruvka on the Heavy Subgraph, and Starting From the Stars.** Having formed these stars, the algorithm continues essentially the same as in Lemma 6 to build a maximal forest  $F$  of the graph  $G'$  made of all edges with at least one heavy endpoint (and the corresponding nodes). There is only one comment in order: We should explain that we can perform the convergecast and broadcasts of Boruvka’s algorithm in the GOSSIP model, because these trees do not necessarily have small degrees. Suppose that we have a rooted tree of depth  $d$  for each fragment of the forest and each node knows its depth in this tree. Then, we can perform a broadcast (sending a single-message from the root to all) or convergecast (e.g., computing one aggregate function such as sum by sending messages from the leaves up towards the root) in this tree in time  $O(d)$  also in the GOSSIP model. In the case of broadcast, in the  $i^{\text{th}}$  round, each node  $v$  at depth  $i$  makes a PULL contact to its parent. A simple induction shows that each node at depth  $i$  receives the message in round  $i$ . A convergecast is similar; in the  $i^{\text{th}}$  round each node  $v$  at depth  $d - i$  makes a PUSH contact to its parent, sending the message of the convergecast to its parent. Again, via a simple induction, we can prove that after  $d$  rounds the root receives the result of the convergecast. Having this broadcast and convergecast, we can build the maximal forest  $F$  of the subgraph  $G'$  similar to Lemma 6. At the end, each component of this maximal forest  $F$  is rooted in one of its nodes and each node of that component knows its depth in the rooted tree of that component. Since we now have  $O(\sqrt{nD} \log n)$  stars, the diameter of each fragment is at most  $O(\sqrt{nD} \log n)$ . Therefore, also the construction of  $F$  finishes in  $O(\sqrt{nD} \log n)$  rounds of the GOSSIP model.

**Gossiping on the Sparse Subgraph.** Now, we use the spanning subgraph defined by  $(G \setminus G') \cup F$  to perform the gossiping of one message to all nodes. For that, we divide time into phases, each made of two rounds. In odd rounds, each informed node – who already has the gossiping message – picks one of its edges at random – if it has any – and pushes the message through that edge and each uninformed node pulls from one of its neighbors chosen at random. In even rounds, each uninformed node pulls from its parent in the forest  $F$  and each informed node pushes to its parent.

We now argue that the message reaches all nodes in  $O(\sqrt{nD} \log n)$  rounds, with high probability. Let  $v$  be the source of the gossiping, i.e., the node that initially holds the gossip message. As in the proof of Lemma 6, we can see that for every vertex  $u$ , there is a

path  $P_{v,u}$  of length  $D + O(\sqrt{nD} \log n)$  in  $(G \setminus G') \cup F$  connecting  $v$  to  $u$ . Let us say edge  $\{w_1, w_2\} \in P_{v,u}$  is in the *waiting mode* during all the rounds that  $w_1$  is the closest informed node on  $P_{v,u}$  to  $u$ . In particular, during these times  $w_1$  is informed and  $w_2$  is not informed. We analyze the time that the edges of  $P_{v,u}$  spend in the waiting mode, in two parts, one for  $(G \setminus G')$  and one for  $F$ . We show that the summation of the time that different edges of  $P_{v,u}$  spend in the waiting mode is at most  $O(\sqrt{nD} \log n)$  rounds. Hence, the message reaches from  $v$  to  $u$  in  $O(\sqrt{nD} \log n)$  rounds. Notice that by definition, per round at most one edge of  $P_{v,u}$  is in the waiting mode.

Similar to the proof of the last part of Lemma 6, we can see that at most  $D$  hops of the path  $P_{v,u}$  are edges of  $(G \setminus G')$  and the rest are edges of  $F$ . Now, because of the odd rounds, whenever a low-degree node  $w_1$  gets informed, each of its neighbors  $w_2$  in the subgraph  $G \setminus G'$  gets informed with high probability in  $O(\sqrt{n/D})$  rounds. Hence, the message travels one hop through  $G \setminus G'$  per  $O(\sqrt{n/D})$  rounds. Let us say edge  $\{w_1, w_2\} \in (G \setminus G')$  is in the *waiting mode* during all the rounds that  $w_1$  is informed but  $w_2$  is not informed (or vice versa). Thus, each edge is in the waiting mode at most  $O(\sqrt{n/D})$  rounds, w.h.p. Overall, the time that the message spends waiting to travel through the edges of  $P_{v,u} \cap (G \setminus G')$  is at most  $D \cdot O(\sqrt{n/D})$ . Now, we focus on the waiting time of edges of  $F$ .

Because of the even rounds, when the message reaches some node of a fragment of  $F$ , it gets pushed to the root and then pulled to all the nodes of the fragment, in a number of phases asymptotically equal to the depth of that forest fragment. Notice that this depth is upper bounded asymptotically by the number of stars in that fragment. Hence, the total waiting time of the edges of  $P_{v,u}$  in that fragment is upper bounded asymptotically by the number of stars in that fragment. Since overall the number of stars is at most  $O(\sqrt{nD} \log n)$ , the message spends at most  $O(\sqrt{nD} \log n)$  rounds going through the edges of  $P_{v,u} \cap F$ . Hence, overall the time the message takes to reach from  $v$  to  $u$  is at most  $D \cdot O(\sqrt{n/D}) + O(\sqrt{nD} \log n) = O(\sqrt{nD} \log n)$ . ◀

---

## References

- 1 Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Graph sketches: sparsification, spanners, and subgraphs. In *Proceedings of the 31st symposium on Principles of Database Systems*, pages 5–14. ACM, 2012.
- 2 Baruch Awerbuch. Optimal distributed algorithms for minimum weight spanning tree, counting, leader election, and related problems. In *Proc. of the Symp. on Theory of Comp. (STOC)*, pages 230–240. ACM, 1987.
- 3 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.
- 4 Ruben Becker, Andreas Karrenbauer, Sebastian Krinninger, and Christoph Lenzen. Near-optimal approximate shortest paths and transshipment in distributed and streaming models. In *LIPICs-Leibniz International Proceedings in Informatics*, volume 91. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.
- 5 Keren Censor-Hillel, Bernhard Haeupler, Jonathan Kelner, and Petar Maymounkov. Global computation in a poorly connected world: fast rumor spreading with no dependence on conductance. In *Proceedings of the forty-fourth annual ACM symposium on Theory of computing*, pages 961–970. ACM, 2012.
- 6 F Chin and HF Ting. An almost linear time and  $O(n \log n + e)$  messages distributed algorithm for minimum-weight spanning trees. In *Proc. of the Symp. on Found. of Comp. Sci. (FOCS)*, pages 257–266. IEEE, 1985.

- 7 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.
- 8 Michael Elkin. Unconditional lower bounds on the time-approximation tradeoffs for the distributed minimum spanning tree problem. In *Proc. of the Symp. on Theory of Comp. (STOC)*, pages 331–340, 2004.
- 9 Michael Elkin. A simple deterministic distributed mst algorithm, with near-optimal time and message complexities. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 157–163. ACM, 2017.
- 10 Michalis Faloutsos and Mart Molle. A linear-time optimal-message distributed algorithm for minimum spanning trees. *Distributed Computing*, 17(2):151–170, 2004.
- 11 Eli Gafni. Improvements in the time complexity of two message-optimal election algorithms. In *the Proc. of the Int’l Symp. on Princ. of Dist. Comp. (PODC)*, pages 175–185. ACM, 1985.
- 12 Robert G. Gallager, Pierre A. Humblet, and Philip M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and systems (TOPLAS)*, 5(1):66–77, 1983.
- 13 J.A. Garay, S. Kutten, and D. Peleg. A sub-linear time distributed algorithm for minimum-weight spanning trees. In *Proc. of the Symp. on Found. of Comp. Sci. (FOCS)*, 1993.
- 14 M. Ghaffari and F. Kuhn. Distributed minimum cut approximation. In *Proc. of the Int’l Symp. on Dist. Comp. (DISC)*, pages 1–15, 2013.
- 15 Mohsen Ghaffari. Near-optimal distributed approximation of minimum-weight connected dominating set. In *International Colloquium on Automata, Languages, and Programming*, pages 483–494. Springer, 2014.
- 16 Mohsen Ghaffari and Bernhard Haeupler. Distributed algorithms for planar networks ii: Low-congestion shortcuts, mst, and min-cut. In *Proceedings of the twenty-seventh annual ACM-SIAM symposium on Discrete algorithms*, pages 202–219. SIAM, 2016.
- 17 Mohsen Ghaffari, Andreas Karrenbauer, Fabian Kuhn, Christoph Lenzen, and Boaz Patt-Shamir. Near-optimal distributed maximum flow. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, pages 81–90. ACM, 2015.
- 18 Mohsen Ghaffari, Fabian Kuhn, and Hsin-Hao Su. Distributed mst and routing in almost mixing time. In *the Proc. of the Int’l Symp. on Princ. of Dist. Comp. (PODC)*, pages 131–140. ACM, 2017.
- 19 Mohsen Ghaffari and Christoph Lenzen. Near-optimal distributed tree embedding. In *International Symposium on Distributed Computing*, pages 197–211. Springer, 2014.
- 20 Mohsen Ghaffari and Merav Parter. Mst in log-star rounds of congested clique. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, pages 19–28. ACM, 2016.
- 21 George Giakkoupis. Tight bounds for rumor spreading in graphs of a given conductance. In *Symposium on Theoretical Aspects of Computer Science (STACS2011)*, volume 9, pages 57–68, 2011.
- 22 Bernhard Haeupler. Simple, fast and deterministic gossip and rumor spreading. *Journal of the ACM (JACM)*, 62(6):47, 2015.
- 23 James W. Hegeman, Gopal Pandurangan, Sriram V. Pemmaraju, Vivek B. Sardeshmukh, and Michele Scquizzato. Toward optimal bounds in the congested clique: Graph connectivity and MST. In *the Proc. of the Int’l Symp. on Princ. of Dist. Comp. (PODC)*, pages 91–100. ACM, 2015.
- 24 Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. A deterministic almost-tight distributed algorithm for approximating single-source shortest paths. In *Proceedings*

- of the forty-eighth annual ACM symposium on Theory of Computing, pages 489–498. ACM, 2016.
- 25 Tomasz Jurdziński and Krzysztof Nowicki. Mst in  $o(1)$  rounds of congested clique. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2620–2632. SIAM, 2018.
  - 26 Maleq Khan and Gopal Pandurangan. A fast distributed approximation algorithm for minimum spanning trees. *Distributed Computing*, 20(6):391–402, 2008.
  - 27 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. ACM, 2015.
  - 28 Liah Kor, Amos Korman, and David Peleg. Tight bounds for distributed mst verification. In *Symposium on Theoretical Aspects of Computer Science (STACS2011)*, volume 9, pages 57–68, 2011.
  - 29 Shay Kutten, Gopal Pandurangan, David Peleg, Peter Robinson, and Amitabh Trehan. On the complexity of universal leader election. *Journal of the ACM (JACM)*, 62(1):7, 2015.
  - 30 Shay Kutten and David Peleg. Fast distributed construction of  $k$ -dominating sets and applications. In *the Proc. of the Int'l Symp. on Princ. of Dist. Comp. (PODC)*, pages 238–251, 1995.
  - 31 Christoph Lenzen and Boaz Patt-Shamir. Fast routing table construction using small messages: Extended abstract. In *Proc. of the Symp. on Theory of Comp. (STOC)*, pages 381–390, 2013.
  - 32 Ali Mashreghi and Valerie King. Time-communication trade-offs for minimum spanning tree construction. In *Proceedings of the 18th International Conference on Distributed Computing and Networking*, page 8. ACM, 2017.
  - 33 Danupon Nanongkai. Distributed approximation algorithms for weighted shortest paths. In *Proc. of the Symp. on Theory of Comp. (STOC)*, 2014.
  - 34 Danupon Nanongkai and Hsin-Hao Su. Almost-tight distributed minimum cut algorithms. In *International Symposium on Distributed Computing*, pages 439–453. Springer, 2014.
  - 35 Jaroslav Nešetřil, Eva Milková, and Helena Nešetřilová. Otakar boruvka on minimum spanning tree problem translation of both the 1926 papers, comments, history. *Discrete Mathematics*, 233(1):3–36, 2001.
  - 36 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.
  - 37 David Peleg. *Distributed Computing: A Locality-sensitive Approach*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.
  - 38 David Peleg and Vitaly Rubinfeld. A near-tight lower bound on the time complexity of distributed MST construction. In *Proc. of the Symp. on Found. of Comp. Sci. (FOCS)*, pages 253–, 1999.
  - 39 Gurdeep Singh and Arthur J Bernstein. A highly asynchronous minimum spanning tree protocol. *Distributed Computing*, 8(3):151–161, 1995.

# New Distributed Algorithms in Almost Mixing Time via Transformations from Parallel Algorithms

Mohsen Ghaffari

ETH Zurich, Switzerland

ghaffari@inf.ethz.ch

Jason Li

Carnegie Mellon University, USA

<http://cs.cmu.edu/~jml>

---

## Abstract

We show that many classical optimization problems – such as  $(1 \pm \epsilon)$ -approximate maximum flow, shortest path, and transshipment – can be computed in  $\tau_{\text{mix}}(G) \cdot n^{o(1)}$  rounds of distributed message passing, where  $\tau_{\text{mix}}(G)$  is the mixing time of the network graph  $G$ . This extends the result of Ghaffari et al. [PODC'17], whose main result is a distributed MST algorithm in  $\tau_{\text{mix}}(G) \cdot 2^{O(\sqrt{\log n \log \log n})}$  rounds in the CONGEST model, to a much wider class of optimization problems. For many practical networks of interest, e.g., peer-to-peer or overlay network structures, the mixing time  $\tau_{\text{mix}}(G)$  is small, e.g., polylogarithmic. On these networks, our algorithms bypass the  $\tilde{\Omega}(\sqrt{n} + D)$  lower bound of Das Sarma et al. [STOC'11], which applies for worst-case graphs and applies to all of the above optimization problems. For all of the problems except MST, this is the first distributed algorithm which takes  $o(\sqrt{n})$  rounds on a (nontrivial) restricted class of network graphs.

Towards deriving these improved distributed algorithms, our main contribution is a general transformation that simulates any work-efficient PRAM algorithm running in  $T$  parallel rounds via a distributed algorithm running in  $T \cdot \tau_{\text{mix}}(G) \cdot 2^{O(\sqrt{\log n})}$  rounds. Work- and time-efficient parallel algorithms for all of the aforementioned problems follow by combining the work of Sherman [FOCS'13, SODA'17] and Peng and Spielman [STOC'14]. Thus, simulating these parallel algorithms using our transformation framework produces the desired distributed algorithms.

The core technical component of our transformation is the algorithmic problem of solving *multi-commodity routing* – that is, roughly, routing  $n$  packets each from a given source to a given destination – in random graphs. For this problem, we obtain a new algorithm running in  $2^{O(\sqrt{\log n})}$  rounds, improving on the  $2^{O(\sqrt{\log n \log \log n})}$  round algorithm of Ghaffari, Kuhn, and Su [PODC'17]. As a consequence, for the MST problem in particular, we obtain an improved distributed algorithm running in  $\tau_{\text{mix}}(G) \cdot 2^{O(\sqrt{\log n})}$  rounds.

**2012 ACM Subject Classification** Theory of computation → Distributed algorithms

**Keywords and phrases** Distributed Graph Algorithms, Mixing Time, Random Graphs, Multi-Commodity Routing

**Digital Object Identifier** 10.4230/LIPIcs.DISC.2018.31

## 1 Introduction and Related Work

This paper presents a general method that allows us to transform work-efficient parallel algorithms – formally in the PRAM model – into efficient distributed message-passing algorithms – formally in the CONGEST model – for a wide range of network graphs of practical interest. We believe that this method can be of significance for the following reasons: (1) parallel algorithms have been studied extensively since the late 1970s [11, 14, 30] and



© Mohsen Ghaffari and Jason Li;  
licensed under Creative Commons License CC-BY  
32nd International Symposium on Distributed Computing (DISC 2018).  
Editors: Ulrich Schmid and Josef Widder; Article No. 31; pp. 31:1–31:16  
Leibniz International Proceedings in Informatics



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

there is a vast collection of known parallel algorithms for a variety of problems, and (2) there is a rather active community of research on developing new parallel algorithms. Our transformation opens the road for exporting these algorithms to the distributed setting and bridging the research in these two subareas in a concrete and formal manner. As immediate corollaries, by translating the recent work-efficient parallel algorithms for flow-type problems, we obtain new distributed algorithms for approximate maximum flow, shortest path, and transshipment.

Of course, such a transformation is bound to have some limitations. Due to the reasons that shall be explained soon, such a general transformation would be inefficient in worst-case network graphs. But we show that there are efficient transformations for many graph families of practical interest, and we also exhibit that these transformations entail interesting and non-trivial theoretical aspects.

To explain our transformations, we first recall (informal) descriptions of the two computational models that we discuss, the distributed model and the parallel model. The more detailed model definitions are presented later in Section 3.

**The Distributed Computing Model – CONGEST [27].** The network is abstracted as an  $n$ -node undirected graph  $G = (V, E)$ . There is one processor on each node of the network. At the risk of a slight informality, we use the words processor and node interchangeably. Each node has a unique  $\Theta(\log n)$ -bit identifier. Communication happens in synchronous rounds where per round, each node can send one  $B$ -bit message to each of its neighboring nodes in the network graph  $G$ , where typically one assumes  $B = O(\log n)$ . During each round, each processor can perform unbounded computation with the information that it has at the time. The graph is known in a distributed fashion: each processor knows the edges incident on its own node. In case that the edges are weighted, the weight is known to both endpoints. At the end of the computation, each node should know its own part of the output: e.g., in computing a coloring, each node should know its own color. One can measure the efficiency of an algorithm in the CONGEST model in different ways, such as number of rounds taken, or total number of messages sent among all nodes. In this paper, we only focus on minimizing the number of rounds that an algorithm takes.

**The Parallel Model – PRAM [15, 18].** The system is composed of  $p$  processors, each with a unique ID in  $\{1, 2, \dots, p\}$ , and a shared memory block of  $M$  entries, including an output tape. In every round, each processor can read from or write to any memory entry (Concurrent Read and Concurrent Write, aka, CRCW); if multiple processors write to the same entry, an arbitrary one takes effect.<sup>1</sup> The input is provided in the shared memory cells in a manner that can be addressed easily, e.g., in the case of a graph, the input can be given as an adjacency list where there is one memory cell for the  $j^{\text{th}}$  neighbor of the  $i^{\text{th}}$  node.

**Limitations to General Transformations?** Notice that the two models are intrinsically focused on different issues. The PRAM model is about speeding up computations, via using more processors, and tries to understand how much parallelism can help in reducing time. On the other hand, the distributed model is relevant where the system is by nature made of

---

<sup>1</sup> We can also support parallel algorithms that work under a more powerful model: if multiple processors write to the same memory, then we can take any associative function (min, max, sum) on the words written, and write that result into memory. However, for simplicity, we will work under the arbitrary CRCW model.



autonomous entities, each of which knows a part of the problem. For instance, in computer networks, which were historically the primary motivation for distributed models such as CONGEST, the computers in the network each know a part of the network graph and they cooperate to compute something about it, e.g., variants of shortest paths or routing tables. Here, *locality of the data* and *limited communication bandwidth* are the main challenges. As such, it is arguably unreasonable to seek a general efficient transformation of *any* parallel algorithm to a distributed one in *any* arbitrary network graph. Let us elaborate on this. (1) The PRAM model is not limited by any *locality* – each processor can access any single register – while this is an intrinsic limitation in distributed systems – it can take time proportional to the diameter of the network graph for a processor to be informed of some bit residing in a far away corner of the network graph <sup>2</sup>. (2) Similarly, the network graph may have a small cut, which means transferring information across this cut, i.e., from the processors on one side of the cut to the other side, may take a long time, while this can be done much faster in the PRAM model.

**So What Can We Hope For?** The above discussions and the two concrete points on *locality* and *congestion* (or in other words communication bandwidth) suggest that there may be some hope left: at least in network graphs that satisfy some mild conditions on diameter and cut sizes (or alternatively expansion, conductance, or other forms of formalizing lack of “communication bottlenecks”), we might be able to find some general transformation. Arguably, these actually capture a range of network graphs of practical interest. For instance, overlay and peer-to-peer networks are designed and dynamically maintained over time in a fashion that ensures these good properties.

One way of classifying some such nice graph families is by selecting all graphs whose mixing time for a random walk is relatively small. We define mixing time in Section 1.1.2, but informally, the mixing time of a graph is the number of steps a lazy random walk needs to take so that the distribution of the last vertex of the walk is roughly uniform over all  $n$  vertices. A wide range of the (overlay) networks used in practical distributed applications exhibit a good (e.g. polylogarithmic in  $n$ ) mixing time. This holds for example for the networks in consideration in [2, 3, 23, 25, 20, 24, 26, 33].

A canonical reason for this good mixing time is because many of these overlay networks are formed in a way where each node is connected to  $\Theta(\log n)$  randomly chosen nodes. Indeed, we present our general transformation primarily for such random graphs. We then also explain how to emulate the communication on random graphs atop arbitrary networks with a round-complexity overhead related to the mixing time of the graph, thus enabling us to extend the transformation to general graphs, with a round complexity overhead proportional to the mixing time.

## 1.1 Our Results

Our results build off of those in [13], whose main result is a distributed MST problem running in nearly *mixing time*. We improve upon their results in two dimensions, one technical and one primarily conceptual. The technical contribution is an improved algorithm for the *multicommodity routing* problem in random graphs, which is equivalent to the *permutation routing* problem in [13] up to  $\tilde{O}(1)$  factors. We solve this problem in  $2^{O(\sqrt{\log n})}$  rounds,

---

<sup>2</sup> And that bit may be relevant, as is in global problems such as minimum spanning tree, shortest path, etc.

improving upon the  $2^{O(\sqrt{\log n \log \log n})}$  round algorithm in [13]. Together with the ideas in [13], this immediately improves the distributed MST algorithm from  $\tau_{\text{mix}}(G) \cdot 2^{O(\sqrt{\log n \log \log n})}$  to  $\tau_{\text{mix}}(G) \cdot 2^{O(\sqrt{\log n})}$ .

Our second, more conceptual contribution is in applying the multicommodity routing problem in a more general way. In particular, we use it to develop a framework that transforms work-efficient algorithms in the PRAM model to distributed algorithms. This *transformation* allows us to port the recent work-efficient parallel algorithms [28, 31, 32, 4] for approximate maximum flow, shortest path, and transshipment to run in the CONGEST model, taking  $\tau_{\text{mix}}(G) \cdot n^{o(1)}$  rounds for all three problems.

We first describe our multi-commodity routing result for random graphs, our main technical result and a key component in our transformations. We believe that this multi-commodity routing scheme and the hierarchical graph partitioning underlying it may be of independent interest. We then state our transformation results and overview some of their applications in deriving efficient distributed algorithms for some central graph problems.

### 1.1.1 Multicommodity Routing on Random Graphs

**Random Graph Model.** We work with the following random (multi-)graph model  $G(n, d)$  is as follows: each node  $v \in V$  picks  $d = \Omega(\log n)$  random nodes in  $V$  independently with replacement, called the **outgoing** neighbors of  $v$ . The network graph consists of all edges  $(u, v)$  where  $u$  is an outgoing neighbor of  $v$  or vice versa. For  $d = \Omega(\log n)$ , this model behaves very similarly to the Erdős-Rényi model  $\mathcal{G}(n, d/n)$  [9]; we use our variant for convenience.<sup>3</sup>

**Multicommodity Routing.** Consider a random graph  $G(n, p)$  for  $p = O(\log n)$ , and suppose that we have pairs of nodes  $(s_i, t_i) \in V \times V$ . Suppose each node  $s_i$  wants to communicate with its respective node  $t_i$ ; we assume that node  $t_i$  does not know  $s_i$  beforehand. Our goal is to identify a path  $P_i$  in  $G$  between each pair  $s_i$  and  $t_i$ . We refer to this problem as **multicommodity routing**, to be formally defined in Section 2. In addition, if every node  $v \in V$  appears at most  $W$  times as  $s_i$  or  $t_i$ , then we say that this multicommodity routing instance has **width**  $W$ .

Our main technical contribution is an improved multi-commodity routing algorithm on random graphs with round complexity  $2^{O(\sqrt{\log n})}$ . This improves on a solution of Ghaffari et al. [13] which has round complexity  $2^{O(\sqrt{\log n \log \log n})}$ . In its simplest form, the theorem can be stated as follows.

► **Theorem 1.** *Consider a multicommodity routing instance of width  $\tilde{O}(1)$ . There is a multicommodity routing algorithm on  $G(n, \Omega(\log n))$  that runs in time  $2^{O(\sqrt{\log n})}$ .*

**General Graphs and Mixing Time.** In fact, our result generalizes to more than random graphs in the same way as [13]. As shown by [13], random graphs can be “embedded” into any network graph with an overhead proportional to the *mixing time*  $\tau_{\text{mix}}$  of the network graph, which we define below. Thus, we can generalize the multicommodity routing algorithm to work on any graph.

<sup>3</sup> Moreover, for many other models of random graphs, we can embed one round of this model (i.e., connecting each node to  $O(\log n)$  randomly selected nodes) with a small, typically  $\text{poly}(\log n)$  round, overhead. This would be by using  $O(n \log n)$  random walks,  $O(\log n)$  starting from each node, and walking them until the mixing time, which is like selected a random connection endpoint. This is similar to [13]. In many random graph families, these walks would mix in  $\text{poly}(\log n)$  rounds [6].



Identically to [13], we define (lazy) random walks as follows: in every step, the walk remains at the current node with probability  $1/2$ , and otherwise, it transitions to a uniformly random neighbor. We formally define the mixing time of a graph as follows:

► **Definition 2.** For a node  $u \in V$ , let  $\{P_u^t(v)\}_{v \in V}$  be the probability distribution on the nodes  $v \in V$  after  $t$  steps of a (lazy) random walk starting at  $u$ . The **mixing time** of the graph, denoted  $\tau_{\text{mix}}$ , is the minimum integer  $t$  such that for all  $u, v \in V$ ,  $\left|P_u^t(v) - \frac{\deg(v)}{2m}\right| \leq \frac{\deg(v)}{2mn}$ .

Our multicommodity routing algorithm for general graphs is therefore as follows:

► **Theorem 3.** *There is a distributed algorithm solving multicommodity routing in  $\tau_{\text{mix}} \cdot 2^{O(\sqrt{\log n})}$  rounds.*

Finally, by substituting our multicommodity routing algorithm into the one in [13], we get an improvement on distributed MST in mixing time.

► **Theorem 4.** *There is a distributed MST algorithm running in  $\tau_{\text{mix}} \cdot 2^{O(\sqrt{\log n})}$  rounds.*

We remark that, by a standard doubling trick, we can assume that the algorithm does not even know the mixing time  $\tau_{\text{mix}}$  beforehand.<sup>4</sup>

### 1.1.2 Transformation

Our second, more conceptual contribution is a transformation from parallel algorithms to distributed algorithms on random graphs. In particular, we show that any work-efficient parallel algorithm running in  $T$  rounds can be simulated on a distributed random graph network in  $T \cdot \tau_{\text{mix}} \cdot 2^{O(\sqrt{\log n})}$  rounds. The actual theorem statement, Theorem 14, requires formalizing the parallel and distributed models, so we do not state it here.

**Applications.** For applications of this transformation, we look at a recent line of work on near-linear time algorithms for flow-type problems. In particular, we investigate the approximate versions of shortest path, maximum flow, and transshipment (also known as uncapacitated minimum cost flow). Parallel  $(1 \pm \epsilon)$ -approximation algorithms for these problems running in  $O(m^{1+o(1)})$  work and  $O(m^{o(1)})$  time result from gradient descent methods combined with a parallel solver for symmetric diagonally dominant systems [28, 31, 32, 4]. Therefore, by combining these parallel algorithms with our distributed transformation, we obtain the following corollaries:

► **Corollary 5.** *There are distributed algorithms running in time*

$$\tau_{\text{mix}} \cdot 2^{O(\sqrt{\log n})}$$

*for  $(1 + \epsilon)$ -approximate single-source shortest path and transshipment, and running time*

$$\tau_{\text{mix}} \cdot 2^{O(\sqrt{\log n \log \log n})}$$

*for  $(1 - \epsilon)$ -approximate maximum flow.*

<sup>4</sup> Indeed, begin with a guess  $\tau = 1$  for the value of  $\tau_{\text{mix}}$  and run the algorithm, assuming that  $\tau_{\text{mix}} = \tau$ . If the algorithm takes more than  $\tau \cdot 2^{O(\sqrt{\log n})}$  rounds, then every node in the distributed network immediately terminates the algorithm early and restarts with  $\tau$  multiplied by 2.

Finally, in the case of random graphs, another classical problem is the computation of a Hamiltonian cycle. Since an  $\tilde{O}(n)$ -work,  $\tilde{O}(1)$ -time parallel algorithm is known [7], we have an efficient distributed algorithm to compute Hamiltonian cycles.

► **Corollary 6.** *For large enough constant  $C$ , we can find a Hamilton cycle on  $G(n, d)$  with  $d = C \log n$  in  $2^{O(\sqrt{\log n})}$  rounds, w.h.p.*

This problem has attracted recent attention in the distributed setting. The main result of [5] is a distributed Hamiltonian cycle algorithm that runs in  $\Omega(n^\delta)$  rounds for graphs  $G(n, d)$  with  $d = \Omega(\log n/n^\delta)$  for any constant  $0 < \delta \leq 1$ . Thus, our algorithm greatly improves upon their result, both in number of rounds and in the parameter  $d$ .

## 1.2 Some Other Related Work

There has been a long history [34, 8, 16, 29, 10] in translating the ideal PRAM model into more practical parallel models, such as the celebrated BSP model of Valiant [34]. These transformations typically track many more parameters, such as communication and computation, than our transformation from PRAM to CONGEST, which only concerns the round complexity of the CONGEST algorithm.

There has also been work in the intersection of distributed computing and algorithms on random graphs. The task of computing a Hamiltonian cycle on a random graph was initiated by Levy et al. [22] and improved recently in [5]. Computation of other graph-theoretic properties on random graphs, such as approximate minimum dominating set and maximum matching, has been studied in a distributed setting in [17].

## 2 Multicommodity Routing

We formally define the multicommodity routing problem below, along with the congestion and dilation of a solution to this problem.

► **Definition 7.** A multicommodity routing instance consists of pairs of nodes  $(s_i, t_i) \in V \times V$ , such that each  $t_i$  is known to node  $s_i$ . A solution consists of a (not necessarily simple) path  $P_i$  connecting nodes  $s_i$  and  $t_i$  for every  $i$ , such that every node on  $P_i$  knows its two neighbors on  $P_i$ .

The input has **width**  $W$  if every node  $v \in V$  appears at most  $W$  times as  $s_i$  or  $t_i$ .

For a given solution of paths, the **dilation** is the maximum length of a path, and the **congestion** is the maximum number of times an edge appears in total over all paths. More precisely, if  $c_i(e)$  is the number of occurrences of edge  $e \in E(G)$  in path  $P_i$ , then the congestion is  $\max_{e \in E(G)} \sum_i c_i(e)$ .

The significance of the congestion and dilation parameters lies in the following lemma from [12], whose proof uses the standard trick of *random delays* from packet routing [21]. In particular, if a multicommodity routing algorithm runs efficiently and outputs a solution of low congestion and dilation, then each node  $s_i$  can efficiently route messages to node  $t_i$ .

► **Theorem 8** ([12]). *Suppose we solve a multicommodity routing instance  $\{(s_i, t_i)\}_i$  and achieve congestion  $c$  and dilation  $d$ . Then, in  $\tilde{O}(c + d)$  rounds, every node  $s_i$  can send one  $O(\log n)$ -bit message to every node  $t_i$ , and vice versa.*

We now provide our algorithm for multicommodity routing, improving the congestion and dilation factors from  $2^{O(\sqrt{\log n \log \log n})}$  in [13] to  $2^{O(\sqrt{\log n})}$ . Like [13], our algorithm uses the concept of *embedding* a graph, defined below.

► **Definition 9.** Let  $H$  and  $G$  be two graphs on the same node set. We say that an algorithm **embeds**  $H$  into  $G$  with congestion  $c$  and dilation  $d$  if the algorithm solves the following multicommodity routing instance on  $G$ : the  $(s_i, t_i)$  pairs are precisely the edges of  $H$ , the congestion is  $c$ , and the dilation is  $d$ . For each  $(s, t) \in E(H)$ , the path  $P_{s,t}$  (in  $G$  from  $s$  to  $t$ ) is called the **embedded path** for edge  $(s, t)$ .

Our multicommodity routing algorithm will *recursively* embed graphs. We use the following helper lemma.

► **Lemma 10.** *Suppose there is a distributed algorithm  $\mathcal{A}_1$  embedding graph  $G_1$  into network  $G_0$  with congestion  $c_1$  and dilation  $d_1$  in  $T_1$  rounds, and another distributed algorithm  $\mathcal{A}_2$  embedding graph  $G_2$  into network  $G_1$  with congestion  $c_2$  and dilation  $d_2$  in  $T_2$  rounds. Then, there is a distributed algorithm embedding  $G_2$  into network  $G_0$  with congestion  $c_1 c_2$  and dilation  $d_1 d_2$  in  $T_1 + T_2 \cdot \tilde{O}(c_1 + d_1)$  rounds.*

**Proof.** First, we provide the embedding without the algorithm. For each pair  $(s, t) \in E(G_1)$ , let  $P_{s,t}^1$  be the embedded path in  $G_0$ , and for each pair  $(s, t) \in E(G_2)$ , let  $P_{s,t}^2$  be the embedded path in  $G_1$ . To embed edge  $(s, t) \in E(G_2)$  into  $E_0$ , consider the path  $P_{s,t}^2 := (s = v_0, v_1, v_2, \dots, v_\ell = t)$ ; the embedded path for  $(s, t)$  in  $G_0$  is precisely the concatenation of the paths  $P_{v_{i-1}, v_i}^1$  for  $i \in [\ell]$  in increasing order. Since  $\ell \leq d_2$  and each path  $P_{v_{i-1}, v_i}^1$  has length at most  $d_1$ , the total length of the embedded path for  $(s, t)$  in  $G_0$  is at most  $d_1 d_2$ , achieving the promised dilation.

For congestion, let  $c_{s,t}^1(e)$  denote the number of occurrences of edge  $e \in E(G_0)$  in  $P_{s,t}^1$ . Since each edge  $(s, t) \in E(G_1)$  shows up at most  $c_2$  times among all  $P_{s',t'}^2$ , the number of times the path  $P_{s,t}^1$  is concatenated in the embedding is at most  $c_{s,t}^1(e) \cdot c_2$ . Therefore, edge  $e \in E(G_0)$  occurs at most  $\sum_{s,t} c_{s,t}^1(e) \cdot c_2 \leq c_1 c_2$  times among all the concatenated paths embedding  $G_2$  into  $G_0$ .

Finally, we describe the embedding algorithm. First, the algorithm on  $G_0$  runs  $\mathcal{A}_1$ , obtaining the embedding of  $G_1$  into  $G_0$  in  $T_1$  rounds. We now show how to emulate a single round of  $\mathcal{A}_2$  running on network  $G_1$  using  $\tilde{O}(c_1 + d_1)$  rounds on network  $G_0$ . Suppose that, on a particular round,  $\mathcal{A}_2$  has each node  $s$  send a message  $x$  to node  $t$  for every  $(s, t) \in E(G_1)$ . Since the embedding of  $G_1$  into  $G_0$  is a multicommodity routing instance, we use Theorem 8, where each node  $s$  tries to route that same message  $x$  to node  $t$ . This runs in  $\tilde{O}(c_1 + d_1)$  rounds for a given round of  $\mathcal{A}_2$ . Altogether, we spend  $T_1 + T_2 \cdot \tilde{O}(c_1 + d_1)$  rounds to emulate the entire  $\mathcal{A}_2$ . ◀

We now prove our main result, Theorem 1. We actually prove a stronger version of it, stated below.

► **Theorem 11.** *Consider a multicommodity routing instance of width  $\tilde{O}(1)$ . There is a multicommodity routing algorithm on  $G(n, \Omega(\log n))$  that achieves congestion and dilation  $2^{O(\sqrt{\log n})}$ , and runs in time  $2^{O(\sqrt{\log n})}$ .*

**Proof.** Following [13], our strategy is to construct graph embeddings recursively, forming a hierarchical decomposition. We start off by embedding a graph of sufficiently high degree in  $G$ , similar to the “Level Zero Random Graph” embedding of [13]. Essentially, the embedded paths are random walks in  $G$  of length  $\tau_{\text{mix}}$ ; we refer the reader to [13] for details. Note that, like in [13], we are embedding the graph  $G(m, d)$ , not the graph  $G(n, d)$ .

► **Lemma 12** ([13], Section 3.1.1). *On any graph  $G$  with  $n$  nodes and  $m$  edges, we can embed a random graph  $G(m, d)$  with  $d \geq 200 \log n$  into  $G$  with congestion  $\tilde{O}(\tau_{\text{mix}} \cdot d)$  and dilation  $\tau_{\text{mix}}$  in time  $\tilde{O}(\tau_{\text{mix}} \cdot d)$ .*

For our instance,  $\tau_{\text{mix}} = O(\log n)$  since  $G \sim G(n, \Omega(\log n))$ . Let  $d := 2^{10\sqrt{\log n}}$ . For this value of  $d$  in the lemma, we obtain an embedding  $G_0 \sim G(m, d)$  into  $G$  in time  $\tilde{O}(2^{10\sqrt{\log n}})$ .

Similarly to [13], our first goal is to obtain graphs  $G_1, G_2, \dots, G_K$  which form some hierarchical structure, such that each graph  $G_i$  embeds into  $G_{i-1}$  with small congestion and dilation. Later on, we will exploit the hierarchical structure of the graphs  $G_0, G_1, G_2, \dots, G_K$  in order to route each  $(s_i, t_i)$  pair.

To begin, we first describe the embedding of  $G_1$  into  $G_0$ . Like [13], we first randomly partition the nodes of  $G_0$  into  $\beta$  sets  $A_1, \dots, A_\beta$  so that  $|A_i| = \Theta(m/\beta)$ . Our goal is to construct and embed  $G_1$  into  $G_0$  with congestion 1 and dilation 2, where  $G_1$  has the following structure: it is a disjoint union, over all  $i \in [\beta]$ , of a random graph  $G_1^{(i)} \sim G(|A_i|, d/4)$  on the set  $A_i$ . By definition,  $G_0$  and  $G_1$  share the same node set. Note that [13] does a similar graph embedding, except with congestion and dilation  $O(\log n)$ ; improving the factors to  $O(1)$  is what constitutes our improvement.

Fix a set  $A_i$ ; we proceed to construct the random graph in  $A_i$ . For a fixed node  $u \in V(G_0)$ , consider the list of outgoing neighbors of  $u$  in  $A_i$ . Note that since  $G_0$  can have multi-edges, a node in  $A_i$  may appear multiple times in the list. Now, inside the local computation of node  $u$ , randomly group the nodes in the list into ordered pairs, leaving one element out if the list size is odd. For each ordered pair  $(v_1, v_2) \in A_i \times A_i$ , add  $v_2$  into  $v_1$ 's list of outgoing edges in  $G_1^{(i)}$ , and embed this edge along the path  $(v_1, u, v_2)$  of length 2. In this case, since the paths are short, node  $u$  can inform each pair  $(v_1, v_2)$  the entire path  $(v_1, u, v_2)$  in  $O(1)$  rounds.

Since node  $u$  has  $d$  outgoing neighbors, the expected number of outgoing neighbors of  $u$  in  $A_i$  is  $d/\beta$ . By Chernoff bound, the actual number is at least  $0.9d/\beta$  w.h.p., so there are at least  $0.4d/\beta$  ordered pairs w.h.p. Over all nodes  $u$ , there are at least  $0.4md/\beta$  pairs total.

We now argue that, over the randomness of the construction of  $G_0$ , the pairs are uniformly and independently distributed in  $A_i \times A_i$ . We show this by revealing the randomness of  $G_0$  in two steps. If, for each node  $u$ , we first reveal which set  $A_j$  each outgoing neighbor of  $u$  belongs to, and then group the outgoing neighbors in  $A_i$  into pairs, and finally reveal the actual outgoing neighbors, then each of the at least  $0.4md/\beta$  pairs is uniformly and independently distributed in  $A_i \times A_i$ . Therefore, each node  $v \in A_i$  is expected to receive at least  $\frac{0.4md/\beta}{m/\beta} = 0.4d$  outgoing neighbors, or at least  $0.25d$  outgoing neighbors w.h.p. by Chernoff bound. Finally, we have each node in  $A_i$  randomly discard outgoing neighbors until it has  $d/4$  remaining. The edges remaining in  $A_i$  form the graph  $G_1^{(i)}$ , which has distribution  $G(|A_i|, d/4)$ . Thus, we have embedded a graph  $G_1$  consisting of  $\beta$  disjoint random graphs  $G(\Theta(m/\beta), d/4)$  into  $G_0$ , where every embedded path is edge-disjoint in  $G_0$  and has length 2. In other words, the embedding has congestion 1 and dilation 2.

We apply recursion in the same manner as in [13]: recurse on each  $G_1^{(i)}$  (in parallel) by partitioning its vertices into another  $\beta$  sets  $A_1, \dots, A_\beta$ , building a random graph on each set, and taking their disjoint union. More precisely, suppose the algorithm begins with a graph  $H_0 \sim G(|V(G')|, d/4^{t-1})$  on depth  $k$  of the recursion tree (where the initial embedding of  $G_1$  into  $G_0$  has depth 1). The algorithm randomly partitions the nodes of  $H$  into  $A_1, \dots, A_\beta$  and defines a graph  $H_1$  similar to  $G_1$  from before: it is a disjoint union, over all  $i \in [\beta]$ , of a random graph  $H_1^{(i)} \sim G(|A_i|, d/4^t)$  on the set  $A_i$ . Finally, the algorithm recurses on each  $H_1^{(i)}$ . This recursion stops when the graphs have size at most  $2^5\sqrt{\log n}$ ; in other words, if  $|V(H_0)| \leq 2^5\sqrt{\log n}$ , then the recursive algorithm exits immediately instead of performing the above routine.

Once the recursive algorithm finishes, we let  $G_k$  be the disjoint union of all graphs  $H_1^{(i)}$  constructed on a recursive call of depth  $k$ . Observe that  $G_k$  has the same node set as  $G_0$ . Moreover, since, on each recursive step the sizes of the  $A_i$  drop by a factor of  $1/\beta$  in

expectation, or at most  $2/\beta$  w.h.p., the recursion goes for at most  $\log_{\beta/2} n \leq 2\sqrt{\log n}$  levels. Therefore, for each disjoint random graph in each  $G_k$ , the number of outgoing neighbors is always at least  $d/4^{2\sqrt{\log n}} \geq 2^{6\sqrt{\log n}}$ . In addition, since every embedding of  $G_k$  into  $G_{k-1}$  has congestion 1 and dilation 2, by applying Lemma 10 repeatedly,  $G_K$  embeds into  $G_0$  with congestion 1 and dilation  $2^{2\sqrt{\log n}}$ , and into  $G$  with congestion and dilation  $2^{O(\sqrt{\log n})}$ . Moreover, on each recursion level  $k$ , the embedding algorithm takes a constant number of rounds on the graph  $G_{k-1}$ , which can be simulated on  $G$  in  $2^{O(\sqrt{\log n})}$  rounds by Lemma 10.

Now we discuss how to route each  $(s_i, t_i)$  pair. Fix a pair  $(s, t)$ ; at a high level, we will iterate over the graphs  $G_0, G_1, G_2, \dots$  while maintaining the invariant that  $s$  and  $t$  belong to the same connected component in  $G_k$ . Initially, this holds for  $G_0$ ; if it becomes false when transitioning from  $G_{k-1}$  to  $G_k$ , then we replace  $s$  with a node  $s'$  in the connected component of  $t$  in  $G_k$ . We claim that in fact, w.h.p., there is such a node  $s'$  that is *adjacent* to  $s$  in  $G_{k-1}$ ; hence,  $s$  can send its message to  $s'$  along the network  $G_{k-1}$ , and the algorithm proceeds to  $G_k$  pretending that  $s$  is now  $s'$ . This process is similar to that in [13], except we make do without their notion of “portals” because of the large degree of  $G_0 - 2^{\Theta(\sqrt{\log n})}$  compared to  $\Theta(\log n)$  in [13].

We now make the routing procedure precise. For a given  $G_k$  with  $k < K$ , if  $s$  and  $t$  belong to the same connected component of  $G_k$ , then we do nothing. Otherwise, since  $s$  has at least  $2^{6\sqrt{\log n}} = \omega(\beta \log n)$  neighbors, w.h.p., node  $s$  has an outgoing neighbor  $s'$  in the connected component of  $G_k$  containing  $t$ ; if there are multiple neighbors, one is chosen at random. Node  $s$  *relays* the message along this edge to  $s'$ , and the pair  $(s, t)$  is replaced with  $(s', t)$  upon applying recursion to the next level.<sup>5</sup> Therefore, we always maintain the invariant that in each current  $(s, t)$  pair, both  $s$  and  $t$  belong in the same random graph.

We now argue that w.h.p., each vertex  $s'$  has  $\tilde{O}(1)$  messages after this routing step. By assumption, every node  $v \in V$  appears  $\tilde{O}(1)$  times as  $t_j$ , so there are  $|A_i| \cdot \tilde{O}(1)$  many nodes  $t_j$  that are inside  $A_i$ . For each such  $t_j$  with  $s_j \notin A_i$ , over the randomness of  $G_{k-1}$ , the neighbor  $s'_j$  of  $s_j$  inside  $A_i$  chosen to relay the message from  $s_j$  is uniformly distributed in  $A_i$ . By Chernoff bound, each node in  $A_i$  is chosen to relay a message  $\tilde{O}(1)$  times when transitioning from  $G_{k-1}$  to  $G_k$ . In total, each node  $v \in V$  appears  $\tilde{O}(1)$  times as  $s_i$  in the beginning, and receives  $\tilde{O}(1)$  messages to relay for each of  $O(\sqrt{\log n})$  iterations. It follows that every node always has  $\tilde{O}(1)$  messages throughout the algorithm.

Finally, in the graph  $G_K$ , we know that each  $(s_j, t_j)$  pair is in the same connected component of  $G_K$ . Recall that each connected component in  $G_K$  has at most  $2^{5\sqrt{\log n}}$  nodes, each with degree at least  $2^{6\sqrt{\log n}}$  (possibly with self-loops and parallel edges). It follows that w.h.p., each connected component is a “complete” graph, in the sense that every two nodes in the component are connected by at least one edge. Therefore, we can route each  $(s_j, t_j)$  pair trivially along an edge connecting them.

As for running time, since each graph  $G_0, G_1, \dots, G_K$  embeds into  $G$  with congestion and dilation  $2^{O(\sqrt{\log n})}$  by Lemma 10, iterating on each graph  $G_k$  takes  $2^{O(\sqrt{\log n})}$  rounds. Therefore, the total running time is  $2^{O(\sqrt{\log n})}$ , concluding Theorem 11. ◀

<sup>5</sup> In reality, node  $s$  does not know which set  $A_i$  contains node  $t$ . Like [13], we resolve this issue using  $\tilde{O}(1)$ -wise independence, which does not affect the algorithm’s performance. Since  $\Theta(W \log n)$  bits of randomness suffice for  $W$ -wise independence [1], we can have one node draw  $\Theta(W \log n) = \tilde{O}(1)$  random bits at the beginning of the iteration and broadcast them to all the nodes in  $\tilde{O}(1)$  time. Then, every node can locally compute the set  $A_i$  that contains any given node  $t$ ; see [13] for details.

## 31:10 Distributed Algorithms via Transformations from Parallel Algorithms

For general graphs, we can repeat the same algorithm, except we embed  $G_0$  with congestion and dilation  $\tilde{O}(\tau_{\text{mix}} \cdot 2^{10\sqrt{\log n}})$  instead of  $\tilde{O}(2^{10\sqrt{\log n}})$ , obtaining the following:

► **Corollary 13.** *Consider a multicommodity routing algorithm where every node  $v \in V$  appears  $\tilde{O}(1)$  times as  $s_i$  or  $t_i$ . There is a multicommodity routing algorithm that achieves congestion and dilation  $\tau_{\text{mix}} \cdot 2^{O(\sqrt{\log n})}$ , and runs in time  $\tau_{\text{mix}} \cdot 2^{O(\sqrt{\log n})}$ .*

Combining Theorem 8 and Corollary 13 proves Theorem 3.

### 3 Parallel to Distributed

In this section, we present our procedure to simulate parallel algorithms on distributed graph networks.

**Parallel Model Assumptions.** To formalize our transformation, we make some standard input assumptions to work-efficient parallel algorithms:

1. The input graph is represented in adjacency list form. There is a pointer array of size  $n$ , whose  $i$ 'th element points to an array of neighbors of vertex  $v_i$ . The  $i$ 'th array of input begins with  $\deg(v_i)$ , followed by the  $\deg(v_i)$  neighbors of vertex  $v_i$ .
2. There are exactly  $2m$  processors.<sup>6</sup> Each processor knows its ID, a unique number in  $[2m]$ , and has unlimited local computation and memory.
3. There is a shared memory block of  $\tilde{O}(mT)$  entries, including the output tape, where  $T$  is the running time of the parallel algorithm.<sup>7</sup> In every round, each processor can read or write from any entry in unit time (CRCW model). If multiple processors write to the same entry on the same round, then an arbitrary write is selected for that round.
4. If the output is a subgraph, then the output tape is an array of the subgraph edges.

**Distributed Model Assumptions.** Similarly, we make the following assumptions on the distributed model.

1. Each node knows its neighbors in the input graph, as well as its ID, a unique number of  $\Theta(\log n)$  bits. Each node has unlimited local computation and memory.
2. If the output is a subgraph, each node should know its incident edges in the subgraph.

► **Theorem 14.** *Under the above parallel and distributed model assumptions, a parallel graph algorithm running in  $T$  rounds can be simulated by a distributed algorithm in  $T \cdot \tau_{\text{mix}} \cdot 2^{O(\sqrt{\log n})}$  rounds.*

**Proof.** Our goal is to simulate one round of the parallel algorithm in  $\tau_{\text{mix}} \cdot 2^{O(\sqrt{\log n})}$  rounds in the distributed model, from which the theorem follows. To do so, we need to simulate the processors, input data, shared memory, and output.

---

<sup>6</sup> If the algorithm uses  $m^{1+o(1)}$  processors, then we can have each of the  $2m$  processors simulate  $m^{o(1)}$  of them.

<sup>7</sup> If the algorithm uses much more than  $mT$  memory addresses, then we can hash the memory addresses down to a hash table of  $\tilde{O}(mT)$  entries.

**Processors.** Embed a random graph  $G_0 = G(2m, \Theta(\log n)/m)$  into the network graph, as in [13]. Every node in  $G_0$  simulates one processor so that all  $2m$  processors are simulated; this means that every node  $v \in V$  in the original network simulates  $\deg(v)$  processors. Let the nodes of  $G_0$  and the processors be named  $(v, j)$ , where  $v \in V$  and  $j \in [\deg(v)]$ . Node/processor  $(v, j)$  knows the  $j$ 'th neighbor of  $v$ , and say,  $(v, 1)$  also knows the value of  $\deg(v)$ . Therefore, all input data to the parallel algorithm is spread over the processors  $(v, j)$ . From now on, we treat graph  $G_0$  as the new network graph in the distributed setting.

**Shared memory.** Shared memory is spread over all  $2m$  processors. Let the shared memory be split into  $2m$  blocks of size  $B$  each, where  $B := \tilde{O}(1)$ . Processor  $(v_i, j)$  is in charge of block  $\sum_{i' < i} \deg(v_{i'}) + j$ , so that each block is maintained by one processor. To look up block  $k$  in the shared memory array, a processor needs to write  $k$  as  $\sum_{i' < i} \deg(v_{i'}) + j$  for some  $(v_i, j)$ . Suppose for now that each processor knows the map  $\phi : [2m] \rightarrow V \times \mathbb{N}$  from index  $k$  to tuple  $(v_i, j)$ ; later on, we remove this assumption.

On a given parallel round, if a processor wants to read or write to block  $k$  of shared memory, it sends a request to node  $\phi(k)$ . One issue is the possibility that many nodes all want to communicate with processor  $\phi(k)$ , and in the multicommodity routing problem, we only allow each target node to appear  $\tilde{O}(1)$  times in the  $(s_i, t_i)$  pairs. We solve this issue below, whose proof is deferred to Appendix A.

► **Lemma 15.** *Consider the following setting: there is a node  $v_0$ , called the root, in possession of a memory block, and nodes  $v_1, \dots, v_k$ , called leaves, that request this memory block. The root node does not know the identities of the leaf nodes, but the leaf nodes know the identity  $v_0$  of the root node. Then, in  $\tilde{O}(1)$  multicommodity routing calls of width  $\tilde{O}(1)$ , the nodes  $v_1, \dots, v_k$  can receive the memory block of node  $v_0$ .*

*Now consider multiple such settings in parallel, where every node in the graph is a root node in at most one setting, and a leaf node in at most one setting. Then, in  $\tilde{O}(1)$  multicommodity routing calls of width  $\tilde{O}(1)$ , every leaf node can receive the memory block of its corresponding root node.*

Combining Lemma 15 and the multicommodity algorithm of Corollary 13 gives the desired distributed running time of  $\tau_{\text{mix}} \cdot 2^{O(\sqrt{\log n})}$  per parallel round, modulo the assumption that each processor knows the map  $\phi$ .

To remove the above assumption, we do the following as a precomputation step. We allocate an auxiliary array of size  $n$ , and our goal is to fill entry  $i$  with  $\sum_{i' < i} \deg(v_{i'}) + j$ . Let processor  $(v_i, 1)$  be in charge of entry  $i$ . Initially, processor  $(v_i, 1)$  fills entry  $i$  with  $\deg(v_i)$ , which it knows. Then, getting the array we desire amounts to computing prefix sums, and we can make the parallel prefix sum algorithm work here [19], since any processor looking for entry  $i$  knows to query  $(v_i, 1)$  for it. Finally, for a node to determine the entry  $\phi(k)$ , it can binary search on this auxiliary array to find the largest  $i$  with  $\sum_{i' < i} \deg(v_{i'}) < k$ , and set  $j := k - \sum_{i' < i} \deg(v_{i'})$ , which is the correct  $(v_i, j)$ .

**Input data.** If a processor in the parallel algorithm requests the value of  $\deg(v)$  or the  $i$ 'th neighbor of vertex  $v$ , we have the corresponding processor send a request to processor  $(v, i)$  for this neighbor. The routing details are the same as above.

**Output.** If the output is a subgraph of the original network graph  $G$ , then the distributed model requires each original node to know its incident edges in the subgraph. One way to do this is as follows: at the end of simulating the parallel algorithm, we can first sort the edges



lexicographically using the distributed translation of a parallel sorting algorithm. Then, each node  $(v_i, i)$  binary searches the output to determine if the edge of  $v$  to its  $i$ 'th neighbor  $u$  is in the output (either as  $(u, v)$  or as  $(v, u)$ ). Since each original node  $v \in V$  simulates each node/processor  $(v_i, i)$ , node  $v$  knows all edges incident to it in the output subgraph. ◀

### 3.1 Applications to Parallel Algorithms

The task of approximately solving symmetric diagonally dominant (SDD) systems  $Mx = b$  appears in many fast algorithms for  $\ell_p$  minimization problems, such as maximum flow and transshipment. Peng and Spielman [28] obtained the first polylogarithmic time parallel SDD solver, stated below. For precise definitions of SDD,  $\epsilon$ -approximate solution, and condition number, we refer the reader to [28].

► **Theorem 16** (Peng and Spielman [28]). *The SDD system  $Mx = b$ , where  $M$  is an  $n \times n$  matrix with  $m$  nonzero entries, can be  $\epsilon$ -approximately solved in parallel in  $\tilde{O}(m \log^3 \kappa)$  work and  $\tilde{O}(\log \kappa)$  time, where  $\kappa$  is the condition number of matrix  $M$ .*

Using our framework, we can translate this algorithm to a distributed setting, assuming that the input and output are distributed proportionally among the nodes.

► **Corollary 17.** *Let  $G$  be a network matrix. Consider a SDD matrix  $M$  with condition number  $\kappa$ , whose rows and columns indexed by  $V$ , and with nonzero entries only at entries  $M_{u,v}$  with  $(u, v) \in E$ . Moreover, assume that each nonzero entry  $M_{u,v}$  is known to both nodes  $u$  and  $v$ , and that each entry  $b_v$  is known to node  $v$ . In  $\tilde{O}(\tau_{mix} \cdot \log^4 \kappa)$  distributed rounds, we can compute an  $\epsilon$ -approximate solution  $x$ , such that each node  $v$  knows entry  $x_v$ .*

By combining parallel SDD solvers with gradient descent, we can compute approximate solutions maximum flow and minimum transshipment in parallel based on the recent work of Sherman and Becker et al. [31, 32, 4]. An added corollary is approximate shortest path, which can be reduced from transshipment [4].

► **Theorem 18** (Sherman, Becker et al. [31, 32, 4]). *The  $(1 + \epsilon)$ -approximate single-source shortest path and minimum transshipment problems can be solved in parallel in  $m \cdot 2^{O(\sqrt{\log n})}$  work and  $2^{O(\sqrt{\log n})}$  time. The  $(1 - \epsilon)$ -approximate maximum flow problem can be solved in parallel in  $m \cdot 2^{O(\sqrt{\log n \log \log n})}$  work and  $2^{O(\sqrt{\log n \log \log n})}$  time.*

► **Corollary 5.** *There are distributed algorithms running in time*

$$\tau_{mix} \cdot 2^{O(\sqrt{\log n})}$$

for  $(1 + \epsilon)$ -approximate single-source shortest path and transshipment, and running time

$$\tau_{mix} \cdot 2^{O(\sqrt{\log n \log \log n})}$$

for  $(1 - \epsilon)$ -approximate maximum flow.

Lastly, we consider the task of computing a Hamiltonian cycle on random graphs. This problem can be solved efficiently in parallel on random graphs  $G(n, d)$ , with  $d = C \log n$  for large enough constant  $C$ , by a result of Coppersmith et al. [7]. We remark that [7] only states that their algorithm runs in  $O(\log^2 n)$  time *in expectation*, but their proof is easily modified so that it holds w.h.p., at the cost of a larger constant  $C$ .

► **Theorem 19** (Coppersmith et al. [7]). *For large enough constant  $C$ , there is a parallel algorithm that finds a Hamiltonian cycle in  $G(n, C \log n)$  in  $O(\log^2 n)$  time, w.h.p.*



This immediately implies our fast distributed algorithm for Hamiltonian cycle; the result is restated below.

► **Corollary 6.** *For large enough constant  $C$ , we can find a Hamilton cycle on  $G(n, d)$  with  $d = C \log n$  in  $2^{O(\sqrt{\log n})}$  rounds, w.h.p.*

## 4 Conclusion and Open Problems

In this paper, we bridge the gap between work-efficient parallel algorithms and distributed algorithms in the CONGEST model. Our main technical contribution lies in a distributed algorithm for multicommodity routing on random graphs.

The most obvious open problem is to improve the  $2^{O(\sqrt{\log n})}$  bound in Theorem 1. Interestingly, finding a multicommodity routing solution with congestion and dilation  $O(\log n)$  is fairly easy if we are allowed  $\text{poly}(n)$  time. In other words, while there exist good multicommodity routing solutions, we do not know how to find them efficiently in a distributed fashion. Hence, finding an algorithm that both runs in  $\tilde{O}(1)$  rounds and computes a solution of congestion and dilation  $\tilde{O}(1)$  is an intriguing open problem, and would serve as evidence that distributed computation on well-mixing network graphs is as easy as work-efficient parallel computation, up to  $\tilde{O}(1)$  factors.

---

### References

- 1 Noga Alon and Joel H Spencer. *The probabilistic method*. John Wiley & Sons, 2004.
- 2 John Augustine, Gopal Pandurangan, Peter Robinson, Scott Roche, and Eli Upfal. Enabling robust and efficient distributed computation in dynamic peer-to-peer networks. In *Foundations of Computer Science (FOCS), 2015 IEEE 56th Annual Symposium on*, pages 350–369. IEEE, 2015.
- 3 Baruch Awerbuch and Christian Scheideler. The hyperring: a low-congestion deterministic data structure for distributed environments. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 318–327. Society for Industrial and Applied Mathematics, 2004.
- 4 Ruben Becker, Andreas Karrenbauer, Sebastian Krinninger, and Christoph Lenzen. Near-optimal approximate shortest paths and transshipment in distributed and streaming models. *arXiv preprint arXiv:1607.05127*, 2016.
- 5 Soumyottam Chatterjee, Reza Fathi, Gopal Pandurangan, and Nguyen Dinh Pham. Fast and efficient distributed computation of hamiltonian cycles in random graphs. *arXiv preprint arXiv:1804.08819*, 2018.
- 6 Colin Cooper and Alan Frieze. Random walks on random graphs. In *International Conference on Nano-Networks*, pages 95–106. Springer, 2008.
- 7 Don Coppersmith, Prabhakar Raghavan, and Martin Tompa. Parallel graph algorithms that are efficient on average. In *Foundations of Computer Science, 1987., 28th Annual Symposium on*, pages 260–269. IEEE, 1987.
- 8 David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten Von Eicken. Logp: Towards a realistic model of parallel computation. *ACM Sigplan Notices*, 28(7):1–12, 1993.
- 9 Paul Erdős and Alfréd Rényi. On random graphs, i. *Publicationes Mathematicae (Debrecen)*, 6:290–297, 1959.
- 10 Carlo Fantozzi, Andrea Pietracaprina, and Geppino Pucci. A general pram simulation scheme for clustered machines. *International Journal of Foundations of Computer Science*, 14(06):1147–1164, 2003.

- 11 Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proc. of the Symp. on Theory of Comp. (STOC)*, pages 114–118. ACM, 1978.
- 12 Mohsen Ghaffari and Bernhard Haeupler. Distributed algorithms for planar networks ii: Low-congestion shortcuts, mst, and min-cut. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 202–219. Society for Industrial and Applied Mathematics, 2016.
- 13 Mohsen Ghaffari, Fabian Kuhn, and Hsin-Hao Su. Distributed mst and routing in almost mixing time. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 131–140. ACM, 2017.
- 14 Leslie M Goldschlager. A unified approach to models of synchronous parallel machines. In *Proc. of the Symp. on Theory of Comp. (STOC)*, pages 89–94. ACM, 1978.
- 15 Richard M Karp. A survey of parallel algorithms for shared-memory machines. Technical report, University of California at Berkeley, 1988.
- 16 Richard M Karp, Michael Luby, and F Meyer auf der Heide. Efficient pram simulation on a distributed memory machine. *Algorithmica*, 16(4-5):517–542, 1996.
- 17 K Krzywdziński and Katarzyna Rybarczyk. Distributed algorithms for random graphs. *Theoretical Computer Science*, 605:95–105, 2015.
- 18 Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to parallel computing: design and analysis of algorithms*, volume 400. Benjamin/Cummings Redwood City, 1994.
- 19 Richard E Ladner and Michael J Fischer. Parallel prefix computation. *Journal of the ACM (JACM)*, 27(4):831–838, 1980.
- 20 Ching Law and Kai-Yeung Siu. Distributed construction of random expander networks. In *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications. IEEE Societies*, volume 3, pages 2133–2143. IEEE, 2003.
- 21 Tom Leighton, Bruce Maggs, and Satish Rao. Universal packet routing algorithms. In *Foundations of Computer Science, 1988., 29th Annual Symposium on*, pages 256–269. IEEE, 1988.
- 22 Eythan Levy, Guy Louchard, and Jordi Petit. A distributed algorithm to find hamiltonian cycles in  $g(n,p)$  random graphs. In *Workshop on Combinatorial and Algorithmic aspects of networking*, pages 63–74. Springer, 2004.
- 23 Peter Mahlmann and Christian Schindelhauer. Peer-to-peer networks based on random transformations of connected regular undirected graphs. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 155–164. ACM, 2005.
- 24 Gopal Pandurangan, Prabhakar Raghavan, and Eli Upfal. Building low-diameter peer-to-peer networks. *Selected Areas in Communications, IEEE Journal on*, 21(6):995–1002, 2003.
- 25 Gopal Pandurangan, Peter Robinson, and Amitabh Trehan. Dex: self-healing expanders. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 702–711. IEEE, 2014.
- 26 Gopal Pandurangan and Amitabh Trehan. Xheal: localized self-healing using expanders. In *Proceedings of the 30th annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 301–310. ACM, 2011.
- 27 David Peleg. *Distributed Computing: A Locality-sensitive Approach*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.
- 28 Richard Peng and Daniel A Spielman. An efficient parallel solver for sdd linear systems. In *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*, pages 333–342. ACM, 2014.

- 29 Andrea Pietracaprina and Geppino Pucci. The complexity of deterministic pram simulation on distributed memory machines. *Theory of Computing Systems*, 30(3):231–247, 1997.
- 30 Walter J Savitch and Michael J Stimson. Time bounded random access machines with parallel processing. *Journal of the ACM (JACM)*, 26(1):103–118, 1979.
- 31 Jonah Sherman. Nearly maximum flows in nearly linear time. In *Foundations of Computer Science (FOCS), 2013 IEEE 54th Annual Symposium on*, pages 263–269. IEEE, 2013.
- 32 Jonah Sherman. Generalized preconditioning and undirected minimum-cost flow. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 772–780. SIAM, 2017.
- 33 Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001.
- 34 Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

## A High Degree Communication

► **Lemma 15.** *Consider the following setting: there is a node  $v_0$ , called the root, in possession of a memory block, and nodes  $v_1, \dots, v_k$ , called leaves, that request this memory block. The root node does not know the identities of the leaf nodes, but the leaf nodes know the identity  $v_0$  of the root node. Then, in  $\tilde{O}(1)$  multicommodity routing calls of width  $\tilde{O}(1)$ , the nodes  $v_1, \dots, v_k$  can receive the memory block of node  $v_0$ .*

*Now consider multiple such settings in parallel, where every node in the graph is a root node in at most one setting, and a leaf node in at most one setting. Then, in  $\tilde{O}(1)$  multicommodity routing calls of width  $\tilde{O}(1)$ , every leaf node can receive the memory block of its corresponding root node.*

**Proof (Lemma 15).** We assume that every node has a unique ID in the range  $\{1, 2, \dots, n\}$ . The reduction from  $\Theta(\log n)$ -bit identifiers is standard: construct a BFS tree of depth  $D$ , where  $D$  is the diameter of the network graph, root the tree arbitrarily, and run prefix/infix/postfix ordering on the tree in  $O(D)$  time. Since  $\tau_{\text{mix}} \geq D$ , this takes  $O(\tau_{\text{mix}})$  time, which is negligible.

For now, consider the first setting of the lemma, with only one root node. Our goal is to establish a low-degree and low-depth (rooted) tree of communication, which contains the leaf nodes and possibly other nodes. Then, through calls of multicommodity routing, the root node sends the memory block to one of the nodes in this tree, which then gets propagated to all other nodes on the tree, including the leaf nodes. The key idea is that a *random low-depth tree* of communication, chosen from a certain distribution, will turn out to be low-degree as well. We now state the precise construction of this random tree.

Let  $K$  be a parameter that starts at  $n/2$  and decreases by a factor of 2 for  $T := \lceil \log_2(n/2) \rceil$  rounds. The node with ID 1 picks a hash function  $f : V \times [K] \rightarrow V$  for this iteration, and broadcasts it to all other nodes in  $D$  rounds. At the end, we will address the problem of encoding hash functions, but for now, assume that the hash function has mutual independence.

On iteration  $i$ , each leaf node computes a private random number  $k \in [K]$  and computes  $f(v_0, k) \in V$ , called the *connection point* for leaf node  $v_i$ . We will later show that, w.h.p., each node in  $V$  is the connection point of  $\tilde{O}(1)$  leaf nodes. Assuming this, we form the multicommodity routing instance where each leaf node requests a routing to its connection point, so that afterwards, each connection point  $v_j$  learns its set  $S_j$  of corresponding leaf nodes. Each connection point elects a random node  $v_j^* \in S_j$  as the *leader*, and routes the

entire set  $S_j$  to node  $v_j^*$  in another multicommodity routing instance. All nodes in  $S_j \setminus v_j^*$ , which did not receive the set  $S_j$ , drop out of the algorithm, leaving the leader  $v_j^*$  to route to other nodes in later iterations. At the end of the algorithm, there is only one leader left, and that leader routes directly to the root node  $v_0$ , receiving the memory block. Finally, the memory block gets propagated from the leaders  $v_j^*$  to the other nodes in  $S_j$  in reverse iteration order.

We now show that, w.h.p., each node in  $V$  is a connection point to  $\tilde{O}(1)$  leaf nodes; this would bound the width of the multicommodity instances by  $\tilde{O}(1)$ . Initially, there are at most  $n$  leaf nodes and  $n/2$  possible connection points, so each connection point has at most 2 leaf nodes in expectation, or  $O(\log n)$  w.h.p. On iteration  $t > 1$ , there are at most  $n/2^{t-1}$  leaf nodes left, since each of the  $n/2^{t-1}$  connection point elected one leader in the previous iteration and those are the only leaf nodes remaining. So each of the  $n/2^t$  connection points has at most 2 leaf nodes in expectation, or  $O(\log n)$  w.h.p.

Now consider the general setting, where we do the same thing in parallel over all groups of leaf nodes. On iteration  $t$ , let the set of remaining leaf nodes in each setting be  $L_1, \dots, L_r$ . For each set of leaf nodes  $L_i$ , a given node  $v_j$  has probability  $1/2^t$  of being selected as a connection point for  $L_i$ , and if so, it is expected to have at most  $\frac{|L_i|}{n/2^t}$  many leaf nodes in  $L_i$ , or  $O(\frac{|L_i|}{n/2^t} \log n) = O(\log n)$  w.h.p., using that  $|L_i| \leq n/2^{t-1}$ . Therefore, if  $X_j^i$  is the random variable of the number of leaf nodes in  $L_i$  assigned to node  $v_j$ , then  $\mathbb{E}[X_j^i] \leq |L_i|/n$ , and  $X_j^i = O(\log n)$  w.h.p. Conditioned on the w.h.p. statement, we use the following variant of Chernoff bound:

► **Theorem 20** (Chernoff bound). *If  $X_1, \dots, X_n$  are independent random variables in the range  $[0, C]$  and  $\mu := \mathbb{E}[X_1 + \dots + X_n]$ , then*

$$\Pr[X_1 + \dots + X_n \geq (1 + \delta)\mu] \leq \exp\left(-\frac{2\delta^2\mu^2}{nC^2}\right).$$

Taking the independent variables  $X_j^1, \dots, X_j^r$  and setting  $\delta := \Theta(\frac{\log^2 n}{\mu})$  and  $C := O(\log n)$ , we get that  $\mu = \sum_i |L_i|/n \leq 1$  and

$$\Pr[X_j^1 + \dots + X_j^r \geq \Theta(\log^2 n)] \leq \exp(-O(\log^2 n)).$$

Therefore, w.h.p., every node has  $O(\log^2 n)$  neighbors at any given round.

Lastly, we address the issue of encoding hash functions, which we solve using  $W$ -wise independent hash families for a small value  $W$ . Since the algorithm runs in  $\tilde{O}(1)$  rounds,  $W = \tilde{O}(1)$  suffices. It turns out that deterministic families of  $2^{O(W \log n)}$  hash functions exist [1], so the node with ID 1 can simply pick a random  $O(W \log n) = \tilde{O}(1)$ -bit string and broadcast it to all other nodes in  $D + \tilde{O}(1) = \tilde{O}(\tau_{\text{mix}})$  rounds. ◀

# Time-Message Trade-Offs in Distributed Algorithms

Robert Gmyr

Department of Computer Science, University of Houston, USA  
rgmyr@uh.edu

Gopal Pandurangan

Department of Computer Science, University of Houston, USA  
gopalpandurangan@gmail.com

---

## Abstract

---

This paper focuses on showing time-message trade-offs in distributed algorithms for fundamental problems such as leader election, broadcast, spanning tree (ST), minimum spanning tree (MST), minimum cut, and many graph verification problems. We consider the synchronous CONGEST distributed computing model and assume that each node has initial knowledge of itself and the identifiers of its neighbors – the so-called  $KT_1$  model – a well-studied model that also naturally arises in many applications. Recently, it has been established that one can obtain (almost) *singularly optimal* algorithms, i.e., algorithms that have *simultaneously* optimal time and message complexity (up to polylogarithmic factors), for many fundamental problems in the standard  $KT_0$  model (where nodes have only local knowledge of themselves and not their neighbors). The situation is less clear in the  $KT_1$  model. In this paper, we present several new distributed algorithms in the  $KT_1$  model that trade off between time and message complexity.

Our distributed algorithms are based on a uniform and general approach which involves constructing a *sparsified spanning subgraph* of the original graph – called a *danner* – that trades off the *number of edges* with the *diameter* of the sparsifier. In particular, a key ingredient of our approach is a distributed randomized algorithm that, given a graph  $G$  and any  $\delta \in [0, 1]$ , with high probability<sup>1</sup> constructs a danner that has diameter  $\tilde{O}(D + n^{1-\delta})$  and  $\tilde{O}(\min\{m, n^{1+\delta}\})$  edges in  $\tilde{O}(n^{1-\delta})$  rounds while using  $\tilde{O}(\min\{m, n^{1+\delta}\})$  messages, where  $n$ ,  $m$ , and  $D$  are the number of nodes, edges, and the diameter of  $G$ , respectively.<sup>2</sup> Using our danner construction, we present a family of distributed randomized algorithms for various fundamental problems that exhibit a trade-off between message and time complexity and that improve over previous results. Specifically, we show the following results (all hold with high probability) in the  $KT_1$  model, which subsume and improve over prior bounds in the  $KT_1$  model (King et al., PODC 2014 and Awerbuch et al., JACM 1990) and the  $KT_0$  model (Kutten et al., JACM 2015, Pandurangan et al., STOC 2017 and Elkin, PODC 2017):

1. **Leader Election, Broadcast, and ST.** These problems can be solved in  $\tilde{O}(D + n^{1-\delta})$  rounds using  $\tilde{O}(\min\{m, n^{1+\delta}\})$  messages for any  $\delta \in [0, 1]$ .
2. **MST and Connectivity.** These problems can be solved in  $\tilde{O}(D + n^{1-\delta})$  rounds using  $\tilde{O}(\min\{m, n^{1+\delta}\})$  messages for any  $\delta \in [0, 0.5]$ . In particular, for  $\delta = 0.5$  we obtain a distributed MST algorithm that runs in optimal  $\tilde{O}(D + \sqrt{n})$  rounds and uses  $\tilde{O}(\min\{m, n^{3/2}\})$  messages. We note that this improves over the singularly optimal algorithm in the  $KT_0$  model that uses  $\tilde{O}(D + \sqrt{n})$  rounds and  $\tilde{O}(m)$  messages.
3. **Minimum Cut.**  $O(\log n)$ -approximate minimum cut can be solved in  $\tilde{O}(D + n^{1-\delta})$  rounds using  $\tilde{O}(\min\{m, n^{1+\delta}\})$  messages for any  $\delta \in [0, 0.5]$ .
4. **Graph Verification Problems such as Bipartiteness, Spanning Subgraph etc.** These can be solved in  $\tilde{O}(D + n^{1-\delta})$  rounds using  $\tilde{O}(\min\{m, n^{1+\delta}\})$  messages for any  $\delta \in [0, 0.5]$ .

---

<sup>1</sup> Throughout, by “with high probability (w.h.p.)” we mean with probability at least  $1 - 1/n^c$  where  $n$  is the network size and  $c$  is some constant.

<sup>2</sup> The notation  $\tilde{O}$  hides a polylog( $n$ ) factor.



2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Randomized Algorithm,  $KT_1$  Model, Sparsifier, MST, Singular Optimality

Digital Object Identifier 10.4230/LIPIcs.DISC.2018.32

Funding Supported, in part, by NSF awards CCF-1527867, CCF-1540512, IIS-1633720, CCF-1717075, and BSF award 2016419.

## 1 Introduction

This paper focuses on a fundamental aspect of distributed algorithms: *trade-offs* for the two basic complexity measures of *time* and *messages*. The efficiency of distributed algorithms is traditionally measured by their time and message (or communication) complexities. Both complexity measures crucially influence the performance of a distributed algorithm. Time complexity measures the number of distributed “rounds” taken by the algorithm and determines the running time of the algorithm. Obviously, it is of interest to keep the running time as small as possible. Message complexity, on the other hand, measures the total amount of messages sent and received by all the processors during the course of the algorithm. In many applications, this is the dominant cost that also plays a major role in determining the running time and additional costs (e.g., energy) expended by the algorithm. For example, communication cost is one of the dominant costs in the distributed computation of large-scale data [17]. In another example, in certain types of communication networks, such as ad-hoc wireless networks, energy is a critical factor for measuring the efficiency of a distributed algorithm [15, 4]. Transmitting a message between two nodes in such a network has an associated energy cost and hence the total message complexity plays an important role in determining the energy cost of an algorithm. At the same time, keeping the number of rounds small also helps in reducing the energy cost. Thus, in various modern and emerging applications such as resource-constrained communication networks and distributed computation on large-scale data, it is crucial to design distributed algorithms that optimize both measures *simultaneously* [13, 17, 24, 26].

Unfortunately, designing algorithms that are simultaneously time- and message-efficient has proven to be a challenging task, which (for some problems) stubbornly defied all prior attempts of attack. Consequently, research in the last three decades in the area of distributed algorithms has focused mainly on optimizing either one of the two measures separately, typically at the cost of neglecting the other. In this paper, we focus on studying the two cost measures of time and messages *jointly*, and exploring ways to design distributed algorithms that work well under both measures (to the extent possible). Towards this goal, it is important to understand the relationship between these two measures. In particular, as defined in [25], we should be able to determine, for specific problems, whether it is possible to devise a distributed algorithm that is either *singularly optimal* or exhibits a *time-message trade-off*:

- **Singularly optimal:** A distributed algorithm that is optimal with respect to both measures simultaneously – in which case we say that the problem enjoys *singular optimality*.
- **Time-message trade-off:** Whether the problem inherently fails to admit a singularly optimal solution, namely, algorithms of better time complexity for it will necessarily incur higher message complexity and vice versa – in which case we say that the problem exhibits a *time-message trade-off*.

We note that, more generally, finding a simultaneously optimal (or almost optimal) solution may sometimes be difficult even for “singular optimality” problems and hence it might be useful to first design algorithms that have trade-offs.

This paper focuses on showing time-message trade-offs in distributed algorithms for fundamental problems such as leader election, broadcast, spanning tree (ST), minimum spanning tree (MST), minimum cut, and many graph verification problems. Throughout, we consider the synchronous CONGEST model (see Section 2.2 for details), a standard model in distributed computing where computation proceeds in discrete (synchronous) *rounds* and in each round only  $O(\log n)$  bits are allowed to be exchanged per edge (CONGEST) where  $n$  is the number of nodes in the network. It turns out that message complexity of a distributed algorithm depends crucially (as explained below) on the initial knowledge of the nodes; in this respect, there are two well-studied models – the  $\text{KT}_0$  and the  $\text{KT}_1$  model.<sup>3</sup> In the  $\text{KT}_0$  model (i.e., **K**nowledge **T**ill radius **0**), also called the *clean network model* [27], where nodes have initial local knowledge of only themselves (and not their neighbors), it has (only) been recently established that one can obtain (almost) *singularly optimal* algorithms, i.e., algorithms that have *simultaneously* optimal time and message complexity (up to polylogarithmic factors), for many fundamental problems such as leader election, broadcast, ST, MST, minimum cut, and approximate shortest paths (under some conditions) [18, 25, 7, 12]. More precisely, for problems such as leader election, broadcast, and ST, it has been shown [18] that one can design a singularly optimal algorithm in the  $\text{KT}_0$  model, that takes  $\tilde{O}(m)$  messages and  $O(D)$  rounds – both are tight (up to a polylog( $n$ ) factor); this is because  $\Omega(m)$  and  $\Omega(D)$  are, respectively, lower bounds on the message and time complexity for these problems in the  $\text{KT}_0$  model [18] – note that these lower bounds hold even for randomized Monte Carlo algorithms. The work of [25] (also see [7]) showed that MST is also (almost) singularly optimal, by giving a (randomized Las Vegas) algorithm that takes  $\tilde{O}(m)$  messages and  $\tilde{O}(D + \sqrt{n})$  rounds (both bounds are tight up to polylogarithmic factors). It can be shown that the singular optimality of MST in the  $\text{KT}_0$  model also implies the singular optimality of many other problems such as approximate minimum cut and graph verification problems (see Section 1.2). Recently, it was shown that approximate shortest paths and several other problems also admit singular optimality in the  $\text{KT}_0$  model [12].

On the other hand, in the  $\text{KT}_1$  model (i.e., **K**nowledge **T**ill radius **1**), in which each node has initial knowledge of itself and the *identifiers*<sup>4</sup> of its neighbors, the situation is less clear. The  $\text{KT}_1$  model arises naturally in many settings, e.g., in networks where nodes know the identifiers of their neighbors (as well as other nodes), e.g., in the Internet where a node knows the IP addresses of other nodes [23]. Similarly in models such as the  $k$ -machine model (as well as the congested clique), it is natural to assume that each processor knows the identifiers of all other processors [17, 13, 24, 26]. For the  $\text{KT}_1$  model, King et al. [16] showed a surprising and elegant result: There is a randomized Monte Carlo algorithm to construct an MST in  $\tilde{O}(n)$  messages ( $\Omega(n)$  is a message lower bound) and in  $\tilde{O}(n)$  time (see Section 2.3). Thus it is also possible to construct an ST, do leader election, and broadcast within these bounds.

<sup>3</sup> On the other hand, for time complexity it does not really matter whether nodes have initial knowledge of just themselves ( $\text{KT}_0$ ) or also of their neighbors ( $\text{KT}_1$ ); this is because this information (i.e., the identifiers of the neighbors) can be exchanged in one round in the CONGEST model. Hence, when focusing solely on optimizing time complexity, which is the typically the case in the literature, the distinction between  $\text{KT}_0$  and  $\text{KT}_1$  is not important and the actual model is not even explicitly specified.

<sup>4</sup> Note that only knowledge of the identifiers of neighbors is assumed, not other information such as the degree of the neighbors.



This algorithm is randomized and *not comparison-based*.<sup>5</sup> While this algorithm shows that one can achieve  $o(m)$  message complexity (when  $m = \omega(n \text{ polylog } n)$ ), it is *not* time-optimal – it can take significantly more than  $\tilde{\Theta}(D + \sqrt{n})$  rounds, which is a time lower bound even for Monte-Carlo randomized algorithms [5]. In subsequent work, Mashreghi and King [19] presented a trade-off between messages and time for MST: a Monte-Carlo algorithm that takes  $\tilde{O}(n^{1+\epsilon})$  messages and runs in  $O(n/\epsilon)$  time for any  $1 > \epsilon \geq \log \log n / \log n$ . We note that this algorithm takes at least  $O(n)$  time.

*A central motivating theme underlying this work is understanding the status of various fundamental problems in the  $KT_1$  model – whether they are singularly optimal or exhibit trade-offs (and, if so, to quantify the trade-offs).* In particular, it is an open question whether one can design a randomized (non-comparison based) MST algorithm that takes  $\tilde{O}(D + \sqrt{n})$  time and  $\tilde{O}(n)$  messages in the  $KT_1$  model – this would show that MST is (almost) singularly optimal in the  $KT_1$  model as well. In fact, King et al [16] ask whether it is possible to construct (even) an ST in  $o(n)$  rounds with  $o(m)$  messages. Moreover, can we take advantage of the  $KT_1$  model to get improved message bounds (while keeping time as small as possible) in comparison to the  $KT_0$  model?

## 1.1 Our Contributions and Comparison with Related Work

In this paper, we present several results that show trade-offs between time and messages in the  $KT_1$  model with respect to various problems. As a byproduct, we improve and subsume the results of [16] (as well as of Awerbuch et al. [2]) and answer the question raised by King et al. on ST/MST construction at the end of the previous paragraph in the affirmative. We also show that our results give improved bounds compared to the results in the  $KT_0$  model, including for the fundamental distributed MST problem.

Our time-message trade-off results are based on a uniform and general approach which involves constructing a *sparsified spanning subgraph* of the original graph – called a *danner* (i.e., “diameter-preserving spanner”) – that trades off the *number of edges* with the *diameter* of the sparsifier (we refer to Section 2.1 for a precise definition). *In particular, a key ingredient of our approach is a distributed randomized algorithm that, given a graph  $G$  and any  $\delta \in [0, 1]$ , with high probability<sup>6</sup> constructs a danner that has diameter  $\tilde{O}(D + n^{1-\delta})$  and  $\tilde{O}(\min\{m, n^{1+\delta}\})$  edges in  $\tilde{O}(n^{1-\delta})$  rounds while using  $\tilde{O}(\min\{m, n^{1+\delta}\})$  messages, where  $n$ ,  $m$ , and  $D$  are the number of nodes, edges, and the diameter of  $G$ , respectively.<sup>7</sup>* Using our danner construction, we present a family of distributed randomized algorithms for various fundamental problems that exhibit a trade-off between message and time complexity and that improve over previous results. Specifically, we show the following results (all hold with high probability) in the  $KT_1$  model (cf. Section 4):

1. **Leader Election, Broadcast, and ST.** These problems can be solved in  $\tilde{O}(D + n^{1-\delta})$  rounds using  $\tilde{O}(\min\{m, n^{1+\delta}\})$  messages for any  $\delta \in [0, 1]$ . These results improve over prior bounds in the  $KT_1$  model [16, 19, 2] as well the  $KT_0$  model [18] – discussed earlier

<sup>5</sup> Awerbuch et al. [2] show that  $\Omega(m)$  is a message lower bound for MST even in the  $KT_1$  model, if one allows only (possibly randomized Monte Carlo) *comparison-based* algorithms, i.e., algorithms that can operate on identifiers only by comparing them. The result of King et al. [16] breaches the  $\Omega(m)$  lower bound by using non-comparison-based technique by using the identifiers as input to hash functions. Our results also breach the  $\Omega(m)$  lower bound, since we use the results of King et al. as subroutines in our algorithms.

<sup>6</sup> Throughout, by “with high probability (w.h.p.)” we mean with probability at least  $1 - 1/n^c$  where  $n$  is the network size and  $c$  is some constant.

<sup>7</sup> The notation  $\tilde{O}$  hides a polylog( $n$ ) factor.



in Section 1. In particular, while the time bounds in [16, 19] are always at least linear, our bounds can be sublinear and the desired time-message trade-off can be obtained by choosing an appropriate  $\delta$ . It is worth noting that the early work of Awerbuch et al. [2] showed that broadcast can be solved by a deterministic algorithm in the  $\text{KT}_\rho$  model using  $O(\min\{m, n^{1+c/\rho}\})$  messages for some fixed constant  $c > 0$  in a model where each node has knowledge of the *topology* (not just identifiers) up to radius  $\rho$ . Clearly, our results improve over this (for the  $\text{KT}_1$  model), since  $\delta$  can be made arbitrarily small.

2. **MST and Connectivity.** These problems can be solved in  $\tilde{O}(D + n^{1-\delta})$  rounds using  $\tilde{O}(\min\{m, n^{1+\delta}\})$  messages for any  $\delta \in [0, 0.5]$ . In addition to getting any desired trade-off (by plugging in an appropriate  $\delta$ ), we can get a time optimal algorithm by choosing  $\delta = 0.5$ , which results in a distributed MST algorithm that runs in  $\tilde{O}(D + \sqrt{n})$  rounds and uses  $\tilde{O}(\min\{m, n^{3/2}\})$  messages. We note that when  $m = \tilde{\omega}(n^{3/2})$ , this improves over the recently proposed singularly optimal algorithms of [25, 7] in the  $\text{KT}_0$  model that use  $\tilde{O}(m)$  messages and  $\tilde{O}(D + \sqrt{n})$  rounds. It also subsumes and improves over the prior results of [19, 16] in the  $\text{KT}_1$  model that take (essentially)  $\tilde{O}(n)$  messages and  $\tilde{O}(n)$  time.<sup>8</sup>
3. **Minimum Cut.** An  $O(\log n)$ -approximation to the minimum cut value (i.e., edge connectivity of the graph) can be obtained in  $\tilde{O}(D + n^{1-\delta})$  rounds using  $\tilde{O}(\min\{m, n^{1+\delta}\})$  messages for any  $\delta \in [0, 0.5]$ . Our result improves over the works of [11, 21] that are (almost) time-optimal (i.e., take  $\tilde{O}(D + \sqrt{n})$  rounds), but not message optimal. In addition to getting any desired trade-off (by plugging in an appropriate  $\delta$ ), in particular, if  $\delta = 0.5$ , we obtain a  $\tilde{O}(\min\{m, n^{3/2}\})$  messages approximate minimum cut algorithm that runs in (near) optimal  $\tilde{O}(D + \sqrt{n})$  rounds. This improves the best possible bounds (for  $m = \tilde{\omega}(n^{3/2})$ ) that can be obtained in the  $\text{KT}_0$  model (cf. Section 1.2).
4. **Graph Verification Problems such as Bipartiteness,  $s - t$  Cut, Spanning Subgraph.** These can be solved in  $\tilde{O}(D + n^{1-\delta})$  rounds using  $\tilde{O}(\min\{m, n^{1+\delta}\})$  messages for any  $\delta \in [0, 0.5]$ .

## 1.2 High-Level Overview of Approach

**Danner.** A main technical contribution of this work is the notion of a *danner* and its efficient distributed construction that jointly focuses on both time and messages. As defined in Section 2.1, a danner of a graph  $G$  is a spanning subgraph  $H$  of  $G$  whose diameter, i.e.,  $\text{diam}(H)$ , is at most  $\alpha(\text{diam}(G)) + \beta$ , where  $\alpha \geq 1$  and  $\beta \geq 0$  are some parameters. The goal is to construct a danner  $H$  with as few edges as possible and with  $\alpha$  and  $\beta$  as small as possible. It is clear that very sparse danners exist: the BFS (breadth-first spanning) tree has only  $n - 1$  edges and its diameter is at most twice the diameter of the graph. However, it is not clear how to construct such a danner in a way that is efficient with respect to both messages and time, in particular in  $\tilde{O}(n)$  messages and  $\tilde{O}(D)$  time, or even  $o(m)$  messages and  $O(D)$  time, where  $D = \text{diam}(G)$ . Note that in the  $\text{KT}_0$  model, there is a tight lower bound (with almost matching upper bound) for constructing a danner: any distributed danner construction algorithm needs  $\Omega(m)$  messages and  $\Omega(D)$  time (this follows by reduction from leader election which has these lower bounds [18] – see Section 1). However, in the  $\text{KT}_1$  model, the status for danner construction is not known. We give (a family of) distributed algorithms for constructing a danner that trade off messages for time (Section 3).

<sup>8</sup> Mashreghi and King [19] also give an algorithm with round complexity  $\tilde{O}(\text{diam}(\text{MST}))$  and with message complexity  $\tilde{O}(n)$ , where  $\text{diam}(\text{MST})$  is the diameter of the output MST which can be as large as  $\Theta(n)$ .

**Danner Construction.** We present an algorithm (see Algorithm 1) that, given a graph  $G$  and any  $\delta \in [0, 1]$ , constructs a danner  $H$  of  $G$  that has  $\tilde{O}(\min\{m, n^{1+\delta}\})$  edges and diameter  $\tilde{O}(D + n^{1-\delta})$  (i.e., an  $\tilde{O}(n^{1-\delta})$  additive danner) using  $\tilde{O}(\min\{m, n^{1+\delta}\})$  messages and in  $\tilde{O}(n^{1-\delta})$  time (note that the time does not depend on  $D$ ). The main idea behind the algorithm is as follows. While vertices with low degree (i.e., less than  $n^\delta$ ) and their incident edges can be included in the danner  $H$ , to handle high-degree vertices we construct a dominating set that dominates the high-degree nodes by sampling roughly  $n^{1-\delta}$  nodes (among all nodes); these are called “center” nodes.<sup>9</sup> Each node  $v$  adds the edges leading to its  $\min\{\deg(v), n^\delta\}$  neighbors with the lowest identifiers (required for maintaining independence from random sampling) to  $H$ . It is not difficult to argue that each high-degree node is connected to a center in  $H$  and we use a relationship between the number of nodes in any dominating set and the diameter (cf. Lemma 5) to argue that the diameter of each connected component (or fragment) of  $H$  is at most  $\tilde{O}(n^{1-\delta})$ . We then use the FindAny algorithm of King et al. [16] to efficiently implement a distributed Boruvka-style merging (which is essentially the GHS algorithm [9]) of fragments in the subgraph induced by the high-degree nodes and the centers. The FindAny algorithm does not rely on identifier comparisons but instead uses random hash-functions to find an edge leaving a set of nodes very efficiently, which is crucial for our algorithm. In each merging phase, each fragment uses FindAny to efficiently find an outgoing edge; discovered outgoing edges are added to  $H$ . Only  $O(\log n)$  iterations are needed to merge all fragments into a connected graph and only  $\tilde{O}(\min\{m, n^{1+\delta}\})$  messages are needed overall for merging. At any time the set of centers in a fragment forms a dominating set of that fragment. Thereby, the above-mentioned relationship between dominating sets and diameters guarantees that the diameters of the fragments stay within  $\tilde{O}(n^{1-\delta})$ . We argue (Lemma 12) that the constructed subgraph  $H$  is an additive  $\tilde{O}(n^{1-\delta})$ -danner of  $G$ .

**Danner Applications.** What is the motivation for defining a danner and why is it useful? The answer to both of these questions is that a danner gives a uniform way to design distributed algorithms that are both time and message efficient for various applications as demonstrated in Section 4. Results for leader election, broadcast, and ST construction follow quite directly (cf. Section 4.1): Simply construct a danner and run the singularly optimal algorithm of [18] for the  $\text{KT}_0$  model on the danner subgraph. Since the danner has  $\tilde{O}(n^{1+\delta})$  edges and has diameter  $\tilde{O}(D + n^{1-\delta})$ , this gives the required bounds.

A danner can be used to construct a MST of a graph  $G$  (which also gives checking connectivity of a subgraph  $H$  of  $G$ ) using  $\tilde{O}(\min\{m, n^{1+\delta}\})$  messages in  $\tilde{O}(D + n^{1-\delta})$  time, for any  $\delta \in [0, 0.5]$ . Note that this subsumes the bounds of the singularly optimal algorithms in the  $\text{KT}_0$  model [25, 7]. The distributed MST construction (cf. Section 4.2) proceeds in three steps; two of these crucially use the danner. In the first step, we construct a danner and use it as a communication backbone to aggregate the degrees of all nodes and thus determine  $m$ , the number of edges. If  $m \leq n^{1+\delta}$ , then we simply proceed to use the singularly optimal algorithm of [25]. Otherwise, we proceed to Step 2, where we do *Controlled-GHS* which is a well-known ingredient in prior MST algorithms [22, 10, 25]. Controlled-GHS is simply Boruvka-style MST algorithm, where the diameter of all the fragments grow at a controlled rate. We use the graph sketches technique of King et al. (in particular the FindMin algorithm – cf. Section 2.3) for finding outgoing edges to keep the message complexity to  $\tilde{O}(n)$ . Crucially we run Controlled-GHS to only  $\lceil (1 - \delta) \log n \rceil$  iterations so that the number of fragments

<sup>9</sup> The idea of establishing a set of nodes that dominates all high-degree nodes has also been used by Aingworth et al. [1] and Dor et al. [6].

remaining at the end of Controlled-GHS is  $O(n^\delta)$  with each having diameter  $\tilde{O}(n^{1-\delta})$ ; all these take only  $\tilde{O}(n^{1-\delta})$  time, since the running time of Controlled-GHS is asymptotically bounded (up to a  $\log n$  factor) by the (largest) diameter of any fragment. In Step 3, we merge the remaining  $O(n^\delta)$  fragments; this is done in a way that is somewhat different to prior MST algorithms, especially those of [25, 10]. We simply continue the Boruvka-merging (not necessarily in a controlled way), but instead of using each fragment as the communication backbone, we do the merging “globally” using a BFS tree of the danner subgraph. The BFS tree of the danner has  $O(n)$  edges and has diameter  $\tilde{O}(D + n^{1-\delta})$ . In each merging phase, each node forwards at most  $O(n^\delta)$  messages (sketches corresponding to so many distinct fragments) to the root of the BFS tree which finds the outgoing edge corresponding to each fragment (and broadcasts it to all the nodes). The total message cost is  $\tilde{O}(n^{1+\delta})$  and, since the messages are pipelined, the total time is  $\tilde{O}(D + n^{1-\delta} + n^\delta) = \tilde{O}(D + n^{1-\delta})$  (for  $\delta = [0, 0.5]$ ). Building upon this algorithm, we give time and message efficient algorithms for  $O(\log n)$ -approximate minimum cut and graph connectivity problems (cf. Section 4.3).

### 1.3 Other Related Work

There has been extensive research on the distributed MST problem in the  $\text{KT}_0$  model, culminating in the singularly optimal (randomized) algorithm of [25] (see also [7]); see [25] for a survey of results in this area. The work of [25] also defined the notions of *singular optimality* versus *time-message trade-offs*. Kutten et al. [18] show the singular optimality of leader election (which implies the same for broadcast and ST) by giving tight lower bounds for both messages and time as well as giving algorithms that simultaneously achieve the tight bounds (see Section 1).

Compared to the  $\text{KT}_0$  model, results in the  $\text{KT}_1$  are somewhat less studied. The early work of Awerbuch et al. [2] studied time-message trade-offs for broadcast in the  $\text{KT}_1$  model. In 2015, King et al. [16] showed surprisingly that the basic  $\Omega(m)$  message lower bound that holds in the  $\text{KT}_0$  model for various problems such as leader election, broadcast, MST, etc. [18] can be breached in the  $\text{KT}_1$  model by giving a randomized Monte Carlo algorithm to construct an MST in  $\tilde{O}(n)$  messages and in  $\tilde{O}(n)$  time. The algorithm of King et al. uses a powerful randomized technique of *graph sketches* which helps in identifying edges going out of a cut efficiently without probing all the edges in the cut; this crucially helps in reducing the message complexity. We heavily use this technique (as a black box) in our algorithms as well. However, note that we could have also used other types of graph sketches (see e.g., [24]) which will yield similar results.

The  $\text{KT}_1$  model has been assumed in other distributed computing models such as the  $k$ -machine model [17, 24, 26] and the congested clique [13]. In [13] it was shown that the MST problem has a message lower bound of  $\Omega(n^2)$  which can be breached in the  $\text{KT}_1$  model by using graph sketches.

Distributed minimum cut has been studied by [11, 21], and the graph verification problems considered in this paper have been studied in [5]. However, the focus of these results has been on the time complexity (where  $\text{KT}_0$  or  $\text{KT}_1$  does not matter). We study these problems in the  $\text{KT}_1$  model focusing on both time and messages and present trade-off algorithms that also improve over the  $\text{KT}_0$  algorithms (in terms of messages) – cf. Section 1.1. We note that  $\tilde{\Omega}(D + \sqrt{n})$  is a time lower bound for minimum cut (for any non-trivial approximation) and for the considered graph verification problems [5]. It can be also shown (by using techniques from [18]) that  $\Omega(m)$  is a message lower bound in the  $\text{KT}_0$  model for minimum cut.

## 2 Preliminaries

Before we come to the main technical part of the paper, we introduce some notation and basic definitions, present our network model, and give an overview of some of the algorithms from the literature that we use for our results.

### 2.1 Notation and Definitions

For a graph  $G$  we denote its *node set* as  $V(G)$  and its *edge set* as  $E(G)$ . For a node  $u \in V(G)$  the set  $N_G(u) = \{v \in V(G) \mid \{u, v\} \in E(G)\}$  is the *open neighborhood* of  $u$  in  $G$  and  $\Gamma_G(u) = N_G(u) \cup \{u\}$  is its *closed neighborhood*. For a set of nodes  $S \subseteq V(G)$  we define  $\Gamma_G(S) = \bigcup_{u \in S} \Gamma_G(u)$ . The *degree* of a node  $u$  in  $G$  is  $\deg_G(u) = |N_G(u)|$ . For a path  $P = (u_0, \dots, u_\ell)$  we define  $V(P)$  to be the set of nodes in  $P$  and we define  $|P| = \ell$  to be the *length* of  $P$ . A set  $S \subseteq V(G)$  is a *dominating set* of  $G$  if  $\Gamma_G(S) = V(G)$ . The *domination number*  $\gamma(G)$  of a graph  $G$  is the size of a smallest dominating set of  $G$ . The *distance*  $d_G(u, v)$  between two nodes  $u, v \in V(G)$  is the length of a shortest path between  $u$  and  $v$  in  $G$ . We define the *diameter* (or the *hop diameter*) of  $G$  as  $\text{diam}(G) = \max_{u, v \in V(G)} d(u, v)$ , where the distances are taken in the graph by ignoring edge weights (i.e., each edge has weight 1). For all of this notation, we omit  $G$  when it is apparent from context. For  $S \subseteq V(G)$  the *induced subgraph*  $G[S]$  is defined by  $V(G[S]) = S$  and  $E(G[S]) = \{\{u, v\} \in E(G) \mid u, v \in S\}$ . A subgraph  $H \subseteq G$  is an  $(\alpha, \beta)$ -*spanner* of  $G$  if  $V(H) = V(G)$  and  $d_H(u, v) \leq \alpha \cdot d_G(u, v) + \beta$  for all  $u, v \in V(G)$ . In this work we make use of the weaker concept of a *diameter-preserving spanner*, or in short, *danner*: A subgraph  $H \subseteq G$  is a  $(\alpha, \beta)$ -*danner* of  $G$  if  $V(H) = V(G)$  and  $\text{diam}(H) \leq \alpha \cdot \text{diam}(G) + \beta$ . We say  $H$  is an additive  $\beta$ -danner if it is a  $(1, \beta)$ -danner. An  $(\alpha, \beta)$ -spanner is also an  $(\alpha, \beta)$ -danner but the reverse is not generally true. Hence, the notion of a danner is weaker than that of a spanner.

### 2.2 Model

We briefly describe the distributed computing model used. This is the synchronous CONGEST model (see, e.g., [22, 27]), which is now standard in the distributed computing literature.

A point-to-point communication network is modeled as an undirected weighted graph  $G = (V, E, w)$ , where the vertices of  $V$  represent the processors, the edges of  $E$  represent the communication links between them, and  $w(e)$  is the weight of edge  $e \in E$ . Let  $n = |V(G)|$  and  $m = |E(G)|$ . Without loss of generality, we assume that  $G$  is connected.  $D$  denotes the hop-diameter (that is, the unweighted diameter) of  $G$ , and, in this paper, by diameter we always mean hop-diameter. Each node hosts a processor with limited initial knowledge. Specifically, we make the common assumption that each node has a unique identifier (from  $\{1, \dots, \text{poly}(n)\}$ ), and at the beginning of computation each vertex  $v$  accepts as input its own identifier and the weights (if any) of the edges incident to it as well as the *identifiers of all its neighbors*. Thus, a node has *local* knowledge of only itself and its neighbor's identifiers; this is called the  $KT_1$  model. Since each node knows the identifier of the node on the other side of an incident edge, both endpoints can define a *common* edge identifier as the concatenation of identifiers of its endpoints, lowest identifier first.

The vertices are allowed to communicate through the edges of the graph  $G$ . It is assumed that communication is synchronous and occurs in discrete rounds (time steps). In each time step, each node  $v$  can send an arbitrary message of  $O(\log n)$  bits through each edge  $e = \{v, u\}$  incident to  $v$ , and each message arrives at  $u$  by the end of this time step. The weights of the edges are at most polynomial in the number of vertices  $n$ , and therefore the

weight of a single edge can be communicated in one time step. This model of distributed computation is called the CONGEST( $\log n$ ) model or simply the CONGEST model [22, 27]. We also assume that each vertex has access to the outcome of unbiased private coin flips. We assume that all nodes know  $n$ .

### 2.3 Underlying Algorithms

We use an algorithm called *TestOut* that was introduced by King et al. [16] in the context of computing MSTs in the  $\text{KT}_1$  model. Consider a tree  $T$  that is a subgraph of a graph  $G$ . The algorithm *TestOut* allows the nodes in  $T$  to determine whether there exists an *outgoing* edge, i.e., an edge that connects a node in  $V(T)$  with a node in  $V(G) \setminus V(T)$ . We also refer to an outgoing edge as an edge *leaving*  $T$ . Let  $u$  be a node in  $T$  that initiates an execution of *TestOut*. On a high level, *TestOut* simply performs a single *broadcast-and-echo* operation: First, the node  $u$  broadcasts a random hash function along  $T$ . Each node in  $T$  computes a single bit of information based on the hash function and the identifiers of the incident edges. The parity of these bits is then aggregated using an echo (or convergecast) along  $T$ . The parity is 1 with constant probability if there is an edge in  $G$  leaving  $T$ , and it is 0 otherwise. The algorithm is always correct if the parity is 1. The running time of the algorithm is  $O(\text{diam}(T))$  and it uses  $O(|V(T)|)$  messages.

The correctness probability of *TestOut* can be amplified to high probability by executing the algorithm  $O(\log n)$  times. Furthermore, *TestOut* can be combined with a binary search on the edge identifiers to find the identifier of an outgoing edge if it exists, which adds another multiplicative factor of  $O(\log n)$  to the running time and the number of messages used by the algorithm. Finally, the procedure can also be used to find the identifier of an outgoing edge with minimum weight in a weighted graph by again using binary search on the edge weights at the cost of another multiplicative  $O(\log n)$  factor. All of these algorithms can be generalized to work on a connected subgraph  $H$  that is not necessarily a tree: A node  $u \in V(H)$  first constructs a breadth-first search tree  $T$  in  $H$  and then executes one of the algorithms described above on  $T$ . We have the following theorems.

► **Theorem 1.** *Consider a connected subgraph  $H$  of a graph  $G$ . There is an algorithm *FindAny* that outputs the identifier of an arbitrary edge in  $G$  leaving  $H$  if there is such an edge and that outputs  $\emptyset$  otherwise, w.h.p. The running time of the algorithm is  $\tilde{O}(\text{diam}(H))$  and it uses  $\tilde{O}(|E(H)|)$  messages.*

► **Theorem 2.** *Consider a connected subgraph  $H$  of a weighted graph  $G$  with edge weights from  $\{1, \dots, \text{poly}(n)\}$ . There is an algorithm *FindMin* that outputs the identifier of a lightest edge in  $G$  leaving  $H$  if there is such an edge and that outputs  $\emptyset$  otherwise, w.h.p. The running time of the algorithm is  $\tilde{O}(\text{diam}(H))$  and it uses  $\tilde{O}(|E(H)|)$  messages.*

We also require an efficient leader election algorithm. The following theorem is a reformulation of Corollary 4.2 in [18].

► **Theorem 3.** *There is an algorithm that for any graph  $G$  elects a leader in  $O(\text{diam}(G))$  rounds while using  $\tilde{O}(|E(G)|)$  messages, w.h.p.*

## 3 Distributed Danner Construction

The distributed danner construction presented in Algorithm 1 uses a parameter  $\delta$  that controls a trade-off between the time and message complexity of the algorithm. At the same time the parameter controls a trade-off between the diameter and the size (i.e., the number of

**Algorithm 1** Distributed Danner Construction.

The algorithm constructs a danner  $H$ . Initially, we set  $V(H) \leftarrow V(G)$  and  $E(H) \leftarrow \emptyset$ .

1. Each node becomes a *center* with probability  $p = \min\{(c \log n)/n^\delta, 1\}$ , where  $c \geq 1$  is a constant determined in the analysis. Let  $C$  be the set of centers.
2. Each node  $v$  adds the edges leading to its  $\min\{\deg(v), n^\delta\}$  neighbors with the lowest identifiers to  $H$ .
3. Each low-degree node sends a message to all its neighbors to inform them about its low degree and whether it is a center. The remaining steps operate on the induced subgraphs  $\hat{G} \leftarrow G[V_{\text{high}} \cup C]$  and  $\hat{H} \leftarrow H[V_{\text{high}} \cup C]$ . Note that every node can deduce which of its neighbors lie in  $V_{\text{high}} \cup C$  from the messages sent by the low-degree nodes.
4. For  $i = 1$  to  $\log n$  do the following in parallel in each connected component  $K$  of  $\hat{H}$ .
  - a. Elect a leader using the algorithm from Theorem 3.
  - b. Use the algorithm FindAny from Theorem 1 to find an edge in  $\hat{G}$  leaving  $K$ . If such an edge exists, add it to  $H$  and  $\hat{H}$ .
  - c. Wait until  $T$  rounds have passed in this iteration before starting the next iteration in order to synchronize the execution between the connected components. The value of  $T$  is determined in the analysis.

edges) of the resulting danner. For Algorithm 1 we assume that  $\delta \in [0, 1)$ . We explicitly treat the case  $\delta = 1$  later on. We say a node  $u$  has *high degree* if  $\deg(u) \geq n^\delta$ . Otherwise, it has *low degree*. Let  $V_{\text{high}}$  and  $V_{\text{low}}$  be the set of high-degree and low-degree nodes, respectively.

We now turn to the analysis of Algorithm 1. We assume that the probability  $p$  defined in Step 1 is such that  $p < 1$  since for  $p = 1$  the analysis becomes trivial. Our first goal is to bound the diameter of any connected component  $K$  of  $\hat{H}$  (defined in Step 3 of Algorithm 1) during any iteration of the loop in Step 4. To achieve this goal, we first show two fundamental lemmas that allow us to bound the diameter of  $K$  in terms of its domination number  $\gamma(K)$  (see Section 2.1). The main observation behind Lemma 4 was also used by Feige et al. in [8].

► **Lemma 4.** *Let  $P$  be a shortest path in a graph  $G$ . For each node  $v \in V(G)$  it holds  $|\Gamma(v) \cap V(P)| \leq 3$ .*

**Proof.** We show the lemma by contradiction. Let  $P = (u_0, \dots, u_\ell)$  be a shortest path in  $G$ . Suppose there is a node  $v \in V(G)$  such that  $|\Gamma(v) \cap V(P)| \geq 4$ . Let  $u_i$  be the node in  $\Gamma(v) \cap V(P)$  with the lowest index in  $P$  and let  $u_j$  be the node in  $\Gamma(v) \cap V(P)$  with the highest index in  $P$ . Since  $|\Gamma(v) \cap V(P)| \geq 4$  at least two nodes lie between  $u_i$  and  $u_j$  in  $P$ . We distinguish two cases. If  $u_i = v$  or  $u_j = v$  then  $P' = (u_0, \dots, u_i, u_j, \dots, u_\ell)$  is a path in  $G$  such that  $|P'| \leq |P| - 2$ , which is a contradiction. Otherwise, the path  $P' = (u_0, \dots, u_i, v, u_j, \dots, u_\ell)$  is a path in  $G$  such that  $|P'| \leq |P| - 1$ , which is again a contradiction. ◀

► **Lemma 5.** *For a connected graph  $G$  it holds  $\text{diam}(G) < 3\gamma(G)$ .*

**Proof.** We show the lemma by contradiction. Suppose there is a shortest path  $P$  in  $G$  such that  $|P| \geq 3\gamma(G)$ . Let  $S$  be a dominating set in  $G$  with  $|S| = \gamma(G)$ . By definition, for each node  $u \in V(P)$  there is a node  $v \in S$  such that  $u \in \Gamma(v)$ . Since  $|V(P)| = |P| + 1 > 3|S|$ , the pigeonhole principle implies that there must be a node  $v \in S$  such that  $|\Gamma(v) \cap V(P)| > 3$ . By Lemma 4, this implies that  $P$  is not a shortest path, which is a contradiction. ◀

With these lemmas in place, we can now turn to the problem of bounding the diameter of a connected component  $K$  of  $\hat{H}$ . We first bound the number of centers established in Step 1.



► **Lemma 6.** *It holds  $|C| = \tilde{O}(n^{1-\delta})$ , w.h.p.*

**Proof.** Let  $X_u$  be a binary random variable such that  $X_u = 1$  if and only if  $u \in C$ . We have  $E[X_u] = (c \log n)/n^\delta$ . By definition it holds  $|C| = \sum_{u \in V} X_u$ . The linearity of expectation implies  $E[|C|] = \sum_{u \in V} E[X_u] = cn^{1-\delta} \log n$ . Since  $|C|$  is a sum of independent binary random variables we can apply Chernoff bounds (see, e.g., [20]) to get  $\Pr[|C| \geq 2cn^{1-\delta} \log n] \leq \exp(-cn^{1-\delta} \log n/3)$ . The lemma follows by choosing  $c$  sufficiently large. ◀

The next two lemmas show that the set of centers in  $K$  forms a dominating set of  $K$ .

► **Lemma 7.** *After Step 2 each high-degree node is adjacent to a center in  $\hat{H}$ , w.h.p.*

**Proof.** Consider a node  $u \in V_{\text{high}}$ . Let  $S$  be the set of the  $n^\delta$  neighbors of  $u$  with lowest identifier. Each node in  $S$  is a center with probability  $p$ . Hence, the probability that no node in  $S$  is a center is  $(1-p)^{|S|} = (1 - (c \log n)/n^\delta)^{n^\delta} \leq \exp(-c \log n)$ . The lemma follows by applying the union bound over all nodes and choosing the constant  $c$  sufficiently large. ◀

► **Lemma 8.** *Let  $K$  be a connected component of  $\hat{H}$  before any iteration of the loop in Step 4 or after the final iteration. The set of centers in  $K$  is a dominating set of  $K$ , w.h.p.*

**Proof.** Recall that  $V(K) \subseteq V_{\text{high}} \cup C$  by definition. Hence, each node  $u \in V(K)$  is a center or has high degree. If  $u$  is a center, there is nothing to show. If  $u$  is not a center, it must be of high degree. According to Lemma 7,  $u$  is connected to a center  $v$  in  $\hat{H}$ . This implies that  $v \in V(K)$  and  $\{u, v\} \in E(K)$ . ◀

By combining the statements of Lemmas 5, 6 and 8, we get the following lemma.

► **Lemma 9.** *Let  $K_1, \dots, K_r$  be the connected components of  $\hat{H}$  before any iteration of the loop in Step 4 or after the final iteration. It holds  $\sum_{i=1}^r \text{diam}(K_i) = \tilde{O}(n^{1-\delta})$ , w.h.p.*

**Proof.** Let  $C(K_i)$  be the set of centers in  $K_i$ . According to Lemma 8,  $C(K_i)$  is a dominating set of  $K_i$ . Therefore, Lemma 5 implies  $\text{diam}(K_i) < 3|C(K_i)|$ . This implies  $\sum_{i=1}^r \text{diam}(K_i) < 3 \sum_{i=1}^r |C(K_i)| = 3|C| = \tilde{O}(n^{1-\delta})$ , where the last equality holds according to Lemma 6. ◀

The following simple corollary gives us the desired bound on the diameter of a connected component  $K$  of  $\hat{H}$ .

► **Corollary 10.** *Let  $K$  be a connected component of  $\hat{H}$  before any iteration of the loop in Step 4 or after the final iteration. It holds  $\text{diam}(K) = \tilde{O}(n^{1-\delta})$ , w.h.p.*

On the basis of Corollary 10, we can bound the value of  $T$ , the waiting time used in Step 4c: Consider an iteration of the loop in Step 4. For each connected component  $K$  the leader election in Step 4a can be achieved in  $\tilde{O}(n^{1-\delta})$  rounds according to Theorem 3. The algorithm FindAny in Step 4b requires  $\tilde{O}(n^{1-\delta})$  rounds according to Theorem 1. Therefore, we can choose  $T$  such that  $T = \tilde{O}(n^{1-\delta})$ .

Our next objective is to show that the computed subgraph  $H$  is an additive  $\tilde{O}(n^{1-\delta})$ -danner. To this end, we first take a closer look at the connected components of  $\hat{H}$  after the algorithm terminates.

► **Lemma 11.** *After Algorithm 1 terminates, the set of connected components of  $\hat{H}$  equals the set of connected components of  $\hat{G}$ .*

**Proof.** Consider a connected component  $K_{\hat{G}}$  of  $\hat{G}$ . We show by induction that after iteration  $i$  of the loop in Step 4, each connected component of  $\hat{H}[V(K_{\hat{G}})]$  has size at least  $\min\{2^i, |V(K_{\hat{G}})|\}$ . Since  $|V(K_{\hat{G}})| \leq n$  and the loop runs for  $\log n$  iterations, this implies that after the algorithm terminates, only one connected components remains in  $\hat{H}[V(K_{\hat{G}})]$ .

The statement clearly holds before the first iteration of the loop, i.e., for  $i = 0$ . Suppose that the statement holds for iteration  $i \geq 0$ . We show that it also holds for iteration  $i + 1$ . If there is only one connected component at the beginning of iteration  $i + 1$  then that connected component must equal  $K_{\hat{G}}$  so the statement holds. If there is more than one connected component at the beginning of iteration  $i + 1$  then by the induction hypothesis each connected component has size at least  $2^i$ . Each connected component finds an edge leading to another connected component in Step 4b and thereby merges with at least one other connected component. The size of the newly formed component is at least  $\min\{2^{i+1}, |K_{\hat{G}}|\}$ . ◀

We are now ready to show that  $H$  is an additive  $\tilde{O}(n^{1-\delta})$ -danner.

► **Lemma 12.** *Algorithm 1 computes an additive  $\tilde{O}(n^{1-\delta})$ -danner  $H$  of  $G$ , w.h.p.*

**Proof.** Let  $P_G = (u_0, \dots, u_\ell)$  be a shortest path in  $G$ . We construct a path  $P_H$  from  $u_0$  to  $u_\ell$  in  $H$  such that  $|P_H| \leq |P_G| + \tilde{O}(n^{1-\delta})$ . Some of the edges in  $P_G$  might be missing in  $H$ . Let  $\{u_i, u_{i+1}\}$  be such an edge. Observe that if  $u_i$  or  $u_{i+1}$  has low degree then the edge  $\{u_i, u_{i+1}\}$  is contained in  $H$  since a low degree node adds all of its incident edges to  $H$  in Step 2. Hence,  $u_i$  and  $u_{i+1}$  must have high degree. Since the nodes share an edge in  $G$ , they lie in the same connected component of  $\hat{G}$ . According to Lemma 11 this means that the nodes also lie in the same connected component of  $\hat{H}$ . Therefore, there is a path in  $\hat{H}$  between  $u_i$  and  $u_{i+1}$ . We construct  $P_H$  from  $P_G$  by replacing each edge  $\{u_i, u_{i+1}\}$  that is missing in  $H$  by a shortest path from  $u_i$  to  $u_{i+1}$  in  $\hat{H}$ .

While  $P_H$  is a valid path from  $u_0$  to  $u_\ell$  in  $H$ , its length does not necessarily adhere to the required bound. To decrease the length of  $P_H$ , we do the following for each connected component  $K$  of  $\hat{H}$ : If  $P_H$  contains at most one node from  $K$ , we proceed to the next connected component. Otherwise, let  $v$  be the first node in  $P_H$  that lies in  $K$  and let  $w$  be the last node in  $P_H$  that lies in  $K$ . We replace the subpath from  $v$  to  $w$  in  $P_H$  by a shortest path from  $v$  to  $w$  in  $\hat{H}$ . After iteratively applying this modification for each connected component, the path  $P_H$  enters and leaves each connected component of  $\hat{H}$  at most once and within each connected component  $P_H$  only follows shortest paths. Hence, according to Lemma 9, the number of edges in  $P_H$  passing through  $\hat{H}$  is bounded by  $\tilde{O}(n^{1-\delta})$ . The remaining edges in  $P_H$  stem from  $P_G$ , so their number is bounded by  $|P_G|$ . In summary, we have  $|P_H| \leq |P_G| + \tilde{O}(n^{1-\delta})$ . ◀

To complete our investigation we analyze the time and message complexity of Algorithm 1 and bound the number of edges in the resulting danner  $H$ .

► **Lemma 13.** *The running time of Algorithm 1 is  $\tilde{O}(n^{1-\delta})$  and the number of messages sent by the algorithm is  $\tilde{O}(\min\{m, n^{1+\delta}\})$ . After the algorithm terminates it holds  $|E(H)| = \tilde{O}(\min\{m, n^{1+\delta}\})$ .*

**Proof.** We begin with the running time. The first three steps of the algorithm can be executed in a single round. The loop in Step 4 runs for  $\log n$  iterations, each of which takes  $T = \tilde{O}(n^{1-\delta})$  rounds.

Next we bound the number of edges in the danner. Steps 1 and 3 do not add any edges to  $H$ . Step 2 adds  $\tilde{O}(\min\{m, n^{1+\delta}\})$  edges to  $H$ . The loop in Step 4 runs for  $\log n$  iterations, and in every iteration each connected component of  $\hat{H}$  adds at most one edge to  $H$ . Since the number of connected components is at most  $n$  at all times, the total number of edges added in Step 4 is  $\tilde{O}(n)$ .



Finally, we turn to the message complexity of the algorithm. In Step 1 the nodes send no messages. The number of messages sent in Step 2 is  $\tilde{O}(\min\{m, n^{1+\delta}\})$ . In Step 3 each low-degree node sends a message to each of its neighbors. By definition a low-degree node has at most  $n^\delta$  neighbors and there are at most  $n$  low-degree nodes. Therefore, at most  $\tilde{O}(\min\{m, n^{1+\delta}\})$  messages are sent in this step. Each iteration of the loop in Step 4 operates on a subgraph  $\hat{H}$  of the final danner  $H$ . Consider a connected component  $K$  of  $\hat{H}$ . Both the leader election in Step 4a and the algorithm FindAny in Step 4b use  $\tilde{O}(|E(K)|)$  messages according to Theorems 3 and 1, respectively. Hence, the overall number of messages used in any iteration is  $\tilde{O}(|E(\hat{H})|)$  which is bounded by  $\tilde{O}(|E(H)|) = \tilde{O}(\min\{m, n^{1+\delta}\})$ . ◀

Finally, we treat the special case  $\delta = 1$ . In this case we do not use Algorithm 1 but instead let each node add all its incident edges to  $H$  such that  $H = G$ . Combining the statements of the previous two lemmas together with the special case of  $\delta = 1$  yields the following theorem.

► **Theorem 14.** *There is an algorithm that for a connected graph  $G$  and any  $\delta \in [0, 1]$  computes an additive  $\tilde{O}(n^{1-\delta})$ -danner  $H$  consisting of  $\tilde{O}(\min\{m, n^{1+\delta}\})$  edges, w.h.p. The algorithm takes  $\tilde{O}(n^{1-\delta})$  rounds and requires  $\tilde{O}(\min\{m, n^{1+\delta}\})$  messages.*

## 4 Applications

In this section we demonstrate that the danner construction presented in Section 3 can be used to establish trade-off results for many fundamental problems in distributed computing.

### 4.1 Broadcast, Leader Election, and Spanning Tree

On the basis of the danner construction presented in Section 3 it is easy to obtain a set of trade-off results for broadcast, leader election, and spanning tree construction. The number of messages required for a broadcast can be limited by first computing a danner and then broadcasting along the danner. For leader election we can run the algorithm of Kutten et al. [18] mentioned in Theorem 3 on the computed danner. Finally, for spanning tree construction we can elect a leader which then performs a distributed breadth-first search on the danner to construct the spanning tree. We have the following theorem.

► **Theorem 15.** *There are algorithms that for any connected graph  $G$  and any  $\delta \in [0, 1]$  solve the following problems in  $\tilde{O}(D + n^{1-\delta})$  rounds while using  $\tilde{O}(\min\{m, n^{1+\delta}\})$  messages, w.h.p.: broadcast, leader election, and spanning tree.*

### 4.2 Minimum Spanning Tree and Connectivity

In this section we assume that we are given a *weighted* connected graph  $G$  with edge weights from  $\{1, \dots, \text{poly}(n)\}$ . Without loss of generality we assume that the edge weights are distinct such that the MST is unique. We present a three step algorithm for computing the MST.

**Step 1.** We compute a spanning tree of  $G$  using the algorithm described in Section 4.1 while ignoring the edge weights. Recall that the algorithm computes the spanning tree by having a leader node initiate a distributed breadth-first search along a danner of diameter  $\tilde{O}(D + n^{1-\delta})$ . Therefore, the algorithm supplies us with a *rooted* spanning tree  $T$  of depth  $\tilde{O}(D + n^{1-\delta})$ . We aggregate the number of edges  $m$  in  $G$  using a convergecast along  $T$ . If  $m \leq n^{1+\delta}$ , we execute the singularly optimal algorithm of Pandurangan et al. [25] on the original graph  $G$  to compute the MST, which takes  $\tilde{O}(D + \sqrt{n})$  rounds and requires  $\tilde{O}(m)$  messages. Otherwise, we proceed with the following steps.

**Step 2.** We execute the so-called Controlled-GHS procedure on  $G$  as described in [25]. This procedure is a modified version of the classical Gallager-Humblet-Spira (GHS) algorithm for distributed MST [9]. It constructs a set of *MST fragments* (i.e., connected subgraphs of the MST). However, in contrast to the original GHS, Controlled-GHS limits the diameter of the fragments by controlling the way in which fragments are merged. By running Controlled-GHS for  $\lceil (1 - \delta) \log n \rceil$  iterations we get a spanning forest consisting of at most  $n^\delta$  MST-fragments, each having diameter  $O(n^{1-\delta})$  in  $\tilde{O}(n^{1-\delta})$  rounds. The Controlled-GHS described in [25] requires  $\tilde{O}(m)$  messages. However, we can reduce the number of messages to  $\tilde{O}(n)$  without increasing the running time by modifying the Controlled-GHS procedure to use the algorithm FindMin described in Theorem 2 to find the lightest outgoing edge of a fragment.

**Step 3.** Our goal in the final step of the algorithm is to merge the remaining  $n^\delta$  MST fragments quickly. This step executes the same procedure for  $\log n$  iterations. Each iteration reduces the number of fragments by at least a factor of two so that in the end only a single fragment remains, which is the MST.

We use a modified version of the algorithm TestOut that only communicates along the spanning tree  $T$  computed in Step 1 (i.e., it ignores the structure of the fragments) and that operates on all remaining fragments in parallel. Recall that the original TestOut algorithm for a single connected component consists of a broadcast-and-echo in which a leader broadcasts a random hash function and the nodes use an echo (or convergecast) to aggregate the parity of a set of bits. We can parallelize this behavior over all fragments as follows: Let  $v_T$  be the root of  $T$ . First,  $v_T$  broadcasts a random hash function through  $T$ . The same hash function is used for all fragments. Each node  $u$  uses the hash function to compute its individual bit as before and prepares a message consisting of the identifier of the fragment containing  $u$  and the bit of  $u$ . These messages are then aggregated up the tree in a pipelined fashion: In each round, a node sends the message with the lowest fragment identifier to its parent. Whenever a node holds multiple messages with the same fragment identifier, it combines them into a single message consisting of the same fragment identifier and the combined parity of the bits of the respective messages. Since  $T$  has depth  $\tilde{O}(D + n^{1-\delta})$  and there are at most  $n^\delta$  different fragment identifiers,  $v_T$  learns the aggregated parity of the bits in each individual fragment after  $\tilde{O}(D + n^{1-\delta} + n^\delta)$  rounds, which completes the parallelized execution of TestOut.

As explained in Section 2.3, a polylogarithmic number of executions of TestOut in combination with binary search can be used to identify the lightest outgoing edge of a fragment. The ranges for the binary search for each fragment can be broadcast by  $v_T$  in a pipelined fashion and the TestOut procedure can be executed in parallel for all fragments as described above. Thereby,  $v_T$  can learn the identifier of a lightest outgoing edge for each fragment in parallel. To merge the fragments,  $v_T$  does the following: First, it learns the fragment identifiers of the nodes at the end of each outgoing edge. It then locally computes the changes in the fragment identifiers that follow from the merges. Finally, it broadcasts these changes along with the identifiers of the leaving edges to merge the fragments. This completes one iteration of the procedure.

Overall, the operations of the final step can be achieved using a polylogarithmic number of pipelined broadcast-and-echo operations. Therefore, the running time of this step is  $\tilde{O}(D + n^{1-\delta} + n^\delta)$  rounds. In each pipelined broadcast-and-echo each node sends at most  $n^\delta$  messages, so the overall number of messages is  $\tilde{O}(n^{1+\delta})$ . This gives us the following theorem.

► **Theorem 16.** *There is an algorithm that for any connected graph  $G$  with edge weights from  $\{1, \dots, \text{poly}(n)\}$  and any  $\delta \in [0, 0.5]$  computes an MST of  $G$  in  $\tilde{O}(D + n^{1-\delta})$  rounds while using  $\tilde{O}(\min\{m, n^{1+\delta}\})$  messages, w.h.p.*

For  $\delta = 0.5$  we get an algorithm with optimal running time up to polylogarithmic factors.

► **Corollary 17.** *There is an algorithm that for any connected graph  $G$  with edge weights from  $\{1, \dots, \text{poly}(n)\}$  computes an MST of  $G$  in  $\tilde{O}(D + \sqrt{n})$  rounds while using  $\tilde{O}(\min\{m, n^{3/2}\})$  messages, w.h.p.*

Using this result on MST, it is not hard to devise an algorithm that computes the connected components of a subgraph  $H$  of  $G$  (and thus also test connectivity): We assign the weight 0 to each edge in  $E(H)$  and the weight 1 to each edge in  $E(G) \setminus E(H)$ . We then run a modified version of the above MST algorithm in which a fragment stops participating as soon as it discovers that its lightest outgoing edge has weight 1. Thereby, fragments only merge along edges in  $H$ . Once the algorithm terminates, any two nodes in the same connected component of  $H$  have the same fragment identifier while any two nodes in distinct connected components have distinct fragment identifiers.

► **Corollary 18.** *There is an algorithm that for any graph  $G$ , any subgraph  $H$  of  $G$ , and any  $\delta \in [0, 0.5]$  identifies the connected components of  $H$  in  $\tilde{O}(D + n^{1-\delta})$  rounds while using  $\tilde{O}(\min\{m, n^{1+\delta}\})$  messages, w.h.p.*

### 4.3 $O(\log n)$ -Approximate Minimum Cut

We describe an algorithm that finds an  $O(\log n)$ -approximation to the edge connectivity of the graph (i.e., the minimum cut value).

► **Theorem 19.** *There is a distributed algorithm for finding an  $O(\log n)$ -approximation to the edge connectivity of the graph (i.e., the minimum cut value) that uses  $\tilde{O}(\min\{m, n^{1+\delta}\})$  messages and runs in  $\tilde{O}(D + n^{1-\delta})$  rounds for any  $\delta \in [0, 0.5]$ , w.h.p.*

The main idea behind the algorithm is based on the following sampling theorem.

► **Theorem 20** ([14, 11]). *Consider an arbitrary unweighted multigraph<sup>10</sup>  $G = (V, E)$  with edge connectivity  $\lambda$  and choose subset  $S \subseteq E$  by including each edge  $e \in E$  in set  $S$  independently with probability  $p$ . If  $p \geq c \log n / \lambda$ , for a sufficiently large (but fixed) constant  $c$ , then the sampled subgraph  $G' = (V, S)$  is connected, w.h.p.*

We next sketch the distributed algorithm that is claimed in Theorem 19. The distributed algorithm implements the above sampling theorem which provides a simple approach for finding an  $O(\log n)$ -approximation of the edge connectivity of  $G$  by sampling subgraphs with exponentially growing sampling probabilities (e.g., start with an estimate of  $\lambda = m$ , the total number of (multi-)edges, and keep decreasing the estimate by a factor of 2) and checking the connectivity of each sampled subgraph. We take the first value of the estimate where the sampled graph is connected as the  $O(\log n)$ -approximate value. This algorithm can be easily implemented in the  $\text{KT}_0$  distributed model using  $\tilde{O}(m)$  messages and  $\tilde{O}(D + \sqrt{n})$  rounds by using the singularly optimal MST algorithm of [25] or [7] (which can be directly used to test connectivity – cf. Section 4.2). However, implementing the algorithm in the  $\text{KT}_1$  model in the prescribed time and message bounds of Theorem 19 requires some care, because of implementing the sampling step; each endpoint has to agree on the sampled edge without actually communicating through that edge.

<sup>10</sup>Note that a weighted graph with polynomially large edge weights can be represented as an unweighted graph with polynomial number of multiedges.

The sampling step can be accomplished by sampling with a *strongly  $O(\log n)$ -universal hash function*. Such a hash function can be created by using only  $O(\log n)$  independent shared random bits (see, e.g., [3]). The main insight is that a danner allows sharing of random bits efficiently. This can be done by constructing a danner<sup>11</sup> and letting the leader (which can be elected using the danner – cf. Section 4.1) generate  $O(\log n)$  independent random bits and broadcast them to all the nodes via the danner. The nodes can then construct and use the hash function to sample the edges. However, since the hash function is only  $O(\log n)$ -universal it is only  $O(\log n)$ -wise independent. But still one can show that the guarantees of Theorem 20 hold if the edges are sampled by a  $O(\log n)$ -wise independent hash function. This can be seen by using Karger’s proof [14] and checking that Chernoff bounds for  $O(\log n)$ -wise independent random variables (as used in [28]) are (almost) as good as that as (fully) independent random variables. Hence edge sampling can be accomplished using time  $\tilde{O}(D + n^{1-\delta})$  (the diameter of the danner) and messages  $\tilde{O}(\min\{m, n^{1+\delta}\})$  (number of edges of the danner). Once the sampling step is done, checking connectivity can be done by using the algorithm of Section 4.2.

#### 4.4 Algorithms for Graph Verification Problems

It is well known that graph connectivity is an important building block for several graph verification problems (see, e.g., [5]). Thus, using the connectivity algorithm of Section 4.2 as a subroutine, we can show that the problems stated in the theorem below (these are formally defined, e.g., in Section 2.4 of [5]) can be solved in the  $KT_1$  model (see, e.g., [5, 24]).

► **Theorem 21.** *There exist distributed algorithms in the  $KT_1$  model that solve the following verification problems in  $\tilde{O}(\min\{m, n^{1+\delta}\})$  messages and  $\tilde{O}(D + n^{1-\delta})$  rounds, w.h.p, for any  $\delta \in [0, 0.5]$  : spanning connected subgraph, cycle containment, e-cycle containment, cut, s-t connectivity, edge on all paths, s-t cut, bipartiteness.*

## 5 Conclusion

This work is a step towards understanding time-message trade-offs for distributed algorithms in the  $KT_1$  model. Using our danner construction, we obtained algorithms that exhibit time-message trade-offs across the spectrum by choosing the parameter  $\delta$  as desired. There are many key open questions raised by our work. First, it is not clear whether one can do better than the algorithms we obtained here for various fundamental problems in the  $KT_1$  model. In particular, it would be interesting to know whether there are singularly optimal algorithms in the  $KT_1$  model – e.g., for leader election, can we show an  $\tilde{O}(n)$  messages algorithm that runs in  $\tilde{O}(D)$  (these are respective lower bounds for messages and time in  $KT_1$ ); and, for MST, can we show an  $\tilde{O}(n)$  messages algorithm that runs in  $\tilde{O}(D + \sqrt{n})$ . A related question is whether one can construct an  $(\alpha, \beta)$ -danner with  $\tilde{O}(n)$  edges and  $\alpha, \beta = \tilde{O}(1)$  in a distributed manner using  $\tilde{O}(n)$  messages and in  $\tilde{O}(D)$  time. Such a construction could be used to obtain singularly optimal algorithms for other problems. Finally, our danner construction is randomized; a deterministic construction with similar guarantees will yield deterministic algorithms.

---

<sup>11</sup>Note that the danner of a multigraph is constructed by treating multi-edges as a single edge.

## References

- 1 Donald Aingworth, Chandra Chekuri, Piotr Indyk, and Rajeev Motwani. Fast estimation of diameter and shortest paths (without matrix multiplication). *SIAM J. Comput.*, 28(4):1167–1181, 1999. doi:10.1137/S0097539796303421.
- 2 Baruch Awerbuch, Oded Goldreich, David Peleg, and Ronen Vainish. A trade-off between information and communication in broadcast protocols. *J. ACM*, 37(2):238–256, 1990.
- 3 Larry Carter and Mark N. Wegman. Universal classes of hash functions. *J. Comput. Syst. Sci.*, 18(2):143–154, 1979.
- 4 Yongwook Choi, Gopal Pandurangan, Maleq Khan, and V. S. Anil Kumar. Energy-optimal distributed algorithms for minimum spanning trees. *IEEE Journal on Selected Areas in Communications*, 27(7):1297–1304, 2009.
- 5 Atish Das Sarma, Stephan Holzer, Liah Kor, Amos Korman, Danupon Nanongkai, Gopal Pandurangan, David Peleg, and Roger Wattenhofer. Distributed verification and hardness of distributed approximation. *SIAM J. Comput.*, 41(5):1235–1265, 2012.
- 6 Dorit Dor, Shay Halperin, and Uri Zwick. All-pairs almost shortest paths. *SIAM J. Comput.*, 29(5):1740–1759, 2000. doi:10.1137/S0097539797327908.
- 7 Michael Elkin. A simple deterministic distributed MST algorithm, with near-optimal time and message complexities. In *Proceedings of the 2017 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 157–163, 2017.
- 8 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.
- 9 Robert G. Gallager, Pierre A. Humblet, and Philip M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. Program. Lang. Syst.*, 5(1):66–77, 1983. doi:10.1145/357195.357200.
- 10 J. Garay, S. Kutten, and D. Peleg. A sublinear time distributed algorithm for minimum-weight spanning trees. *SIAM Journal on Computing*, 27(1):302–316, February 1998.
- 11 Mohsen Ghaffari and Fabian Kuhn. Distributed minimum cut approximation. In *Distributed Computing - 27th International Symposium, DISC 2013, Jerusalem, Israel, October 14-18, 2013. Proceedings*, pages 1–15, 2013.
- 12 Bernhard Haeupler, D. Ellis Hershkowitz, and David Wajc. Round- and message-optimal distributed graph algorithms. In *PODC*, 2018.
- 13 James W. Hegeman, Gopal Pandurangan, Sriram V. Pemmaraju, Vivek B. Sardeshmukh, and Michele Scquizzato. Toward optimal bounds in the congested clique: graph connectivity and MST. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 91–100, 2015.
- 14 David R. Karger. Random sampling in cut, flow, and network design problems. *Math. Oper. Res.*, 24(2):383–413, 1999.
- 15 Maleq Khan, Gopal Pandurangan, and V. S. Anil Kumar. Distributed algorithms for constructing approximate minimum spanning trees in wireless sensor networks. *IEEE Trans. Parallel Distrib. Syst.*, 20(1):124–139, 2009.
- 16 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, PODC 2015, Donostia-San Sebastián, Spain, July 21 - 23, 2015*, pages 71–80, 2015. doi:10.1145/2767386.2767405.
- 17 Hartmut Klauck, Danupon Nanongkai, Gopal Pandurangan, and Peter Robinson. Distributed computation of large-scale graph problems. In *Proceedings of the 26th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 391–410, 2015.
- 18 Shay Kutten, Gopal Pandurangan, David Peleg, Peter Robinson, and Amitabh Trehan. On the complexity of universal leader election. *J. ACM*, 62(1), 2015.

- 19 Ali Mashreghi and Valerie King. Time-communication trade-offs for minimum spanning tree construction. In *Proceedings of the 18th International Conference on Distributed Computing and Networking, Hyderabad, India, January 5-7, 2017*, page 8, 2017. URL: <http://dl.acm.org/citation.cfm?id=3007775>.
- 20 Michael Mitzenmacher and Eli Upfal. *Probability and computing - randomized algorithms and probabilistic analysis*. Cambridge University Press, 2005.
- 21 Danupon Nanongkai and Hsin-Hao Su. Almost-tight distributed minimum cut algorithms. In *Distributed Computing - 28th International Symposium, DISC 2014, Austin, TX, USA, October 12-15, 2014. Proceedings*, pages 439–453, 2014.
- 22 Gopal Pandurangan. Distributed network algorithms. Draft, 2018. URL: <https://sites.google.com/site/gopalpandurangan/dna>.
- 23 Gopal Pandurangan and Maleq Khan. Theory of communication networks. In *Algorithms and Theory of Computation Handbook*. CRC Press, 2009.
- 24 Gopal Pandurangan, Peter Robinson, and Michele Scquizzato. Fast distributed algorithms for connectivity and MST in large graphs. 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 429–438, 2016.
- 25 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, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 743–756, 2017. doi:10.1145/3055399.3055449.
- 26 Gopal Pandurangan, Peter Robinson, and Michele Scquizzato. On the distributed complexity of large-scale graph computations. In *Proceedings of the 30th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA, 2018*.
- 27 D. Peleg. *Distributed Computing: A Locality Sensitive Approach*. SIAM, 2000.
- 28 Jeanette P. Schmidt, Alan Siegel, and Aravind Srinivasan. Chernoff-hoeffding bounds for applications with limited independence. *SIAM J. Discrete Math.*, 8(2):223–250, 1995.



# Faster Distributed Shortest Path Approximations via Shortcuts

Bernhard Haeupler<sup>1</sup>

Carnegie Mellon University, USA

<http://cs.cmu.edu/~haeupler>

Jason Li

Carnegie Mellon University, USA

<http://cs.cmu.edu/~jqli>

---

## Abstract

---

A long series of recent results and breakthroughs have led to faster and better distributed approximation algorithms for single source shortest paths (SSSP) and related problems in the CONGEST model. The runtime of all these algorithms, however, is  $\tilde{\Omega}(\sqrt{n})$ , regardless of the network topology<sup>2</sup>, even on nice networks with a (poly)logarithmic network diameter  $D$ . While this is known to be necessary for some pathological networks, most topologies of interest are arguably not of this type.

We give the first distributed approximation algorithms for shortest paths problems that adjust to the topology they are run on, thus achieving significantly faster running times on many topologies of interest. The running time of our algorithms depends on and is close to  $Q$ , where  $Q$  is the quality of the best shortcut that *exists* for the given topology. While  $Q = \tilde{\Theta}(\sqrt{n} + D)$  for pathological worst-case topologies, many topologies of interest<sup>3</sup> have  $Q = \tilde{\Theta}(D)$ , which results in near *instance optimal* running times for our algorithm, given the trivial  $\Omega(D)$  lower bound.

The problems we consider are as follows:

- an approximate shortest path tree and SSSP distances,
- a polylogarithmic size distance label for every node such that from the labels of any two nodes alone one can determine their distance (approximately), and
- an (approximately) optimal flow for the transshipment problem.

Our algorithms have a tunable tradeoff between running time and approximation ratio. Our fastest algorithms have an arbitrarily good polynomial approximation guarantee and an essentially optimal  $\tilde{O}(Q)$  running time. On the other end of the spectrum, we achieve polylogarithmic approximations in  $\tilde{O}(Q \cdot n^\epsilon)$  rounds for any  $\epsilon > 0$ . It seems likely that eventually, our non-trivial approximation algorithms for the SSSP tree and transshipment problem can be bootstrapped to give fast  $Q \cdot 2^{O(\sqrt{\log n \log \log n})}$  round  $(1 + \epsilon)$ -approximation algorithms using a recent result by Becker et al.

**2012 ACM Subject Classification** Theory of computation → Shortest paths, Theory of computation → Distributed algorithms, Theory of computation → Approximation algorithms analysis

**Keywords and phrases** Distributed Graph Algorithms, Shortest Path, Shortcuts

**Digital Object Identifier** 10.4230/LIPIcs.DISC.2018.33

**Related Version** A full version of the paper is available at <https://arxiv.org/abs/1802.03671>.

---

<sup>1</sup> Supported in part by NSF grants CCF-1527110, CCF-1618280 and NSF CAREER award CCF-1750808.

<sup>2</sup> We use  $\tilde{\cdot}$ -notation to hide polylogarithmic factors in  $n$ , e.g.,  $\tilde{O}(f(n)) = O(f(n) \log^{O(1)} n)$ .

<sup>3</sup> For example, [8] and [10] show that large classes of interesting network topologies, including planar networks, bounded genus topologies, and networks with polylogarithmic treewidth have shortcuts of quality  $Q = \tilde{O}(D)$ . A similar statement is likely to hold for any minor closed graph family [11].



© Bernhard Haeupler and Jason Li;

licensed under Creative Commons License CC-BY

32nd International Symposium on Distributed Computing (DISC 2018).

Editors: Ulrich Schmid and Josef Widder; Article No. 33; pp. 33:1–33:14

Leibniz International Proceedings in Informatics



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



## 1 Introduction

This paper gives new distributed approximation algorithms for computing single source shortest path (SSSP) distances and various generalizations, such as computing a SSSP tree, distance labels, and a min-cost uncapacitated flow.

In the last few years, CONGEST algorithms for shortest path problems have seen a tremendous amount of interest and progress [5, 12, 17]. The main difference of the algorithms developed here, compared to those works, is that our algorithms achieve significantly faster running times for non-pathological network topologies by building on the recently developed [8, 9] low-congestion shortcut framework; for a detailed overview, see Appendix A of the full version on arXiv.

The low-congestion shortcut framework leads to faster algorithms for optimization problems with simple parallel divide and conquer style algorithms, such as the minimum spanning tree problem. However it initially seemed less applicable to shortest path problems, particularly because all previous approaches for CONGEST algorithms for these problems led to  $\Omega(\sqrt{n})$  running times, for reasons that are independent of issues where shortcuts can help. Indeed, our approach for achieving non-trivial approximation ratios for shortest path problems deviates notably from these approaches, and uses different tools to obtain non-trivial approximation guarantees.

This paper is organized as follows: We briefly summarize the key technical concepts of the shortcut framework in Section 1.1; a more detailed treatment of the framework is given in Appendix A in the full version. In Section 1.2, we define the different problems we treat in this paper, and explain the difficulties in beating the  $\tilde{\Omega}(\sqrt{n} + D)$  barrier for approximating shortest path distances. We state our results in Section 1.3, compare it to related works in Section 1.4, and devote the remaining paper to describing our algorithms and proving them correct.

### 1.1 The Low-Congestion Shortcut Framework: A Brief Summary

This section provides the key technical definitions and facts about the low-congestion shortcut framework. However, it does not attempt to explain the reasons, generality or importance behind the definitions given here. Appendix A of the full version gives a more detailed treatment, and we highly recommend to readers not familiar with the low-congestion shortcut framework to read Appendix A first.

The shortcut framework is built around a simple and basic communication problem, given in the next two definitions:

► **Definition 1** (Valid Partitioning and Parts). For a graph  $G = (V, E)$ , we say that a collection of parts  $S_1, S_2, \dots \subset V$  is a **valid partition** if the parts are vertex disjoint and each induces a connected graph.

► **Definition 2** (The Part-wise Communication Problem). Let  $G$  be a network with a valid partitioning  $S_1, S_2, \dots$  and a value  $x_v$  for every node  $v \in V$ . Suppose  $\oplus$  is an associative and commutative function. The **partwise communication problem** asks for every  $S_i$  and every  $u \in S_i$  to compute the value  $\bigoplus_{v \in S_i} x_v$ .

We remark that for convenience, the parts of a valid partition do not necessarily need to contain every vertex in  $V$ . Alternatively, it can be convenient to think of each node

in  $V \setminus \bigcup_i S_i$  as forming its own single-vertex part, thus making any valid partitioning a partitioning in the usual sense.

The key findings of the shortcut framework can now be summarized as follows:

The shortcut framework allows us to characterize how hard it is to solve the part-wise communication problem described in Definition 2 in the CONGEST model for any given topology  $G$ . For any network with topology  $G$  this is captured by the quantity  $Q_G$ . In the worst-case, the value of  $Q_G$  is  $\tilde{\Theta}(\sqrt{n} + D)$  for a network with  $n$  nodes and diameter  $D$ , such as the pathological network that shows a  $\tilde{\Omega}(\sqrt{n} + D)$  lower bound for MST and related problems [19]. In many other networks of interest, including planar networks, networks which embed into a surface with bounded or polylogarithmic genus, networks with bounded or polylogarithmic tree-width or networks with small separators, the hardness  $Q_G$  is much lower and in fact only  $\tilde{O}(D)$ . Most importantly, whatever the hardness  $Q_G$  of a given topology is, there is a simple distributed algorithm which solves the part-wise communication problem in  $\tilde{O}(Q_G)$  rounds for any valid partitioning in  $G$ . Thus,  $\tilde{O}(Q_G)$  round shortcut-based algorithms necessarily have a worst-case running time of  $\tilde{O}(\sqrt{n} + D)$  when expressed in terms of  $n$  and  $D$ ; however, they are essentially running as fast as the given topology (and to some extent even the given input) allows it, which in many cases of interest is significantly faster, e.g.,  $\tilde{O}(D)$  rounds.

## 1.2 CONGEST model and Shortest Path Problems

### 1.2.1 CONGEST Model

We consider the classical CONGEST model of distributed computing where a network is given by a connected graph  $G = (V, E)$  with  $n$  nodes and (hop-)diameter  $D$ . Communication proceeds in synchronous rounds. In each round, each node can send a different  $O(\log n)$  bit message to each of its neighbors. Local computations are free and require no time. Nodes have no initial knowledge of the topology  $G$ , except that we assume that they know  $n$  and  $D$  up to constants (because these parameters can be computed in  $O(D)$  time, which is negligible in our context). All of our algorithms are randomized and succeed with high probability<sup>4</sup>. In particular, we assume that each node has access to a private string of randomness, which it can also use to create an  $O(\log n)$  bit ID that is unique w.h.p.

In all problems considered here, we assume that every edge  $e$  of the network  $G$  has a length or cost  $w(e)$  associated with it. We assume that all lengths lie in the range  $[1, n^C]$  for some constant  $C$ , and are initially only known to nodes adjacent to an edge. Interestingly, our algorithms also easily handle edges of length zero, but for sake of simplicity, we do not consider such edges in this paper. Any such length or cost function  $w$  produces a weighted graph which we call  $G(w)$ , and induces a distance between any two nodes  $u, v \in V$ , which we denote with  $d_{G(w)}(u, v)$ , or simply  $d(u, v)$  when the weighted graph  $G(w)$  is clear. We denote the weighted diameter of a network with  $L = \max_{u, v} d_G(u, v)$ .

### 1.2.2 Shortest Path Problems

The most important and most basic problem we are studying in this paper is the single source shortest path problem:

<sup>4</sup> Throughout this work, “with high probability” or w.h.p. means with probability at least  $1 - n^{-C}$  for any desired constant  $C$ .

► **Definition 3.** The  $\alpha$ -approximate SSSP distance problem assumes as input a weighted graph and a designated source node  $s \in V$ , and asks for every node  $v \in V$  to compute an approximate distance  $d_v$  which satisfies  $d(s, v) \leq d_v \leq \alpha \cdot d(s, v)$ .

We furthermore consider the following generalizations of the SSSP distance problem:

► **Definition 4.** The  $\alpha$ -approximate SSSP tree problem assumes that a weighted graph with a designated source node  $s \in V$  is given and asks to compute a subtree  $T \subseteq G$  such that for every node  $v \in V$  distance  $d_T(s, v) \leq \alpha \cdot d(s, v)$ . Each node should know which of its adjacent edges belong to  $T$ .

► **Definition 5** (Approximate distance labeling scheme). An  $(l(n), \alpha)$ -approximate distance labeling scheme is a function that labels the vertices of an input graph with distinct labels up to  $l(n)$  bits, such that there exists a polynomial time algorithm that, given the labels of vertices  $x$  and  $y$ , provides an estimate  $\tilde{d}(x, y)$  for the distance between these vertices such that

$$\tilde{d}(x, y) \leq d(x, y) \leq \alpha \cdot \tilde{d}(x, y).$$

► **Definition 6** (Transshipment Problem). The transshipment problem is the problem of uncapacitated min-cost flow. In it every node in a weighted graph  $G$  has some real demand  $d_v$  such that  $\sum_v d_v = 0$ . The cost of routing  $x$  amount of flow over an edge  $e$  of weight  $w(e)$  is  $xw(e)$ . The problem is to compute a flow satisfying all demands of approximate minimum cost. Each node should know the flow on all edges incident to it.

## 1.3 Our Results

### 1.3.1 SSSP

Our first result is on computing an approximate, single source shortest path tree in a distributed setting. Note that due to communication limits in the CONGEST model, it is infeasible for each vertex to know the entire shortest path tree. However, it is sufficient that each vertex computes the *local* structure of the tree, which is made specific below.

► **Theorem 7.** *Let  $G$  be a network graph with edge weights in  $[1, \text{poly}(n)]$ , with a specified source vertex, and let  $\beta := (\log n)^{-\Omega(1)}$ . There is a distributed algorithm that, w.h.p., runs for  $\tilde{O}(\frac{1}{\beta} Q_G)$  rounds and outputs a spanning tree that approximates distances to the source to factor  $O(L^{O(\log \log n)/\log(1/\beta)})$ .<sup>5</sup> By output, we mean that at the end of the algorithm, every vertex knows its set of incident edges in the spanning tree.*

By setting  $\beta := n^{-\epsilon}$ ,  $\beta := 2^{-\Theta(\sqrt{\log n})}$ , and  $\beta := \log^{-\Theta(1/\epsilon)} n$  for constant  $\epsilon$ , respectively, we obtain the following three corollaries:

► **Corollary 8.** *Let  $G$  be a network graph with edge weights in  $[1, \text{poly}(n)]$ , with a specified source vertex. For any constant  $\epsilon > 0$ , there is a distributed algorithm that, w.h.p., runs for  $\tilde{O}(Q_G n^\epsilon)$  rounds and outputs a spanning tree that approximates distances to the source to factor  $\text{polylog}(n)$ .*

► **Corollary 9.** *Let  $G$  be a network graph with edge weights in  $[1, \text{poly}(n)]$ , with a specified source vertex. There is a distributed algorithm that, w.h.p., runs for  $\tilde{O}(Q_G 2^{O(\sqrt{\log n})})$  rounds and outputs a spanning tree that approximates distances to the source to factor  $2^{O(\sqrt{\log n})}$ .*

<sup>5</sup> Recall that  $L = \max_{u,v} d_G(u, v)$ .

► **Corollary 10.** *Let  $G$  be a network graph with edge weights in  $[1, \text{poly}(n)]$ , with a specified source vertex. For any constant  $\epsilon > 0$ , there is a distributed algorithm that, w.h.p., runs for  $\tilde{O}(Q_G)$  rounds and outputs a spanning tree that approximates distances to the source to factor  $O(L^\epsilon)$ .*

### 1.3.2 Distance labeling schemes

For distance labeling schemes, we have the following result.

► **Theorem 11.** *Let  $G$  be a network graph with edge weights in  $[1, \text{poly}(n)]$ . There exists a  $(\text{polylog}(n), n^{O(\log \log n)/\log(1/\beta)})$  approximate distance labeling scheme that runs in  $\tilde{O}(\frac{1}{\beta}Q_G)$  rounds.*

Setting  $\beta := n^\epsilon$  gives the following corollary:

► **Corollary 12.** *Let  $G$  be a network graph with edge weights in  $[1, \text{poly}(n)]$ . There exists a  $(\text{polylog}(n), \text{polylog}(n))$  approximate distance labeling scheme that runs in  $\tilde{O}(Q_G n^\epsilon)$  rounds.*

### 1.3.3 Transshipment problem

We also provide a distributed algorithm to compute an approximate flow for the transshipment problem.

► **Theorem 13.** *Let  $G$  be a network graph with edge weights in  $[1, \text{poly}(n)]$  and demands that sum to zero, and let  $\beta := (\log n)^{-\Omega(1)}$ . There is an algorithm that, w.h.p., runs for  $\tilde{O}(\frac{1}{\beta}Q_G)$  rounds and computes a  $\tilde{O}(\frac{1}{\beta}n^{O(\log \log n)/\log(1/\beta)})$ -approximate flow.*

## 1.4 Related Work

The complexity theoretic issues in the design of distributed graph algorithms for the CONGEST model have received much attention in the last decade, and extensive progress has been made for many problems: Minimum-Spanning Tree [13], Minimum Cut [18], Diameter [14], Shortest Path [5], and so on. Most of those problems have  $\tilde{\Theta}(\sqrt{n} + D)$ -round upper and lower bounds for some sort of approximation guarantee [19]. The notion of low-congestion shortcuts was invented as a framework of circumventing these lower bounds [8]. Specifically, the ideas present in [8] can be turned into very short and clean  $\tilde{O}(D + \sqrt{n})$  round algorithms for general graphs, and near-optimal  $\tilde{O}(D)$  round algorithms for special classes of graphs, for problems such as MST and Min-Cut.

However, the shortcut framework cannot be applied directly to the SSSP problem, since, unlike MST and Min-Cut, shortest path problems are not inherently parallelizable. For SSSP, a new technique based on multiplicative weights results in a  $(1 + \epsilon)$ -approximation to SSSP in  $\tilde{O}(D + \sqrt{n})$  time on general graphs [5]. However, until this paper, not much work has been done on circumventing the  $\tilde{\Omega}(D + \sqrt{n})$  lower bound on restricted classes of graphs or otherwise.

As a subroutine to computing shortest paths, we will be running low-diameter graph decompositions. Low diameter decompositions have a long history in the centralized [4, 15] and parallel [3, 6, 16] settings, and have been applied in the distributed setting to compute a network decomposition with low “chromatic number” [7].

## 2 Distance-Preserving Tree

Let  $G$  be a weighted graph with  $Q_G$ -quality shortcuts. For a reader not familiar with shortcuts or the material in Appendix A of the full version, the parameter  $Q_G$  intuitively

measures how easy it is for connected components of  $G$  to communicate within each other. As a general rule, the “nicer” the graph  $G$  is, the smaller the quantity  $Q_G$  and the closer it gets to the optimal  $D$ . For example, if  $G$  is a planar graph, then  $Q_G = \tilde{O}(D)$ .

We first consider the problem of finding a tree such that, for every pair of vertices  $x, y \in V$ , their distance is well-approximated with constant probability. Our algorithm is an adaptation of the algorithm of Section 5.4 from [2].

To motivate the ideas behind the algorithm, we describe it in a parallel framework with graph contraction support. In each iteration, the algorithm runs a low diameter decomposition (defined below; see Appendix B of the full version for details) on the graph and contracts each component into a single vertex. To compute the tree as described above, take the set of edges inside the BFS trees formed by each LDD, and map them back to the original graph. The resulting tree is simply the (disjoint) union of these edges over all iterations. Of course, in a distributed framework, we cannot maintain contracted graphs, so we substitute each contracted vertex with a part of the original graph with zero-weight edges inside. To communicate efficiently between the parts, we establish shortcuts within each part.

► **Definition 14.** For a weighted graph  $G = (V, E)$ , a low-diameter decomposition (LDD) of  $G$  is a probabilistic distribution over partitions of  $V$  into connected components  $S_1, \dots, S_k$ , such that

1. W.h.p., every induced graph  $G[S_i]$  has low weighted diameter.
2. For every two vertices  $x, y \in V$ , the probability that they belong to the same component is bounded from below by some function depending on  $d_G(x, y)$ .

We now describe the algorithm in detail. For a weight function  $w : E \rightarrow \mathbb{R}$ , denote  $G(w)$  to be the graph  $G$  whose edges are reweighted according to  $w$ . The algorithm maintains a weight function  $w : E \rightarrow \{0\} \cup [R, \text{poly}(n)]$  on the set of edges, for a given value  $R$ . The zero-weight edges connect vertices within each component, while the threshold  $R$  increases geometrically over time. With a larger threshold  $R$ , we can compute the LDD on  $G(\frac{1}{R}w)$ , allowing the LDD to travel farther in the same amount of time. If  $R$  is large enough, this graph still has edge weights at least 1 in between components, so computing the LDD is feasible in a distributed manner.

In addition to  $w$ , the algorithm also maintains a forest  $T$ , which gets new edges every iteration until it results in the approximate shortest path tree. Consider the following `LDDSubroutine`, which we apply iteratively to  $w$  and  $T$ .

**Algorithm  $(w', T') = \text{LDDSubroutine}(w, T, \beta, R)$**

Algorithm:

1. Initially, set  $w' := w$  and  $T' := T$ .
2. Consider  $G_0(w)$ , the subgraph of  $G$  with only the edges  $e$  with  $w(e) = 0$ .
3. Let  $H$  be the (multi-)graph with every connected component of  $G_0(w)$  contracted to a single vertex. Denote  $w_H$  as the function  $w$  restricted to the edges in  $H$ .
4. Simulate a LDD on  $H(\frac{1}{R}w_H)$  with parameter  $\frac{1}{\beta}$  (see Appendix C of the full version). The specifics are deferred to the next section.
5. For every edge in  $H$  that is part of a BFS tree in the LDD, add that edge to  $T'$ .
6. For every edge  $e$  in  $H$  completely inside a LDD component, set  $w'(e) := 0$ .
7. For every other edge  $e$  in  $H$ , set  $w'(e) := w(e) + \frac{c_1}{\beta} \log n$  (for large enough constant  $c_1$ ).
8. Output  $(w', T')$ .

## 2.1 Correctness

The following two lemmas bound the maximum weighted diameter of a component, and therefore also the running time of the subroutine, as well as the probability that two vertices close together belong to the same component. Their proofs are natural generalizations of those in [16] and appear in Appendix C of the full version.

► **Lemma 15.** *W.h.p., each component in `LowDiameterDecomposition` has weighted diameter  $O(\frac{1}{\beta} \log n)$ .*

► **Lemma 16.** *For vertices  $u, v \in V$  of (weighted) distance  $d$ , the probability that  $u$  and  $v$  belong to the same component is  $e^{-O(d\beta)}$ .*

We now describe in more detail how to simulate the LDD in  $H(\frac{1}{R}w_H)$  in the desired running time. Observe that we cannot directly compute the LDD on the contracted graph, since the contracted vertices are actually entire parts with limited communication between them. However, we can apply shortcuts to communicate quickly within the parts, up to the quality of the shortcut.

► **Lemma 17.** *The LDD on the contracted graph (step 4 of `LDDSubroutine`) can be simulated with a  $\tilde{O}(Q_G)$  multiplicative overhead in running time. In other words, if the LDD takes  $d$  rounds, then it can be simulated in  $\tilde{O}(Q_G d)$  rounds in the network  $G$ .*

**Proof.** Define the parts of  $V$  to be the connected components of  $G$ , and compute a set of  $\tilde{O}(Q_G)$ -quality shortcuts, one for each part. In every round of the LDD on  $H(\frac{1}{R}w_H)$ , we perform two steps sequentially: one to traverse nonzero weight edges between parts, and one to flood through the zero weight edges within each part. To take care of the edges between parts, note that every such edge has weight at least 1, so we can send them directly through the network  $G$ . To flood through the zero edges within each part, it suffices to compute the minimum time  $t$  that is received by any vertex, and then broadcast the message “ $t$ ” to the entire part. By routing through shortcuts, this can be done in  $\tilde{O}(Q_G)$  time per partition. Overall, every round of the LDD is replaced by  $\tilde{O}(Q_G)$  rounds in the network  $G$ , hence the multiplicative overhead. ◀

Together with Lemma 15, we get a running time of  $\tilde{O}(\frac{1}{\beta}Q_G)$ .

► **Definition 18.** Let  $w : E \rightarrow \mathbb{R}$  be a weight function, and  $T \subseteq G$  a forest. Define  $G_0(w)$  to be the subgraph of  $G$  with only the edges  $e$  with  $w(e) = 0$ . Let  $C_1, C_2, \dots$  of  $G$  be the connected components of  $G_0(w)$ . We say that  $(w, T)$  satisfies the **subroutine invariant with parameter  $R$**  if the following conditions hold:

1. The weighted diameter of each part  $C_i$  using edge weights in  $G$  is at most  $R$ .
2. Every edge within a part  $C_i$  has weight 0 in  $w$ .
3. Every edge between two parts  $C_i, C_j$  has weight at least  $R$  in  $w$ .
4. For all  $x, y$  belonging to the same part  $C_i$ ,  $d_T(x, y) \leq R$ .
5.  $T$  has a spanning tree within each part  $C_i$ , and no edges in between parts.

► **Lemma 19.** *Fix parameter  $\beta$ . Suppose that the input  $(w, T)$  to `LDDSubroutine` satisfies the subroutine invariant with parameter  $R$ . Then, w.h.p., for large enough constants  $c_1$  and  $c_2$ ,*

- *The output  $(w', T')$  satisfies the subroutine invariant with parameter  $(\frac{c_1}{\beta} \log n)R$ .*
- *For all  $x, y \in V$ ,  $\mathbb{E}[d_{G(w')}(x, y)] \leq (c_2 \log n)d_{G(w)}(x, y)$ .*

**Proof.** Note that the following properties of the invariant follow immediately:

2. Every edge within a part  $C'_i$  has weight 0 in  $w'$ .
3. Every edge between two parts  $C'_i, C'_j$  has weight at least  $(\frac{c_1}{\beta} \log n)R$  in  $w'$ .
5.  $T'$  has a spanning tree within each part  $C'_i$ , and no edges in between parts.

To prove invariant (4), suppose that  $x, y \in V$  are in the same  $C'_i$ . If they are also in the same  $C_i$ , then the property holds by the input guarantee. Otherwise, by Lemma 15, w.h.p. the parts containing  $x$  and  $y$  have distance  $O(\frac{1}{\beta} \log n)$  in the BFS tree on  $H(\frac{1}{R}w_H)$ , which means that there is a path in the BFS tree that travels through  $O(\frac{1}{\beta} \log n)$  vertices in  $H(\frac{1}{R}w_H)$ . We consider the distance through edges in  $H(\frac{1}{R}w_H)$  and through vertices in  $H(\frac{1}{R}w_H)$  (which are actually parts in  $G$ ) separately. For the edges, the distance is at most  $O(\frac{1}{\beta} \log n)R$  in  $H$ , and each of these edges has weight at least that in  $G$ , giving  $O(\frac{1}{\beta} \log n)R$  total distance. For the vertices, traversing through  $T$  inside the  $O(\frac{1}{\beta} \log n)$  parts takes  $O(R)$  distance each, by the input guarantee, and  $O(\frac{1}{\beta} \log n)R$  distance overall. Combining the two arguments proves (4)  $d_{T'}(x, y) \leq (\frac{c_1}{\beta} \log n)R$ . Note that (4) immediately implies that (1) the weighted diameter of each part  $C'_i$  using edge weights in  $G$  is at most  $(\frac{c_1}{\beta} \log n)R$ .

Finally, we prove that  $\mathbb{E}[d_{G(w')}(x, y)] \leq (c_2 \log n)d_{G(w)}(x, y)$ . If  $x, y \in V$  are in the same  $C'_i$ , then their distance in  $G(w')$  is zero and the claim follows. Otherwise, consider the shortest path in  $H$ , which is also the shortest path in  $H(\frac{1}{R}w_H)$ . By Lemma 16, every edge  $e$  on this path has probability at most  $1 - e^{-O(w(e)\beta)} = O(w(e)\beta)$  of being cut between two components, so the expected length is at most  $O(w(e)\beta) \cdot \frac{c_1}{\beta} \log n = O(w(e) \log n)$ . By linearity of expectation, the expected multiplicative increase of the path in  $H(\frac{1}{R}w_H)$ , and also in  $G(w')$ , is  $O(\log n)$ .  $\blacktriangleleft$

## 2.2 Algorithm Main Loop

In this section, we apply `LDDSubroutine` recursively with geometrically increasing values of  $R$ . We show that the resulting forest approximates distances in expectation.

### Algorithm $T = \text{ExpectedSPForest}(G, \beta, R_0)$

Input:

- $G = (V, E)$ , the network graph with edge weights in  $[1, \text{poly}(n)]$ .
- $\beta = (\log n)^{-\Omega(1)}$ , freely chosen.
- $R_0 \in [\frac{c_2\beta}{c_1}, 1]$

Algorithm:

1. Initially, set  $R^{(0)} := R_0$ ,  $T^{(0)} := \emptyset$ , and  $w^{(0)}$  to have the same edge weights as  $G$ .
2. For  $t = 1, 2, \dots$ , while  $R < n^c$  for large enough  $c$ :
  - a.  $(w^{(t)}, T^{(t)}) := \text{LDDSubroutine}(w^{(t-1)}, T^{(t-1)}, \beta, R^{(t-1)})$ .
  - b. Set  $R^{(t)} := (\frac{c_1}{\beta} \log n)R^{(t-1)}$ .
3. Output the forest obtained on the last iteration.

Note that  $T$  is not guaranteed to be a tree at the end of the algorithm, so distances within  $T$  can be infinite. However, a simple induction with linearity of expectation shows that the expected increase in length behaves in a controlled way:

**► Lemma 20.** *Let  $G$  be a network graph with edge weights in  $[1, \text{poly}(n)]$ , and let  $\beta := (\log n)^{-\Omega(1)}$ . On the  $t^{\text{th}}$  iteration of `ExpectedSPForest`, for any two vertices  $x, y \in V$ ,  $\mathbb{E}[d_{G(w^{(t)})}(x, y)] \leq (c_2 \log n)^t d_G(x, y)$ .*



We now show that we get approximate shortest paths with constant probability.

► **Lemma 21.** *Let  $G$  be a network graph with edge weights in  $[1, \text{poly}(n)]$ , and let  $\beta := (\log n)^{-\Omega(1)}$ . The algorithm *ExpectedSPForest* runs in  $\tilde{O}(\frac{1}{\beta}Q_G)$  rounds. Consider the output forest  $T$ , and fix any two vertices  $x, y \in V$ . Then,  $d_T(x, y) \geq d_G(x, y)$  always<sup>6</sup>, and with constant probability,  $d_T(x, y) \leq O(\frac{1}{\beta} \log n \cdot d_G(x, y)^{O(\log \log n)/\log(1/\beta)}) \cdot d_G(x, y)$ .*

**Proof.** For the running time, there are  $O(\frac{\log n}{\log(1/\beta)})$  iterations of the LDD, each of which takes  $\tilde{O}(Q_G)$  time.

For simpler notation, define  $L := d_G(x, y)$ . Since every edge added to  $T$  has weight at least the weight of that same edge in  $G$ , we clearly have  $d_T(x, y) \geq L$ . We now prove the other bound on  $d_T(x, y)$ .

Consider any iteration  $t$  such that  $R^{(t)} \geq 2(c_2 \log n)^t L$ . (We later argue that such an iteration  $t$  must exist.) By Lemma 20 and Markov's inequality,  $d_{\tilde{G}^{(t)}}(x, y) < R^{(t)}$  with probability at least  $\frac{1}{2}$ . If this occurs, then  $x$  and  $y$  cannot belong to different parts at iteration  $t$ , since the distance between parts is at least  $R^{(t)}$ . By the subroutine guarantee,  $d_{T^{(t)}}(x, y) = O(\frac{1}{\beta} \log n) R^{(t-1)} = O(R^{(t)})$ , and since the edges of  $T^{(t)}$  are preserved for the rest of the algorithm,  $d_T(x, y) = O(R^{(t)})$  as well. Therefore, for this value of  $t$ , the approximation factor is  $2(c_2 \log n)^t$  with probability at least  $\frac{1}{2}$ .

It remains to find the smallest satisfying  $t$ . The condition on  $t$  is equivalent to  $(\frac{c_1}{\beta} \log n)^t R_0 \geq 2(c_2 \log n)^t L$ , or  $t \geq \lceil \frac{\log(2L) - \log(R_0)}{\log(c_1/(c_2\beta))} \rceil$ . For  $t$  achieving equality, we get

$$R^{(t)} = O\left(\frac{c_1}{\beta} \log n\right)^t R_0 \leq \left(\frac{c_1}{\beta} \log n\right)^{\frac{\log(2L) - \log(R_0)}{\log(c_1/(c_2\beta))} + 1} = \frac{c_1}{\beta} \log n \cdot \left(\frac{c_1}{\beta} \log n\right)^{\frac{\log(2L) - \log(R_0)}{\log(c_1/(c_2\beta))}}.$$

First, consider the case when  $L \leq c_1/(c_2\beta)$ . Since  $R_0 \geq (c_2\beta)/c_1$ , we have  $\log(2L) - \log(R_0) \leq \log 2$ , so

$$R^{(t)} \leq \frac{c_1}{\beta} \log n \cdot \left(\frac{c_1}{\beta} \log n\right)^{\frac{\log 2}{\log(c_1/(c_2\beta))}} \leq \frac{c_1}{\beta} \log n \cdot O(1) \leq O\left(\frac{c_1}{\beta} \log n\right) \cdot L,$$

where the second-to-last inequality uses that  $\beta = (\log n)^{\Omega(1)}$ .

Now consider the case when  $L \geq c_1/(c_2\beta)$ . Since  $R_0 \geq (c_2\beta)/c_1$ , we have  $\log(2L) - \log(R_0) \leq \log(2L^2)$ , so

$$\begin{aligned} R^{(t)} &\leq \left(\frac{c_1}{\beta} \log n\right)^{\frac{\log(2L^2)}{\log(c_1/(c_2\beta))} + 1} = \frac{c_1}{\beta} \log n \cdot \left(\frac{c_1}{\beta} \log n\right)^{\frac{\log(2L^2)}{\log(c_1/(c_2\beta))}} \\ &= \frac{c_1}{\beta} \log n \cdot (2L^2)^{\frac{\log(c_1/\beta \cdot \log n)}{\log(c_1/(c_2\beta))}} = O\left(\frac{1}{\beta} \log n \cdot (2L^2)^{\frac{O(\log \log n)}{\log(1/\beta)}}\right), \end{aligned}$$

as desired.

Lastly, we show that such an iteration  $t$  must exist. In particular, we show that the value of  $t$  chosen above satisfies  $R^{(t)} \leq n^c$  for some large enough constant  $c$  in the algorithm. Since  $L = \text{poly}(n)$  and  $R = 1/\text{poly}(n)$ , we have

$$t = \left\lceil \frac{\log(2L) - \log(R_0)}{\log(c_1/(c_2\beta))} \right\rceil = O\left(\frac{\log n}{\log(1/\beta)}\right).$$

<sup>6</sup> In particular,  $d_T(x, y) = \infty$  if  $x$  and  $y$  are not in the same connected component in  $T$

### 33:10 Faster Distributed Shortest Path Approximations via Shortcuts

Therefore,

$$\begin{aligned} R^{(t)} &= \left(\frac{c_1}{\beta} \log n\right)^t R_0 = \left(\frac{\log n}{\beta}\right)^{O\left(\frac{\log n}{\log(1/\beta)}\right)} = \left(\frac{1}{\beta}\right)^{O\left(\frac{\log n}{\log(1/\beta)}\right)} \cdot (\log n)^{O\left(\frac{\log n}{\log(1/\beta)}\right)} \\ &= n^{O(1)} \cdot n^{O(1)}, \end{aligned}$$

where the last equality uses the fact that  $\beta = (\log n)^{-\Omega(1)} \implies \log(1/\beta) = \Omega(\log \log n)$ . Therefore,  $R^{(t)} \leq n^c$  for large enough  $c$ .  $\blacktriangleleft$

From the shortest path forest, we can also derive the distances to each vertex  $v$  from a specified source  $s$ . Below is the algorithm, which runs in  $\tilde{O}(\frac{1}{\beta}Q_G)$  rounds.

#### Algorithm ExpectedSPDistance( $G, \beta, s$ )

1. Run `ExpectedSPForest( $G, \beta$ )` to obtain forest  $T$ . Set  $\tilde{T}$  to be the connected component of  $T$  that contains the source  $s$ .
2. For all vertices  $v \notin \tilde{T}$ , set  $d(s, v) := \infty$ .
3. Run `AggregatePathToRoot` (see Appendix B of the full version) with  $x_v = 1$  for all  $v \in \tilde{T}$  to determine the depth of each vertex in the tree  $\tilde{T}$  rooted at  $s$ .
4. Every vertex  $v \in \tilde{T} \setminus \{s\}$  computes its parent in the rooted tree, which it can determine by finding the one neighbor with smaller depth.
5. For each  $v \in \tilde{T} \setminus \{s\}$ , set  $x_v$  to be the weight of the edge to its parent, and set  $x_s := 0$ . Run `AggregatePathToRoot` on these values to determine  $d(s, v)$  for  $v \in T$ .

## 3 Solving SSSP and Related Problems

### 3.1 SSSP Trees

In this section, we describe an algorithm that outputs an approximate single source shortest path tree with source  $s$ . At a high level, to boost the probability that distances are well-approximated, we construct many randomized trees and take a collective “best” tree.

#### Algorithm SSSPTree( $G, \beta, s$ )

1. Repeat `ExpectedSPDistance( $G, \beta, s$ )`  $\Theta(\log n)$  times to obtain distances  $d_{T_i}(v) := d_{T_i}(s, v)$ .
2. For each vertex  $v$ , set  $d_{\min}(v) := \min_i d_{T_i}(v)$ .
3. For each vertex  $v$  except the source, connect an edge to some neighbor  $u$  that satisfies  $d_{\min}(u) + w_{(u,v)} \leq d_{\min}(v)$ . Return the tree  $T^*$  of all such edges.

► **Lemma 22.** *Let  $G$  be a network graph with edge weights in  $[1, \text{poly}(n)]$ , and let  $\beta := (\log n)^{-\Omega(1)}$ . W.h.p., `SSSPTree` runs for  $\tilde{O}(\frac{1}{\beta}Q_G)$  rounds and outputs a shortest path tree that  $O(\frac{1}{\beta}d_G(v) \frac{O(\log \log n)}{\log(1/\beta)}) \log n$ -approximates distances from the source to each  $v$ .*

**Proof.** Observe that in step 3 of `SSSPTree`, such a neighbor always exists, since in the tree  $T_i$  that achieves distance  $d_{\min}(v)$  to  $v$ , the parent  $u$  of  $v$  in  $T_i$  satisfies  $d_{\min}(u) + w_{(u,v)} = d_{\min}(v)$ . To show that  $d_{T^*}(v) \leq d(v)$  for each  $v$ , consider the path  $s = v_0, v_1, v_2, \dots, v_\ell = v$  in  $T^*$ . We have  $w(v_i, v_{i-1}) \leq d_{\min}(v_i) - d_{\min}(v_{i-1})$  for each  $i$ , and summing up the inequalities gives the result.

From Lemma 21, each vertex  $v$  achieves the desired approximation with constant probability. By taking the minimum  $d_{T_i}(v)$  over  $\Theta(\log n)$  trees, this approximation is satisfied w.h.p. for every  $v$ , giving  $d_{T^*}(v) \leq d_{\min}(v) = O(\frac{1}{\beta}d_G(v))^{1+\frac{O(\log \log n)}{\log(1/\beta)}} \log n \cdot d_G(v)$ . ◀

By repeating `SSSPTree` multiple times with differing  $R_0$ , we can shave off the  $\frac{1}{\beta} \log n$  in the approximation as follows, giving our main result for SSSP.

► **Theorem 7.** *Let  $G$  be a network graph with edge weights in  $[1, \text{poly}(n)]$ , with a specified source vertex, and let  $\beta := (\log n)^{-\Omega(1)}$ . There is a distributed algorithm that, w.h.p., runs for  $\tilde{O}(\frac{1}{\beta}Q_G)$  rounds and outputs a spanning tree that approximates distances to the source to factor  $O(L^{O(\log \log n)/\log(1/\beta)})$ .<sup>7</sup> By output, we mean that at the end of the algorithm, every vertex knows its set of incident edges in the spanning tree.*

**Proof.** We first handle the pairs  $u, v \in V$  with  $d(u, v) \geq c_1/(c_2\beta)$ .

Run `SSSPTree`  $\lceil \log(c_1/(c_2\beta)) \rceil = O(\log n)$  many times, setting  $R_0 := 2^{-t}$  on the  $t^{\text{th}}$  iteration. The total number of rounds is  $\tilde{O}(\frac{1}{\beta}Q_G)$ . Consider the case  $L \geq c_1/(c_2\beta)$  in the proof of 21. Observe that the factor  $\frac{1}{\beta}$  comes from the  $+1$  in the ceiling computation in the expression  $\lceil \frac{\log(2L) - \log(R_0)}{\log(c_1/(c_2\beta))} \rceil$ . However, with the differing values of  $R_0$ , there exists one such  $R_0$  such that taking the ceiling increases the value by at most  $\frac{1}{\log(c_1/(c_2\beta))}$ . This factor gets absorbed in the exponent  $\frac{O(\log \log n)}{\log(1/\beta)}$ . Therefore, for each  $v$ , there exists a tree with this approximation factor, and running steps 2 and 3 from `SSSPTree` on these trees gives the result.

Now we handle the pairs  $u, v \in V$  with  $d(u, v) \leq c_1/(c_2\beta)$ . Intuitively, this should not be a problem: if we run an LDD with  $\beta \in [1/L, 2/L]$ , then by Lemma 16, with constant probability,  $u$  and  $v$  are in a common component of diameter  $O(L \log n)$ , stretching the distance by a factor  $O(\log n)$ .

Let us define  $w_G$  to be the weights of the input graph  $G$ . Then the algorithm runs `LDDSubroutine`( $w_G, \emptyset, \beta', 1$ ) times for each  $\beta' \in [1, c_1/(c_2\beta)]$  satisfying  $\beta' = 2^{-i}$  for some positive integer  $i$ , and repeats this loop  $O(\log n)$  times. In total, this takes  $\tilde{O}(\frac{1}{\beta}Q_G)$ . W.h.p., for each pair  $u, v \in V$  with  $d(u, v) \leq c_1/(c_2\beta)$ , there is a forest  $T$  returned by one of the `LDDSubroutines` for which  $d_T(u, v) \leq O(\log n)d_G(u, v)$ . Finally, running steps 2 and 3 from `SSSPTree` on these forests, along with the tree obtained from the case  $L \geq c_1/(c_2\beta)$ , gives the desired SSSP tree. ◀

## 3.2 Distance Labeling Schemes

We restate our main result on approximate distance labeling schemes.

► **Theorem 11.** *Let  $G$  be a network graph with edge weights in  $[1, \text{poly}(n)]$ . There exists a  $(\text{polylog}(n), n^{O(\log \log n)/\log(1/\beta)})$  approximate distance labeling scheme that runs in  $\tilde{O}(\frac{1}{\beta}Q_G)$  rounds.*

**Proof.** For each  $t$  from 1 to  $\lceil \log(c_1/(c_2\beta)) \rceil$ , run `ExpectedSPForest`  $\Theta(\log n)$  times with  $R_0 := 2^{-t}$ . By analysis from Lemma 21 and Theorem 7, w.h.p., for every  $x, y \in V$ , there is an iteration of `ExpectedSPForest` with  $R = O(d_G(v))^{1+\frac{O(\log \log n)}{\log(1/\beta)}}$  that outputs a cluster containing both  $x$  and  $y$ . The total number of rounds is  $\tilde{O}(\frac{1}{\beta}Q_G)$ .

<sup>7</sup> Recall that  $L = \max_{u,v} d_G(u, v)$ .

In each of the  $O(\log^2 n)$  iterations of **ExpectedSPForest**, consider all of the clusters formed throughout the algorithm, and give each one a unique ID. For every iteration with parameter  $R$  and a cluster formed in that iteration, assign to every vertex within the cluster the label  $(\text{ID}, R)$ . Each vertex is assigned to  $O(\frac{\log n}{\log(1/\beta)})$  clusters per **ExpectedSPForest**, so the label size is  $\text{polylog}(n)$ .

To compute distances given two vertices  $x, y \in V$ , simply output the minimum possible  $R$  over all clusters that contain both  $x$  and  $y$ , which is easily computed with the labels of  $x$  and  $y$ . By the analysis above, the minimum possible  $R$  gives the desired approximation factor  $O(d_G(v)^{O(\log \log n)/\log(1/\beta)}) = O(n^{O(\log \log n)/\log(1/\beta)})$ . ◀

### 3.3 Transshipment Problem

Let  $G$  be a transshipment network with demand  $d_v$  at each node  $v$ . The following algorithm computes an approximate transshipment flow in expectation.

**Algorithm ExpectedTS**

1. Run **ExpectedSPForest** and root the tree  $T$  arbitrarily.
2. Using **AggregateSubtree** (see Appendix B) compute  $F(v) := \sum_{u \in S_v} d_u$  for all  $v$ , where  $S_v$  is the subtree rooted at  $v$ .
3. For each edge  $(v, p) \in T$  with  $p$  the parent of  $v$  in the rooted tree, direct  $F(v)$  flow from  $v$  to  $p$ . (If  $F(v)$  is negative, then direct the flow the other way.)

► **Lemma 23.** *Let  $G$  be a network graph with edge weights in  $[1, \text{poly}(n)]$  and demands that sum to zero, and let  $\beta := (\log n)^{-\Omega(1)}$ . The expected total cost of **ExpectedTS** is within  $\tilde{O}(\frac{1}{\beta} n^{O(\log \log n)/\log(1/\beta)})$  of optimum.*

**Proof.** Decompose the optimal solution into a set of (shortest) paths. For a path from  $s$  to  $t$ , we have  $\mathbb{E}[d_T(s, t)] = \tilde{O}(\frac{1}{\beta} n^{O(\log \log n)/\log(1/\beta)}) \cdot d_G(s, t)$  by Lemma 21, and by linearity of expectation, the cost  $C$  of routing each of these paths through  $T$  gives an expected  $\tilde{O}(\frac{1}{\beta} n^{O(\log \log n)/\log(1/\beta)})$  approximation. It remains to show that the total cost of **ExpectedTS** is at most  $C$ . If **ExpectedTS** places  $F(e)$  flow along an edge  $e$ , then the total demand difference between the two halves of the tree split at  $e$  is  $|2F|$ . Therefore, any sequence of paths along  $T$  that satisfies all demands must route at least  $|F|$  flow along edge  $e$ . It follows that  $C$  must be at least the cost of **ExpectedTS**. ◀

By running **ExpectedTS** repeatedly and taking the overall best flow, we obtain our main result for transshipment.

► **Theorem 13.** *Let  $G$  be a network graph with edge weights in  $[1, \text{poly}(n)]$  and demands that sum to zero, and let  $\beta := (\log n)^{-\Omega(1)}$ . There is an algorithm that, w.h.p., runs for  $\tilde{O}(\frac{1}{\beta} Q_G)$  rounds and computes a  $\tilde{O}(\frac{1}{\beta} n^{O(\log \log n)/\log(1/\beta)})$ -approximate flow.*

**Proof.** Run **ExpectedTS**  $\Theta(\log n)$  many times and output the minimum total cost. By Markov's inequality and Lemma 23, w.h.p., some iteration achieves within twice the expected approximation of  $\tilde{O}(\frac{1}{\beta} n^{O(\log \log n)/\log(1/\beta)})$ . ◀

## 4 Conclusion and Future Work

Using the shortcuts framework from [8, 9], we give the first nontrivial approximation algorithms for shortest path problems which run in  $o(\sqrt{n} + D)$  time on non-pathological

network topologies. Our algorithms feature a tuneable parameter  $\beta$  that represents the balance between approximation ratio and running time. For certain values of  $\beta$ , we obtain polylogarithmic-approximate solutions in  $\tilde{O}(n^\epsilon \cdot Q_G)$  rounds for the shortest path and distance labeling problems. While sublogarithmic approximation ratios are known to be impossible (even existentially) for labeling schemes with polylogarithmic labels we believe that our approximation guarantees can likely be improved for nice family of graphs, and, in the case of the SSSP-tree and transshipment problems, even generally.

In particular, for the quite general set of minor closed families of graphs one might be able to use more sophisticated low-diameter decompositions, such as [1], which would directly lead to  $O(1)$ -approximation guarantees for such networks in our framework. However, [1] is written for the sequential setting and making the algorithms in [1] distributed and compatible with the shortcut framework is a nontrivial extension, which we plan to explore for the journal version of this work.

More importantly, it seems possible that our non-trivial approximation ratios for the SSSP-tree and transshipment problem can be improved all the way to  $(1 + \epsilon)$ -approximations using tools from continuous optimization, such as, gradient descent or the multiplicative weights method. As one example, the recent and brilliant work of Becker et al. [5] shows how to obtain a  $(1 + \epsilon)$ -approximation for the SSSP-tree problem and the transshipment problem by computing  $\tilde{O}(\alpha^2)$  many  $\alpha$ -approximations to the transshipment problem. This work also demonstrates that the required updates to weight and demand vectors can be performed in various non-centralized models, including CONGEST. If this method could be applied to our transshipment algorithm, we could choose  $\beta = 2^{-O(\sqrt{\log n \log \log n})}$  to get a  $2^{O(\sqrt{\log n \log \log n})}$ -approximate solution to the transshipment problem in  $Q_G \cdot 2^{O(\sqrt{\log n \log \log n})}$  rounds, which could then be transformed into a  $(1 + \epsilon)$  approximation with the exact same running time (up to the constant hidden by the  $O$ -notation). This extension is highly nontrivial as well and left for future work.

---

## References

- 1 Ittai Abraham, Cyril Gavoille, Anupam Gupta, Ofer Neiman, and Kunal Talwar. Cops, robbers, and threatening skeletons: Padded decomposition for minor-free graphs. In *Proceedings of the 46th Annual ACM Symposium on Theory of Computing*, pages 79–88. ACM, 2014.
- 2 Noga Alon, Richard M Karp, David Peleg, and Douglas West. A graph-theoretic game and its application to the k-server problem. *SIAM Journal on Computing*, 24(1):78–100, 1995.
- 3 Baruch Awerbuch, Bonnie Berger, Lenore Cowen, and David Peleg. Low-diameter graph decomposition is in nc. In *Scandinavian Workshop on Algorithm Theory*, pages 83–93. Springer, 1992.
- 4 Yair Bartal. Probabilistic approximation of metric spaces and its algorithmic applications. In *Foundations of Computer Science, 1996. Proceedings., 37th Annual Symposium on*, pages 184–193. IEEE, 1996.
- 5 Ruben Becker, Andreas Karrenbauer, Sebastian Krinninger, and Christoph Lenzen. Near-optimal approximate shortest paths and transshipment in distributed and streaming models. In *International Symposium on Distributed Computing*, 2017.
- 6 Guy E Blelloch, Anupam Gupta, Ioannis Koutis, Gary L Miller, Richard Peng, and Kanat Tangwongsan. Nearly-linear work parallel sdd solvers, low-diameter decomposition, and low-stretch subgraphs. *Theory of Computing Systems*, 55(3):521–554, 2014.

- 7 Michael Elkin and Ofer Neiman. Distributed strong diameter network decomposition. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, pages 211–216. ACM, 2016.
- 8 Mohsen Ghaffari and Bernhard Haeupler. Distributed algorithms for planar networks ii: Low-congestion shortcuts, mst, and min-cut. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 202–219. Society for Industrial and Applied Mathematics, 2016.
- 9 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.
- 10 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.
- 11 Bernhard Haeupler, Goran Zuzic, and Jason Li. Low-congestion shortcuts for any minor closed family. In *personal communications*, 2017.
- 12 Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. An almost-tight distributed algorithm for computing single-source shortest paths. In *Proceedings of the ACM Symposium on Theory of Computing*, 2016.
- 13 Shay Kutten and David Peleg. Fast distributed construction of  $k$ -dominating sets and applications. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 238–251. ACM, 1995.
- 14 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. ACM, 2015.
- 15 Nathan Linial and Michael E Saks. Decomposing graphs into regions of small diameter. In *SODA*, volume 91, pages 320–330, 1991.
- 16 Gary L Miller, Richard Peng, and Shen Chen Xu. Parallel graph decompositions using random shifts. In *Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures*, pages 196–203. ACM, 2013.
- 17 Danupon Nanongkai. Distributed approximation algorithms for weighted shortest paths. In *Proceedings of the ACM Symposium on Theory of Computing*, pages 565–573, 2014.
- 18 Danupon Nanongkai and Hsin-Hao Su. Almost-tight distributed minimum cut algorithms. In *International Symposium on Distributed Computing*, pages 439–453. Springer, 2014.
- 19 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.

# A Lower Bound for Adaptively-Secure Collective Coin-Flipping Protocols

**Yael Tauman Kalai**

Microsoft Research, 1 Memorial Dr, Cambridge, MA 02142, USA  
yael@microsoft.com

**Ilan Komargodski**<sup>1</sup>

Cornell Tech, 2 W Loop Rd, New York, NY 10044, USA  
komargodski@cornell.edu

**Ran Raz**<sup>2</sup>

Department of Computer Science, Princeton University, Princeton, NJ 08544, USA  
ran.raz.mail@gmail.com

---

## Abstract

In 1985, Ben-Or and Linial (Advances in Computing Research '89) introduced the collective coin-flipping problem, where  $n$  parties communicate via a single broadcast channel and wish to generate a common random bit in the presence of *adaptive* Byzantine corruptions. In this model, the adversary can decide to corrupt a party in the course of the protocol as a function of the messages seen so far. They showed that the majority protocol, in which each player sends a random bit and the output is the majority value, tolerates  $O(\sqrt{n})$  adaptive corruptions. They conjectured that this is optimal for such adversaries.

We prove that the majority protocol is optimal (up to a poly-logarithmic factor) among all protocols in which each party sends a single, *possibly long*, message.

Previously, such a lower bound was known for protocols in which parties are allowed to send only a *single* bit (Lichtenstein, Linial, and Saks, Combinatorica '89), or for symmetric protocols (Goldwasser, Kalai, and Park, ICALP '15).

**2012 ACM Subject Classification** Theory of computation → Complexity theory and logic

**Keywords and phrases** Coin flipping, adaptive corruptions, byzantine faults, lower bound

**Digital Object Identifier** 10.4230/LIPIcs.DISC.2018.34

## 1 Introduction

In the *collective coin-flipping* problem, introduced by Ben-Or and Linial [7], a set of  $n$  computationally unbounded parties, each equipped with a private source of randomness, are required to generate a common random bit. The communication model is the “full information” model [7], where all parties communicate via a single broadcast channel. The goal of the parties is to agree on a common random bit even in the case that some  $t = t(n)$  of the parties are faulty and controlled by an adversary whose goal is to bias the output of the protocol in some direction. We say that a protocol  $\Pi$  is *resilient* (or *secure*) to  $t$  corruptions if for any adversary  $\mathcal{A}$  that makes at most  $t$  corruptions it holds that

$$\min \left\{ \Pr [\text{Output of } \mathcal{A}(\Pi) = 0], \Pr [\text{Output of } \mathcal{A}(\Pi) = 1] \right\} \geq \Omega(1),$$

---

<sup>1</sup> Supported in part by a Packard Foundation Fellowship and by an AFOSR grant FA9550-15-1-0262.

<sup>2</sup> Research supported by the Simons Collaboration on Algorithms and Geometry and by the National Science Foundation grants No. CCF-1714779 and CCF-1412958.





where “Output of  $\mathcal{A}(\Pi)$ ” is a random variable that corresponds to the output of the protocol  $\Pi$  when executed in the presence of the adversary  $\mathcal{A}$ .

The adversary is Byzantine, namely, once it corrupts a party, it completely controls it and can send arbitrary messages on its behalf. Usually, two types of Byzantine adversaries are considered, static or adaptive ones. A *static* adversary is an adversary that chooses which parties to corrupt ahead of time, before the protocol begins. An *adaptive* adversary, on the other hand, is allowed to choose which parties to corrupt adaptively in the course of the protocol as a function of the messages seen so far. In the case of static adversaries, collective coin-flipping is well studied and almost matching upper and lower bounds are known; see Section 1.1. However, the case of adaptive adversaries is much less understood. In this work, we focus on the setting of adaptive adversaries.

In the seminal work of Ben-Or and Linial [7], they showed that the majority protocol (in which each party sends a uniformly random bit and the output of the protocol is the majority of the bits sent) is resilient to  $O(\sqrt{n})$  adaptive corruptions. Moreover, with  $\tilde{\Omega}(\sqrt{n})$  corruptions,<sup>3</sup> one can break the security of this protocol. They conjectured that the majority protocol is *optimal*: any collective coin-flipping protocol is resilient to at most  $O(\sqrt{n})$  adaptive corruptions, even if parties send multiple messages, each of which may be long.

The first step towards this conjecture was made by Lichtenstein, Linial, and Saks [19]. They proved that there is no *single-bit* and *single-turn* protocol which is resilient to more than  $\tilde{\Omega}(\sqrt{n})$  adaptive corruptions. A single-bit protocol is one in which parties’ messages consist of a single bit (perhaps over multiple rounds), and a single-turn protocol is such that each party speaks at most once (perhaps with a long message). More recently, Goldwasser, Kalai, and Park [15] proved another special case of the conjecture: Any *symmetric*<sup>4</sup> single-turn protocol cannot be resilient to more than  $\tilde{\Omega}(\sqrt{n})$  adaptive corruptions.

Despite significant efforts, more than three decades after posting the conjecture, fully resolving it remains an intriguing open problem.

**Our results.** We prove that any  $n$ -party collective coin-flipping protocol in which each party sends a single, possibly long, message cannot be secure against more than  $t = \tilde{\Omega}(\sqrt{n})$  adaptive corruptions.

► **Theorem 1.** *Any  $n$ -party single-turn collective coin-flipping protocol is insecure against more than  $t = \tilde{\Omega}(\sqrt{n})$  adaptive corruptions.*

As a warm-up, in Section 3, we recover the result of Lichtenstein et al. [19] for single-bit single-turn protocols. Whereas the original proof of [19] is based on combinatorial arguments in extremal set theory, our proof is elementary and uses basic tools from probability theory. A different yet related variant to our simplification was previously given by Cleve and Impagliazzo [11]; see Section 1.1 below.

## 1.1 Related Work

The *full information model* was introduced by Ben Or and Linial [7] to study the collective coin-flipping problem. Since then, this problem was central in the study of distributed protocols.

<sup>3</sup> Throughout this work, the notation  $\tilde{\Omega}$  and  $\tilde{O}$  suppresses poly-logarithmic factors.

<sup>4</sup> A symmetric protocol  $\Pi$  is one that is oblivious to the order of its inputs: namely, for any permutation  $\pi: [n] \rightarrow [n]$  of the parties, it holds that  $\Pi(r_1, \dots, r_n) = \Pi(r_{\pi(1)}, \dots, r_{\pi(n)})$ .

**Static adversaries.** The case of static corruptions has been extensively studied since the introduction of the collective coin-flipping problem. The original work of Ben-Or and Linial [7] showed that a polynomial number (i.e.,  $O(n^{63})$ ) of corrupted parties can be tolerated. Later, Ajtai and Linial [1] showed a different protocol that withstands  $O(n/\log^2 n)$  corruptions. For single-round single-bit protocols, in which the global coin is obtained by each party contributing one bit for an  $n$ -input predefined Boolean function, Kahn, Kalai and Linial [17] showed that no protocol is resilient to more than  $\Omega(n/\log n)$  corruptions. Saks [22] introduced a multi-round protocol called the “Baton Passing” game<sup>5</sup> and showed that it is resilient to  $O(n/\log n)$  corruptions. The protocol of Saks was modified by Alon and Naor [3] such that it tolerates a constant fraction of corrupted parties. The optimal resilience of  $t = (1/2 - \delta)n$  for any  $\delta > 0$  was obtained by Boppana and Narayanan [8] shortly afterwards. Since then the focus has been on improving the explicitness of the protocol, the round complexity, and the bias of the output bit. Two of the most notable results are that of Feige [14] and of Russell, Saks, and Zuckerman [21]. Feige gave an explicit  $(\log^* n + O(1/\delta))$ -round protocol that tolerates  $(1/2 - \delta)n$  corruptions for any constant  $\delta > 0$ . Russell, Saks, and Zuckerman proved that any protocol that is secure against  $\Omega(n)$  corruptions must either have at least  $(1/2 - o(1)) \cdot \log^* n$  rounds, or communicate multiple bits per round.

Interestingly, many protocols for collective coin-flipping that consist of more than one round of communication per party, achieve a seemingly stronger goal. In these protocols, first an honest leader is elected and then it outputs a bit that is taken as the protocol outcome. This approach, while being useful for the static case, is unsuitable for adaptive adversaries, since the adversary may always wait for the leader to be elected and then corrupt it.

**Adaptive adversaries.** The literature on collective coin-flipping with adaptive adversaries is much more scarce. The best known protocol is the majority one suggested by Ben-Or and Linial [7]. Lichtenstein, Linial, and Saks [19] proved that there is no protocol in which each party is allowed to send *one* bit (in total) which is resilient to more than  $\Omega(\sqrt{n})$  corruptions. The same lower bound was shown by Goldwasser, Kalai and Park [15] for any single-turn *symmetric* protocol (where each message can be long).

Dodis [12] proved that through “black-box” reductions from non-adaptive collective coin-flipping protocols, it is impossible to tolerate significantly more corruptions than the majority protocol. His definition of “black-box” is rather restricted: It only considers sequential composition of non-adaptive coin-flipping protocols, followed by a (non-interactive, predefined) function operating on the coin-flips thus obtained.

Kalai and Komargodski [18] showed that for any collective coin-flipping protocol in which messages are long there is a collective coin-flipping protocol with the same communication pattern, the same output distribution, the same security guarantees, and where parties send messages of length  $\ell = \text{polylog}(n, d)$ , where  $d$  is the number of rounds in the original protocol. In particular, their transformation guarantees that the resulting protocol is resilient against  $t$  adaptive (resp. static) corruptions as long as the original one is resilient against  $t$  adaptive (resp. static) corruptions. The transformation is non-uniform, that is, they only show that the required protocol exists.

**More types of adversaries.** En route to resolving the conjecture of Ben-Or and Linial, stronger types of adversaries were considered.

<sup>5</sup> In this game, each party receiving the baton, passes it to a random party that did not have it yet. The last party having the baton is the leader, and the leader chooses the random bit to be outputted.

Cleve and Impagliazzo [11] studied *re-sampling* adaptive adversaries that can decide whether to intervene in the next message or not *after seeing* it. More precisely, at the  $i$ 'th round, the adversary, after seeing all the messages exchanged in the first  $i - 1$  rounds *and* the message to be sent in the current round, can ask to rewind the process back to the beginning of the  $i$ 'th round and have the  $i$ 'th message be re-sampled. They showed that for any protocol whose expected output is  $1/2$  in an honest execution and in which each party sends just one (possibly long) message, there is an adversary that corrupts a *single* party and biases the expectation of the output of the protocol away from  $1/2$  by  $\Omega(1/\sqrt{n})$  in some direction.

More recently, Goldwasser, Kalai, and Park [15] studied an even stronger variant called *strong* adaptive adversaries in which the adversary sees all messages sent by honest parties in any given round and, based on the message content, decide whether to corrupt a party or not (and alter its message for that same round). Here, a corruption allows the adversary to send any message on behalf of the party (and not only re-sample it, as in [11]). They proved that any one-round protocol (i.e., all parties talk simultaneously once), in which messages can be arbitrarily long, can tolerate at most  $\tilde{O}(\sqrt{n})$  such (strong) adaptive corruptions. They got a similar lower bound in the standard adaptive corruptions model for symmetric protocols.

**Fair Coin-Flipping.** There is a rich literature on coin-flipping protocols in settings with dishonest majority (and static corruptions), starting from the seminal work of Cleve [10]. In such protocols, the output of the protocol is a random bit, and the requirement is that even in the presence of an adversary, the output cannot be skewed towards 0 or towards 1 except with very small probability.<sup>6</sup>

Cleve [10] proved that for  $r$ -round coin-flipping protocol there exists a (static) adversary corrupting  $1/2$  of the parties and efficiently biases the output by  $\Theta(1/r)$ . This lower bound was shown to be tight in the two-party case by Moran, Naor, and Segev [20] and in the three-party case (up to a polylog factor) by Haitner and Tsfadia [16]. In the general  $n$ -party case, as long as  $n \leq \log \log r$ , an almost tight upper bound was given by Buchbinder et al. [9]. When there are less than  $(2/3)n$  corruptions, Beimel et al. [6] have constructed an  $n$ -party  $r$ -round coin-flipping protocol with bias  $2^{2^k}/r$ , tolerating up to  $t = (n + k)/2$  corrupt parties. Alon and Omri [2] constructed an  $n$ -party  $r$ -round coin-flipping protocol with bias  $\tilde{O}(2^{2^n}/r)$ , tolerating up to  $t$  corrupted parties, for constant  $n$  and  $t < 3n/4$ . Very recently, Beimel et al. [5] gave an improved lower bound in the multi-party case: For any  $n$ -party  $r$ -round coin-flipping protocol with  $n^k \geq r$  for  $k \in \mathbb{N}$ , there exists an adversary corrupting  $n - 1$  parties and biases the output of the honest party by  $1/(\sqrt{r} \log^k r)$ .

## 1.2 Proof Overview

Since we are in the full information model, we can assume (without loss of generality) that any collective coin-flipping protocol (in which the parties do not have private inputs except for a perfect source of randomness), can be transformed into a protocol in which the honest parties' messages consist only of uniformly random bits. A sketch of this folklore fact appears in [18, Section 4]. Thus, from now on, we assume that each party sends a uniformly random message chosen independently of the previous messages.

---

<sup>6</sup> We emphasize that in our work, we only require that the adversary cannot skew the output with probability  $1 - o(1)$ , whereas in fair protocols the adversary should not skew the output with probability greater than  $1/2 + o(1)$ .

Concretely, we consider protocols in which each party sends a single message of length  $\ell$ , possibly across  $n$  rounds. Such a protocol can be thought of as a complete  $2^\ell$ -ary tree whose leaves are labeled by 0 and 1, and whose internal nodes are labeled by numbers in  $[n]$ . If a node is labeled by  $i \in [n]$ , then we say that the node is owned by party  $i$ . (Without loss of generality, we can assume that the order in which the parties send messages is fixed in advance). The protocol starts at the root and at each time step we are at an internal node whose owner samples a random string in  $\{0, 1\}^\ell$  to determine where the protocol proceeds. The protocol ends once we reach a leaf and the output of the protocol is the bit  $b$  corresponding to the label of that leaf.

Let us start with the simpler case where  $\ell = 1$ . In this case, we present an attacker that biases the outcome of any protocol towards 0 with probability  $1 - \text{negl}(n)$ , while corrupting at most  $\tilde{O}(\sqrt{n})$  parties with probability  $1 - o(1)$ . (An analogous adversary can bias towards 1 with similar parameters.) The adversary at any point in time computes its possible gain in the expected output of the protocol by corrupting the next party (either to 0 or to 1). If the gain is larger than  $\epsilon = \frac{1}{\sqrt{n \cdot \log^2 n}}$ , then the adversary corrupts and sends the maliciously chosen bit (that biases the output towards 0). A standard application of Azuma's inequality shows that (with high probability) the influence of the parties that were not corrupted on the expected output of the protocol is negligible, as there are at most  $n$  of them and the contribution of each of them is at most  $\frac{1}{\sqrt{n \cdot \log^2 n}}$ . Intuitively, this means that only the corrupted parties influence the final output of the protocol and since the adversary controls these parties, the adversary succeeds in forcing the output to be 0 with high probability. Moreover, since the adversary gains at least  $\frac{1}{\sqrt{n \cdot \log^2 n}}$  in the expected value of the protocol, with the corruption of each party, and the total gain is at most 1, with high probability the number of corruptions is at most  $\tilde{O}(\sqrt{n})$ . This gives an alternative (elementary) proof for the result of [19]. This is formally proved in Section 3.

The proof for the case  $\ell > 1$  is more involved. We define two adversaries  $\mathcal{A}_0$  and  $\mathcal{A}_1$ , where  $\mathcal{A}_b$  tries to bias the outcome of the protocol towards  $b$ . Here, as opposed to the case  $\ell = 1$ , only one of the adversaries will be guaranteed to succeed. For  $\mathcal{A}_0$ , we associate with each node  $v$  in the protocol tree three values (we do the same for  $\mathcal{A}_1$ ):

1.  $\alpha_v$  : The expectation of the outcome of the protocol in the presence of the adversary  $\mathcal{A}_0$ , given that the protocol is at node  $v$ .
2.  $c_v^0$  : A bit that is 1 if and only if the adversary  $\mathcal{A}_0$  corrupts node  $v$ .
3.  $p_v^0$  : A ‘‘penalty’’ value that is proportional to the expected number of corruptions made by  $\mathcal{A}_0$  from node  $v$  onward.

We set these values inductively from the leaves of the protocol tree to the root. For a leaf labeled by  $b \in \{0, 1\}$ , we set  $\alpha_v = b$  and  $c_v^0 = p_v^0 = 0$ .

Going one level up to the parents of the leaves, for each such node we compute the expected  $\alpha$  value if we proceed to a random child, compared with the minimal possible  $\alpha$  value over all children (this corresponds to the maximal gain possible via corruption). If the possible gain by corruption is larger than  $\epsilon = \frac{1}{\sqrt{n \cdot \log^3 n}}$ , the adversary will corrupt  $v$ , so we set  $c_v^0 = 1$ , and we update the penalty value by setting it to be  $p_v^0 = \epsilon$ , to appropriately accommodate for this.

In the next levels, the situation is more complicated as we need to take into account the penalty values. For example, if there is a strategy for corrupting the next message that will increase our chance of outputting 0 by much, but has a high penalty (i.e., will require many corruptions in the future), this move is not always worthwhile for the attacker. So, instead of comparing only the expected outcome of the protocol, we take into account also the penalty.

For every node  $v$ , we define  $\alpha'_v = \alpha_v + p_v^0$ , and compare the expected gain versus the best possible gain *with respect to*  $\alpha'_v$  (rather than  $\alpha_v$ ). Namely, we compute the expected  $\alpha'$  value if we proceed to a random child, and compare it to the minimal possible  $\alpha'$  value of all children. If this gap is larger than  $\epsilon$ , the adversary corrupts  $v$ , and thus we set  $c_v^0 = 1$  and we set the penalty value  $p_v^0$  to be  $p_v^0 = p_u^0 + \epsilon$ , where  $u$  is the child that the adversary proceeds to.

The inductive process ends with a triple of values  $(\alpha_{\text{root}}, c_{\text{root}}^0, p_{\text{root}}^0)$ , corresponding to the root node and the adversary  $\mathcal{A}_0$ . The penalty value  $p_{\text{root}}^0$  is equal to  $\epsilon$  times the expected number of corruptions that the adversary  $\mathcal{A}_0$  makes. The probability that the protocol outputs 0 with adversary  $\mathcal{A}_0$  is  $1 - \alpha_{\text{root}}$ .

Similarly, we define the adversary  $\mathcal{A}_1$  and obtain the values  $(\beta_{\text{root}}, c_{\text{root}}^1, p_{\text{root}}^1)$ , where the penalty value  $p_{\text{root}}^1$  is equal to  $\epsilon$  times the expected number of corruptions that the adversary  $\mathcal{A}_1$  makes, and the probability that the protocol outputs 1 with adversary  $\mathcal{A}_1$  is  $\beta_{\text{root}}$ .

It is not possible to prove that both adversaries succeed with high probability (as there are protocols that can only be biased towards one of the two possible values, with the corruption of  $\tilde{O}(\sqrt{n})$  parties). Technically, the problem with using an argument similar to the case  $\ell = 1$  is that we cannot apply Azuma's inequality as before, because we do not have an upper bound on the absolute value of each variable.

Nevertheless, we are able to prove that at least one of the two adversaries succeeds with high probability, while corrupting  $\tilde{O}(\sqrt{n})$  parties. This argument is more complicated, but the main idea is to define another “adversary”, “in between”  $\mathcal{A}_0$  and  $\mathcal{A}_1$ . (In the actual proof we refer to that adversary as a random walk). The new adversary is defined similarly to  $\mathcal{A}_0$  and  $\mathcal{A}_1$ , but instead of minimizing  $\alpha'_v$  (or maximizing  $\beta'_v$ ) it tries to minimize  $\beta'_v - \alpha'_v$  (after they were defined by the definitions of  $\mathcal{A}_0$  and  $\mathcal{A}_1$ ). Very roughly speaking, since the new adversary is “sandwiched” between  $\mathcal{A}_0$  and  $\mathcal{A}_1$ , we are able to apply Azuma's theorem for the new adversary and to derive a contradiction. Technically, the contradiction is derived by showing that if  $\alpha'_{\text{root}}$  is not close to 0 and  $\beta'_{\text{root}}$  is not close to 1, then the new adversary gets (with high probability) to a leaf that is labeled with neither 0 nor 1.

The full proof is the technical heart of the paper and is given in Section 4.

## 2 Definitions & Preliminaries

For an integer  $n \in \mathbb{N}$ , we denote by  $[n]$  the set  $\{1, \dots, n\}$ . Throughout the paper, we denote by  $\Pi$  a collective coin-flipping protocol, denote by  $n \in \mathbb{N}$  the number of parties participating in the protocol, and denote the parties by  $P_1, \dots, P_n$ . We assume that  $\Pi$ , when executed honestly, outputs the bit 0 (and similarly for 1) with probability  $\Omega(1)$ .

**Communication model.** The full information model [7] is a synchronous model. Namely, each protocol consists of *rounds* in which parties send messages. There exists a global counter which synchronizes parties in between rounds but they are asynchronous within a round. The parties communicate via a broadcast channel.

We define two restricted types of protocols: single-bit and single-turn.

► **Definition 2** (Single-bit protocol). We say that a protocol is a *single-bit protocol* for  $n$  parties if this protocol is executed in rounds such that in each round each party sends a single random bit.

► **Definition 3** (Single-turn protocol). We say that a protocol is a *single-turn protocol* for  $n$  parties if this protocol is executed in  $n$  rounds such that party  $P_i$  sends a single (possibly long) message at round  $i$ .

The above two restricted families of protocols can be naturally described by a game tree (of arity two in the single-bit case and bigger arity in the single-turn case) whose leaves are labeled by 0 and 1, and whose internal nodes (including the root) are labeled by numbers in  $[n]$ .

Without loss of generality, we restrict our attention to public-coin protocols.

► **Definition 4** (Public-coin protocol). A protocol is *public-coin* if each honest party broadcasts all of the randomness he generates (i.e., his “local coin-flips”), and does not send any other messages.

**Corruption model.** We consider the Byzantine model, where a bound  $t = t(n) \leq n$  is specified, and the adversary is allowed to corrupt up to  $t$  parties. The adversary can see the entire transcript (i.e., all the messages sent thus far), has full control over all the corrupted parties, and can broadcast any messages on their behalf. Moreover, the adversary has control over the order of the messages sent within each round of the protocol (i.e., “rushing”).

Within this model, two main types of adversaries were considered in the literature: *static* adversaries, who need to specify the parties they corrupt *before* the protocol begins, and *adaptive* adversaries, who can corrupt the parties *adaptively* based on the transcript so far. We focus on adaptive adversaries

► **Definition 5** (Adaptive adversary). Within each round, the adversary chooses parties one-by-one to send their messages; and he can perform corruptions at any point during this process based on the messages sent thus far and the protocol specification.

**Security.** The security of a collective coin-flipping protocol is usually measured by the extent to which an adversary can, by corrupting a subset of parties, bias the protocol outcome towards his desired bit.

► **Definition 6** ( $\epsilon$ -security). Fix  $\epsilon = \epsilon(n)$  and  $t = t(n)$ . A coin-flipping protocol  $\Pi$  is  $\epsilon$ -secure against  $t$  adaptive corruptions if for all  $n \in \mathbb{N}$ , it holds that for any adaptive adversary  $\mathcal{A}$  that corrupts at most  $t$  parties,

$$\min \left\{ \Pr [\text{Output of } \mathcal{A}(\Pi) = 0], \Pr [\text{Output of } \mathcal{A}(\Pi) = 1] \right\} \geq \epsilon(n),$$

where “Output of  $\mathcal{A}(\Pi)$ ” is a random variable that corresponds to the output of the protocol  $\Pi$  when executed in the presence of the adversary  $\mathcal{A}$ .

We next define a secure protocol as one where an adversary cannot “almost always” get the outcome he wants.

► **Definition 7** (Security). A coin-flipping protocol is secure against  $t = t(n)$  corruptions if it is  $\epsilon$ -secure against  $t$  corruptions for some constant  $\epsilon \in (0, 1)$ .

## 2.1 Azuma’s Inequality

We state Azuma’s inequality which is extensively used in our proofs. This formulation is standard and can be found, for example, in Alon-Spencer [4] and in Dubhashi-Panconesi [13].

► **Theorem 8.** Let  $X_1, \dots, X_N$  be random variables, such that for every  $i \in [N]$ ,  $|X_i| \leq \epsilon_i$ . If for every  $i \in [N]$  it holds that  $\mathbb{E}[X_i \mid X_1, \dots, X_{i-1}] \leq 0$ , then for any  $s \geq 0$ ,

$$\Pr \left[ \sum_{i=1}^N X_i \geq s \right] \leq 2 \cdot e^{-\frac{s^2}{2 \sum_{i=1}^N \epsilon_i^2}}$$



Similarly, if for every  $i \in [N]$  it holds that  $\mathbb{E}[X_i \mid X_1, \dots, X_{i-1}] \geq 0$ , then for any  $s \geq 0$ ,

$$\Pr \left[ \sum_{i=1}^N X_i \leq -s \right] \leq 2 \cdot e^{-\frac{s^2}{2 \sum_{i=1}^N \epsilon_i^2}}$$

### 3 A Lower Bound for Single-Bit Single-Turn Protocols

In this section, we give a simplified proof for the following theorem, originally proved in [19]

► **Theorem 9** ([19]). *There does not exist a single-bit single-turn collective coin-flipping protocol that is resilient to more than  $\tilde{\Omega}(\sqrt{n})$  adaptive corruptions.*

**Proof.** Fix any single-bit single-turn collective coin-flipping protocol  $\Pi$ . Consider the binary protocol tree of depth  $n$  corresponding to  $\Pi$ . We construct an adversary  $\mathcal{A}_0$  that with probability  $1 - o(1)$ , biases the outcome towards 0 while corrupting at most  $\tilde{O}(\sqrt{n})$  players.<sup>7</sup>

For each node  $v$  in the protocol tree, we associate a sequence of bits  $b_1, \dots, b_i$  that lead to it from the root of the tree, and a value  $\alpha_v$  which stands for the probability that the outcome of the protocol is 0, when executed *honestly* starting from the node  $v$ . Namely,  $\alpha_v \triangleq \Pr[\Pi^v = 0]$ , where  $\Pi^v$  is a random variable that corresponds to the output of the protocol  $\Pi$  when executed honestly starting from node  $v$ . Let  $p_0 \triangleq \Pr[\Pi^{\text{root}} = 0] \geq \Theta(1)$  be the probability that the protocol, executed honestly from the root, outputs 0. Further, observe that for every leaf  $v$  that is labeled by  $b \in \{0, 1\}$ , it holds that  $\alpha_v = 1 - b$ .

Let  $\epsilon \triangleq \frac{1}{\sqrt{n} \cdot \log^2 n}$ . Given that the protocol is in node  $v$ , the adversary  $\mathcal{A}_0$  computes two values

$$\alpha_v^{\min} = \min\{\alpha_{v0}, \alpha_{v1}\} \quad \text{and} \quad \alpha_v^{\max} = \max\{\alpha_{v0}, \alpha_{v1}\},$$

where  $\alpha_{v0}$  is the value associated with the left child of  $v$  and  $\alpha_{v1}$  is the value associated with the right child of  $v$ . Note that  $\alpha_v = (\alpha_{v0} + \alpha_{v1})/2$ . If  $\alpha_v \geq \alpha_v^{\min} + \epsilon$  (or, equivalently,  $\alpha_v \leq \alpha_v^{\max} - \epsilon$ ), then the adversary corrupts the party that is associated with node  $v$  and sends  $b \in \{0, 1\}$  such that  $\alpha_{vb} > \alpha_{v\bar{b}}$  (where  $\bar{b} = 1 - b$ ). Otherwise, if  $\alpha_v^{\max} - \epsilon < \alpha_v < \alpha_v^{\min} + \epsilon$ , then the adversary  $\mathcal{A}_0$  does not corrupt the corresponding party and lets it send a random bit. This completes the description of the adversary.

We next show that with overwhelming probability over the execution of the protocol with the adversary  $\mathcal{A}_0$ , the leaf with which the protocol concludes is a leaf that is labeled with 0. In addition, with overwhelming probability, the number of corruptions along the way is bounded by  $1/\epsilon$ .

Let  $(b_1, \dots, b_n) \in \{0, 1\}^n$  be a random variable corresponding to the  $n$  bits sent during the execution of the protocol  $\Pi_{\mathcal{A}_0}$ . Namely, if  $\mathcal{A}_0$  corrupts the party sending the  $i$ 'th bit in the protocol  $\Pi$ , given that the previous  $i - 1$  bits sent were  $(b_1, \dots, b_{i-1})$ , and sends the bit  $b^* \in \{0, 1\}$  on its behalf, then we set  $b_i = b^*$ . Otherwise, if  $\mathcal{A}_0$  does not corrupt this party, then  $b_i$  is randomly chosen in  $\{0, 1\}$ . Every prefix of the  $n$  bits  $b_1, \dots, b_n$  sent during the course of the protocol, corresponds to a node  $v$  in the protocol tree. Thus, we can write  $\alpha_{b_1, \dots, b_i}$  for  $\alpha_v$ , where the vertex  $v$  corresponds to the path  $b_1, \dots, b_i$  from the root to  $v$  in the protocol tree. Let  $\delta_i$  be a random variable defined as

$$\delta_i \triangleq \alpha_{b_1, \dots, b_i} - \alpha_{b_1, \dots, b_{i-1}}.$$

<sup>7</sup> One can analogously construct an adversary  $\mathcal{A}_1$  that with probability  $1 - o(1)$ , biases the outcome towards 1 while corrupting at most  $\tilde{O}(\sqrt{n})$  players.



Denote by  $I \subseteq [n]$  the set of indices in which the adversary  $\mathcal{A}_0$  corrupts the corresponding party. It holds that

$$\sum_{i=1}^n \delta_i = \sum_{i \in I} \delta_i + \sum_{i \notin I} \delta_i = \alpha_{b_1, \dots, b_n} - \alpha_{\text{root}}. \quad (1)$$

We first argue that with overwhelming probability  $\sum_{i \notin I} \delta_i \leq o(1)$ .

► **Claim 10.**  $\Pr \left[ \left| \sum_{i \notin I} \delta_i \right| \geq \frac{1}{\log n} \right] \leq \text{negl}(n)$ .

**Proof.** Define  $n$  random variables  $X_1, \dots, X_n$  as follows: For every  $i \in I$  we set  $X_i = 0$ , and for every  $i \notin I$  we define  $X_i = \delta_i$ . Note that for every  $i \in [n]$ , it holds that  $|X_i| \leq \epsilon$  and

$$\mathbb{E}[X_i \mid X_1, \dots, X_{i-1}] = 0.$$

Thus, by Azuma's inequality, for any  $s > 0$ ,

$$\Pr \left[ \left| \sum_{i=1}^n X_i \right| \geq s \right] \leq 4 \cdot e^{-\frac{s^2}{2n\epsilon^2}}.$$

Setting  $s = \epsilon \cdot \sqrt{n} \cdot \log n = \frac{1}{\log n}$ , we conclude that

$$\Pr \left[ \left| \sum_{i=1}^n X_i \right| \geq \frac{1}{\log n} \right] \leq \text{negl}(n). \quad \blacktriangleleft$$

We condition on the event that  $\left| \sum_{i \notin I} \delta_i \right| \leq \frac{1}{\log n}$  occurs. Also, recall that  $\alpha_{\text{root}} = \Theta(1)$ . Plugging these into Equation (1), we get that

$$\alpha_{b_1, \dots, b_n} \geq \alpha_{\text{root}} + \sum_{i \in I} \delta_i - \frac{1}{\log n}.$$

By the definition of  $\mathcal{A}_0$ , whenever it corrupts an index  $i$ , it causes  $\delta_i$  to be positive. Thus,

$$\alpha_{b_1, \dots, b_n} \geq \alpha_{\text{root}} - \frac{1}{\log n}.$$

This implies that  $\alpha_{b_1, \dots, b_n} = 1$  since  $\alpha_{b_1, \dots, b_n} \in \{0, 1\}$  and  $\alpha_{\text{root}} \geq \Omega(1)$ .

We proceed with the bound on the number of corruptions made by  $\mathcal{A}_0$ . By Equation (1), the fact that  $\alpha_{\text{root}} \in (0, 1)$ , and that with overwhelming probability  $\left| \sum_{i \notin I} \delta_i \right| \leq \frac{1}{\log n}$  and  $\alpha_{b_1, \dots, b_n} = 1$ , it holds that (with overwhelming probability)

$$\sum_{i \in I} \delta_i = \alpha_{b_1, \dots, b_n} - \alpha_{\text{root}} - \sum_{i \notin I} \delta_i \leq 1.$$

Since for each  $i \in I$ , it holds that  $\delta_i \geq \epsilon$ , the number of corruptions is bounded by  $1/\epsilon$ , as required.  $\blacktriangleleft$

## 4 A Lower Bound for Single-Turn Protocols

In this section, we prove our lower bound for single-turn collective coin-flipping protocols.

► **Theorem 11.** *There does not exist a single-turn collective coin-flipping protocol that is resilient to more than  $\tilde{\Omega}(\sqrt{n})$  adaptive corruptions.*

**Proof.** Fix any single-turn collective coin-flipping protocol  $\Pi$ . Since we are in the full information model, we can assume without loss of generality that the protocol is public-coin (see Definition 4). Namely, each player sends a *random* message from some universe  $\{0, 1\}^\ell$ . We denote  $L \triangleq 2^\ell$ .

Consider the  $L$ -ary protocol tree corresponding to  $\Pi$ . We define two adversaries  $\mathcal{A}_0$  and  $\mathcal{A}_1$ , where  $\mathcal{A}_0$  tries to bias the output towards 0 and  $\mathcal{A}_1$  tries to bias the output towards 1. We prove that at least one of these adversaries succeeds with probability  $1 - o(1)$  while corrupting at most  $\sqrt{n} \cdot \text{polylog}(n)$  players.

To this end, we associate with each node  $v$  in the protocol tree, three pairs of values

$$(\alpha_v, \beta_v), (c_v^0, c_v^1), \text{ and } (p_v^0, p_v^1).$$

Intuitively,  $\alpha_v$  is the expectation of the outcome of the protocol in the presence of the adversary  $\mathcal{A}_0$ , given that we are at node  $v$ , and  $\beta_v$  is the expectation of the outcome of the protocol in the presence of the adversary  $\mathcal{A}_1$ , given that we are at node  $v$ . The pair  $(c_v^0, c_v^1)$  is a pair of bits, where  $c_v^0 = 1$  if and only if  $\mathcal{A}_0$  corrupts node  $v$ , and  $c_v^1 = 1$  if and only if  $\mathcal{A}_1$  corrupts node  $v$ .<sup>8</sup> The pair  $(p_v^0, p_v^1)$  are a pair of “penalty” values. Intuitively, the penalty  $p_v^0$  (resp.  $p_v^1$ ) is proportional to the expected number of corruptions the adversary  $\mathcal{A}_0$  (resp.  $\mathcal{A}_1$ ) does, from node  $v$  onwards.

The penalty values  $\{p_v^0\}_{v \in V}$ , along with the values  $\{\alpha_v\}_{v \in V}$ , are used by the adversary  $\mathcal{A}_0$  to decide which nodes to corrupt (i.e., for which nodes  $v$  to set  $c_v^0 = 1$ ). Similarly, the penalty values  $\{p_v^1\}_{v \in V}$ , along with the values  $\{\beta_v\}_{v \in V}$ , are used by the adversary  $\mathcal{A}_1$  to decide which nodes to corrupt (i.e., for which nodes  $v$  to set  $c_v^1 = 1$ ).

Formally, the values  $(\alpha_v, \beta_v)$ ,  $(c_v^0, c_v^1)$ , and  $(p_v^0, p_v^1)$  are defined by induction starting from the leaves. For any leaf  $v$  labeled by 0 we define  $\alpha_v = \beta_v = 0$ , and for any leaf  $v$  labeled by 1 we define  $\alpha_v = \beta_v = 1$ . For all leaves  $v$  we define  $c_v^0 = c_v^1 = 0$  and  $p_v^0 = p_v^1 = 0$ .

Let  $k = \sqrt{n} \cdot \log^3 n$  and let  $\epsilon = \frac{1}{k}$ . For a non-leaf node  $v$ , suppose that its  $L$  children are associated with

$$\{(\alpha_i, \beta_i)\}_{i=1}^L \text{ and } \{(p_i^0, p_i^1)\}_{i=1}^L.$$

For every  $i \in [L]$ , define

$$\alpha'_i = \alpha_i + p_i^0 \text{ and } \beta'_i = \beta_i - p_i^1.$$

Let

$$\alpha_{\text{avg}} \triangleq \frac{1}{L} \cdot \sum_{i=1}^L \alpha_i \quad , \quad \alpha'_{\text{avg}} \triangleq \frac{1}{L} \cdot \sum_{i=1}^L \alpha'_i \quad , \quad \alpha'_{\min} \triangleq \min\{\alpha'_1, \dots, \alpha'_L\}$$

and let

$$\beta_{\text{avg}} \triangleq \frac{1}{L} \cdot \sum_{i=1}^L \beta_i \quad , \quad \beta'_{\text{avg}} \triangleq \frac{1}{L} \cdot \sum_{i=1}^L \beta'_i \quad , \quad \beta'_{\max} = \max\{\beta'_1, \dots, \beta'_L\}$$

If  $\alpha'_{\min} \leq \alpha'_{\text{avg}} - \epsilon$ , then set  $c_v^0 = 1$ . In this case, if the protocol arrives at node  $v$ , then the adversary  $\mathcal{A}_0$  corrupts node  $v$  and proceeds to its child  $i^*$  with minimal  $\alpha'_i$ ; i.e.,  $i^* = \text{argmin}_{i \in \text{child}(v)} \{\alpha'_i\}$ , and we set  $\alpha_v = \alpha_{i^*}$  and  $p_v^0 = p_{i^*}^0 + \epsilon$ . Otherwise, set  $c_v^0 = 0$ . In this

<sup>8</sup> When we say that an adversary corrupts node  $v$  we mean that it corrupts the party associated with node  $v$ .

case, the adversary  $\mathcal{A}_0$  does not corrupt node  $v$ , and we set  $\alpha_v = \alpha_{\text{avg}}$  and  $p_v^0 = \frac{1}{L} \cdot \sum_{i=1}^L p_i^0$ . We denote

$$\alpha'_v = \alpha_v + p_v^0.$$

Similarly, if  $\beta'_{\text{max}} \geq \beta'_{\text{avg}} + \epsilon$ , then set  $c_v^1 = 1$ . In this case, if the protocol arrives at node  $v$ , then the adversary  $\mathcal{A}_1$  corrupts node  $v$  and proceeds to its child  $i^*$  with maximal  $\beta'$ ; i.e.,  $i^* = \text{argmax}_{i \in \text{child}(v)} \{\beta'_i\}$ , and we set  $\beta_v = \beta_{i^*}$  and  $p_v^1 = p_{i^*}^1 + \epsilon$ . Otherwise, set  $c_v^1 = 0$ . In this case, the adversary  $\mathcal{A}_1$  does not corrupt node  $v$ , and we set  $\beta_v = \beta_{\text{avg}}$  and  $p_v^1 = \frac{1}{L} \cdot \sum_{i=1}^L p_i^1$ . We denote

$$\beta'_v = \beta_v - p_v^1.$$

In what follows, we denote by  $\alpha_{\text{root}}$  and  $\beta_{\text{root}}$  the  $\alpha$  and  $\beta$  values of the root, respectively. Similarly, we denote by  $\alpha'_{\text{root}}$  and  $\beta'_{\text{root}}$  the  $\alpha'$  and  $\beta'$  values of the root, respectively. We denote by  $p_{\text{root}}^0$  and  $p_{\text{root}}^1$  the penalty values of the root.

The following claim follows immediately from the definition of  $p_{\text{root}}^0$  and  $p_{\text{root}}^1$ .

► **Claim 12.** *For every  $b \in \{0, 1\}$ , it holds that*

$$p_{\text{root}}^b = \frac{1}{k} \cdot \mathbb{E}[\# \text{ of corruptions } \mathcal{A}_b \text{ makes}].$$

In what follows, we denote by  $\Pi_{\mathcal{A}_0}$  the random variable which is the outcome of protocol  $\Pi$  with adversary  $\mathcal{A}_0$ , and similarly we denote by  $\Pi_{\mathcal{A}_1}$  the random variable which is the outcome of protocol  $\Pi$  with adversary  $\mathcal{A}_1$  (in both  $\Pi_{\mathcal{A}_0}$  and  $\Pi_{\mathcal{A}_1}$  the randomness is over the coin tosses of the honest players). In order to complete the proof of the theorem it suffices to prove the following two lemmas.

► **Lemma 13.**  $\Pr[\Pi_{\mathcal{A}_0} = 0] = 1 - \alpha_{\text{root}}$  and  $\Pr[\Pi_{\mathcal{A}_1} = 1] = \beta_{\text{root}}$ .

► **Lemma 14.**  $\alpha'_{\text{root}} = o(1)$  or  $\beta'_{\text{root}} = 1 - o(1)$ .

The reason why these two lemmas suffice is that for any node  $v$  in the protocol tree (and in particular for the root),  $\alpha_v \leq \alpha'_v$  and  $\beta_v \geq \beta'_v$ . Thus, the two lemmas imply that either

$$\Pr[\Pi_{\mathcal{A}_0} = 0] = 1 - o(1) \quad \text{or} \quad \Pr[\Pi_{\mathcal{A}_1} = 1] = 1 - o(1).$$

Moreover, by definition,  $\alpha'_{\text{root}} = \alpha_{\text{root}} + p_{\text{root}}^0$  and  $\beta'_{\text{root}} = \beta_{\text{root}} - p_{\text{root}}^1$ . Thus, if  $\alpha'_{\text{root}} = o(1)$  then Claim 12, together with the fact that  $\alpha_{\text{root}} \geq 0$  (see Lemma 13), implies that the adversary  $\mathcal{A}_0$  is expected to make only  $o(k)$  corruptions. By Markov's inequality  $\mathcal{A}_0$  makes  $o(k)$  corruptions with probability  $1 - o(1)$ . Similarly, if  $\beta'_{\text{root}} = 1 - o(1)$  then Claim 12, together with the fact that  $\beta_{\text{root}} \leq 1$  (see Lemma 13), implies that the adversary  $\mathcal{A}_1$  is expected to make only  $o(k)$  corruptions. By Markov's inequality  $\mathcal{A}_1$  makes  $o(k)$  corruptions with probability  $1 - o(1)$ . Since we set  $k = \sqrt{n} \cdot \log^3 n$ , this completes the proof.

We proceed with the proof of Lemma 13, followed by the proof of Lemma 14.

## 4.1 Proof of Lemma 13

We prove the more general statement that for *any* node  $v$  in the protocol tree, the probability that  $\Pi_{\mathcal{A}_0} = 0$  (respectively,  $\Pi_{\mathcal{A}_1} = 1$ ), conditioned on the event that the protocol arrives at node  $v$ , is  $1 - \alpha_v$  (respectively,  $\beta_v$ ). To this end, for any node  $v$  in the protocol tree, denote by  $\Pi^v$  the protocol execution starting from node  $v$ . We prove that for every node  $v$ ,

$$\Pr[\Pi_{\mathcal{A}_0}^v = 0] = 1 - \alpha_v \quad \text{and} \quad \Pr[\Pi_{\mathcal{A}_1}^v = 1] = \beta_v. \quad (2)$$

The proof is by induction from the leaves to the root. For leaf nodes, Equation (2) holds trivially. Suppose that Equation (2) holds for nodes at layer  $d+1$ , and we shall prove that it holds for nodes at layer  $d$ . To this end, fix a node  $v$  at layer  $d$ , and denote its  $L$  (layer  $d+1$ ) children by  $u_1, \dots, u_L$ . To be consistent with our previous notation, we denote  $\alpha_i \triangleq \alpha_{u_i}$  and let  $\alpha_{\text{avg}} = \frac{1}{L} \sum_{i=1}^L \alpha_i$ . We show that  $\Pr[\Pi_{\mathcal{A}_0}^v = 0] = 1 - \alpha_v$  and mention that the proof that  $\Pr[\Pi_{\mathcal{A}_1}^v = 1] = \beta_v$  is analogous.

We distinguish between two cases:

- **Case 1:**  $c_v^0 = 0$ . This case corresponds to the case where  $\mathcal{A}_0$  does not corrupt node  $v$ . In this case,

$$\Pr[\Pi_{\mathcal{A}_0}^v = 0] = \frac{1}{L} \sum_{i=1}^L \Pr[\Pi_{\mathcal{A}_0}^{u_i} = 0] = \frac{1}{L} \sum_{i=1}^L (1 - \alpha_i) = 1 - \alpha_{\text{avg}} = 1 - \alpha_v,$$

where the second equality follows from the induction assumption, and the other equalities follow from the definition of  $\mathcal{A}_0$ ,  $\alpha_{\text{avg}}$  and  $\alpha_v$ .

- **Case 2:**  $c_v^0 = 1$ . This case corresponds to the case where  $\mathcal{A}_0$  corrupts node  $v$ . We denote by  $i^*$  the child with minimal  $\alpha'$ . In this case,

$$\Pr[\Pi_{\mathcal{A}_0}^v = 0] = \Pr[\Pi_{\mathcal{A}_0}^{u_{i^*}} = 0] = (1 - \alpha_{i^*}) = 1 - \alpha_v,$$

where the second equality follows from our induction assumption, and the other equalities follow from the definition of  $\mathcal{A}_0$ , and  $\alpha_v$ .

This completes the proof of the lemma.

## 4.2 Proof of Lemma 14

Suppose towards contradiction that there exists a constant  $c > 0$  such that  $\alpha'_{\text{root}} > c$  and  $\beta'_{\text{root}} < 1 - c$ . We prove that at each layer of the circuit there exists a node  $v$  for which  $\alpha'_v > c - o(1)$  and  $\beta'_v < 1 - c + o(1)$ . This would imply a contradiction since at each leaf  $v$  it holds that either  $\alpha'_v = 0$  or  $\beta'_v = 1$ .

We define a random walk on the protocol tree from the root to the leaves. Since  $\Pi$  is a single turn protocol on  $n$  players, the protocol tree is of depth  $n$ . We denote the nodes on the walk by  $v_0, v_1, \dots, v_n$ , where  $v_0$  is the root and  $v_n$  is a leaf. The random walk is defined as follows:

1. Let  $V_1$  be the set of all nodes  $v$  such that for every child  $u \in \text{child}(v)$  it holds that

$$|\alpha'_u - \alpha'_v| \leq \epsilon \cdot \log n \quad \text{and} \quad |\beta'_u - \beta'_v| \leq \epsilon \cdot \log n.$$

If we are at node  $v_i \in V_1$ , then  $v_{i+1}$  is a random child of  $v_i$ .

2. Let  $V_2$  be the set of all nodes that are not in  $V_1$ . If  $v_i \in V_2$ , then choose a child  $v_{i+1} \in \text{child}(v_i)$  that minimizes the value  $\beta'_u - \alpha'_u$ . Namely,  $v_{i+1} = \text{argmin}_{u \in \text{child}(v_i)} \{\beta'_u - \alpha'_u\}$ .

Recall that in order to get a contradiction, it suffices to prove that with overwhelming probability,  $\alpha'_{v_n} \geq c - o(1)$  and  $\beta'_{v_n} \leq 1 - c + o(1)$ . To this end, we define  $n$  random variables  $X_1, \dots, X_n$ , and  $n$  random variables  $Y_1, \dots, Y_n$ , as follows:

$$X_{i+1} = \alpha'_{v_{i+1}} - \alpha'_{v_i} \quad \text{and} \quad Y_{i+1} = \beta'_{v_{i+1}} - \beta'_{v_i}.$$

Notice that

$$\alpha'_{v_n} = \alpha'_{v_0} + \sum_{i=1}^n X_i \quad \text{and} \quad \beta'_{v_n} = \beta'_{v_0} + \sum_{i=1}^n Y_i.$$

To get a contradiction it suffices to prove that for any constant  $t > 0$ , with overwhelming probability (over the random walk)

$$\sum_{i=1}^n X_i \geq -t \quad \text{and} \quad \sum_{i=1}^n Y_i \leq t. \quad (3)$$

To this end, we partition the set  $[n]$  into two sets  $I_1, I_2 \subseteq [n]$ , such that  $i \in I_b$  if and only if  $v_i \in V_b$  for  $b \in \{0, 1\}$  and  $i \in [n]$  and where  $V_1$  and  $V_2$  are the sets defined above. Namely,

$$I_1 = \{i : v_i \in V_1\} \quad \text{and} \quad I_2 = \{i : v_i \in V_2\}.$$

In order to prove Equation (3), it suffices to prove the following two claims.

► **Claim 15.** For any constant  $t > 0$ , with overwhelming probability (over the random walk),

$$\sum_{i \in I_1} X_i \geq -t \quad \text{and} \quad \sum_{i \in I_1} Y_i \leq t.$$

► **Claim 16.** For any constant  $t > 0$ , with overwhelming probability (over the random walk),

$$\sum_{i \in I_2} X_i \geq -t \quad \text{and} \quad \sum_{i \in I_2} Y_i \leq t. \quad (4)$$

We start by stating the following claim which we will use in the proofs of Claims 15 and 16.

► **Claim 17.** For every node  $v$  in the protocol tree,

$$\alpha'_v \leq \alpha'_{\text{avg}} \quad \text{and} \quad \beta'_v \geq \beta'_{\text{avg}},$$

where  $\alpha'_{\text{avg}}$  denotes the average of the values of  $\{\alpha'_u\}_{u \in \text{child}\{v\}}$  over the children of  $v$ , and  $\beta'_{\text{avg}}$  denotes the average of the values of  $\{\beta'_u\}_{u \in \text{child}\{v\}}$  over the children of  $v$ .

**Proof of Claim 17.** Fix a node  $v$  in the protocol tree. We show that  $\alpha'_v \leq \alpha'_{\text{avg}}$  and note that the proof that  $\beta'_v \geq \beta'_{\text{avg}}$  is analogous.

If  $c_v^0 = 0$ , then  $\alpha'_v = \alpha'_{\text{avg}}$  and the claim holds. Suppose that  $c_v^0 = 1$ . In this case,  $\alpha'_v = \alpha'_{\min} + \epsilon$ , where  $\alpha'_{\min} = \min_{u \in \text{child}\{v\}} \{\alpha'_u\}$  is the minimal value of  $\alpha'$  over all the children of  $v$ . Also, by definition,  $\alpha'_{\min} \leq \alpha'_{\text{avg}} - \epsilon$ . Thus,  $\alpha'_v \leq \alpha'_{\text{avg}} - \epsilon + \epsilon = \alpha'_{\text{avg}}$ , as desired. ◀

**Proof of Claim 15.** By definition, for every  $i \in I_1$ ,  $|X_i|, |Y_i| \leq \epsilon \cdot \log n$ . Claim 17 implies that

$$\mathbb{E}[X_i \mid X_1, \dots, X_{i-1}] \geq 0 \quad \text{and} \quad \mathbb{E}[Y_i \mid Y_1, \dots, Y_{i-1}] \leq 0.$$

We extend the series of random variables  $(X_i)_{i \in I_1}$  and  $(Y_i)_{i \in I_1}$ , and define two sequences of  $n$  random variables

$$(X'_1, \dots, X'_n) \quad \text{and} \quad (Y'_1, \dots, Y'_n)$$

such that for every  $i \in [n]$  it holds that

$$X'_i = \begin{cases} X_i & \text{if } i \in I_1 \\ 0 & \text{otherwise} \end{cases} \quad \text{and} \quad Y'_i = \begin{cases} Y_i & \text{if } i \in I_1 \\ 0 & \text{otherwise.} \end{cases}$$

### 34:14 A Lower Bound for Adaptively-Secure Collective Coin-Flipping Protocols

Note that (by Claim 17) it still holds that for every  $i \in [n]$ ,

$$\mathbb{E}[X'_i \mid X'_1, \dots, X'_{i-1}] \geq 0 \quad \text{and} \quad \mathbb{E}[Y'_i \mid Y'_1, \dots, Y'_{i-1}] \leq 0.$$

Thus, by Azuma's inequality (see Theorem 8), for any real  $s > 0$ ,

$$\Pr \left[ \sum_{i=1}^n X'_i \leq -s \right] \leq 2 \cdot e^{-\frac{s^2}{2n(\epsilon \cdot \log n)^2}} \quad \text{and} \quad \Pr \left[ \sum_{i=1}^n Y'_i \geq s \right] \leq 2 \cdot e^{-\frac{s^2}{2n(\epsilon \cdot \log n)^2}}.$$

By definition  $\sum_{i \in I_1} X_i = \sum_{i=1}^n X'_i$  and  $\sum_{i \in I_1} Y_i = \sum_{i=1}^n Y'_i$  and thus

$$\Pr \left[ \sum_{i \in I_1} X_i \leq -\epsilon \cdot \sqrt{n} \cdot \log^2 n \right] = \text{negl}(n) \quad \text{and} \quad \Pr \left[ \sum_{i \in I_1} Y_i \geq \epsilon \cdot \sqrt{n} \cdot \log^2 n \right] = \text{negl}(n).$$

Since we set  $\epsilon = \frac{1}{k} = \frac{1}{\sqrt{n \cdot \log^3 n}}$ , we have that  $\epsilon \cdot \sqrt{n} \cdot \log^2 n = o(1)$ , which completes the proof.  $\blacktriangleleft$

We proceed with the proof of Claim 16. In the proof, we make use of the following two claims.

► **Claim 18.** *For any node  $v$  in the protocol tree and for any  $u \in \text{child}(v)$ ,*

$$\beta'_u \leq \beta'_v + \epsilon \quad \text{and} \quad \alpha'_u \geq \alpha'_v - \epsilon.$$

**Proof.** Fix any node  $v$  in the protocol tree and fix any child  $u \in \text{child}(v)$ . We prove that  $\beta'_u \leq \beta'_v + \epsilon$ . The proof that  $\alpha'_u \geq \alpha'_v - \epsilon$  is analogous and thus omitted.

We distinguish between two cases. First, if  $c_v^1 = 0$ , then  $\beta'_v = \beta'_{\text{avg}}$  and all the children of  $v$  have  $\beta'$  which is at most  $\beta'_{\text{avg}} + \epsilon$  which implies that  $\beta'_u \leq \beta'_v + \epsilon$ . Second, if  $c_v^1 = 1$ , then  $\beta'_v = \beta'_{\text{max}} - \epsilon$ , where  $\beta'_{\text{max}} = \max_{u \in \text{child}(v)} \{\beta'_u\}$  is the maximal  $\beta'$  of all the children of  $v$ . This also implies that  $\beta'_u \leq \beta'_v + \epsilon$ .  $\blacktriangleleft$

► **Claim 19.** *For every node  $v$  in the protocol tree, it holds that  $\beta'_v \geq \alpha'_v$ .*

**Proof.** The proof is by induction from the leaves to the root. For any leaf  $v$ , it holds that  $\beta'_v = \alpha'_v$  by definition, and in particular  $\beta'_v \geq \alpha'_v$ . Suppose that  $\beta'_v \geq \alpha'_v$  holds for every node  $v$  in layer  $d+1$  and we prove that it holds for every node in layer  $d$ .

To this end, fix any node  $v$  in layer  $d$ . Suppose that its  $L$  children (in layer  $d+1$ ) are associated with values  $\{(\alpha'_i, \beta'_i)\}_{i=1}^L$ , and denote

$$\alpha'_{\text{avg}} \triangleq \frac{1}{L} \sum_{i=1}^L \alpha'_i \quad \text{and} \quad \beta'_{\text{avg}} \triangleq \frac{1}{L} \sum_{i=1}^L \beta'_i.$$

The induction assumption implies that  $\beta'_{\text{avg}} \geq \alpha'_{\text{avg}}$ . This, together with Claim 17, implies that

$$\beta'_v \geq \beta'_{\text{avg}} \geq \alpha'_{\text{avg}} \geq \alpha'_v,$$

as desired.  $\blacktriangleleft$

**Proof of Claim 16.** We first show that for every  $i \in I_2$ ,

$$\beta'_{v_{i+1}} - \alpha'_{v_{i+1}} \leq (\beta'_{v_i} - \alpha'_{v_i}) - \epsilon \cdot (\log n - 1). \quad (5)$$

Fix any  $v_i \in V_2$ . By definition of  $V_2$ , there exists a child  $u \in \text{child}(v_i)$  such that

$$|\alpha'_u - \alpha'_{v_i}| \geq \epsilon \cdot \log n \quad \text{or} \quad |\beta'_u - \beta'_{v_i}| \geq \epsilon \cdot \log n.$$

Claim 18 implies that there exists a child  $u \in \text{child}(v_i)$  such that

$$\alpha'_u \geq \alpha'_{v_i} + \epsilon \cdot \log n \quad \text{or} \quad \beta'_u \leq \beta'_{v_i} - \epsilon \cdot \log n.$$

For concreteness, suppose that  $\alpha'_u \geq \alpha'_{v_i} + \epsilon \cdot \log n$  (the proof for  $\beta'_u \leq \beta'_{v_i} - \epsilon \cdot \log n$  is analogous). Claim 18 implies that  $\beta'_u \leq \beta'_{v_i} + \epsilon$ . These two inequalities imply that

$$\beta'_u - \alpha'_u \leq \beta'_{v_i} + \epsilon - \alpha'_{v_i} - \epsilon \cdot \log n = (\beta'_{v_i} - \alpha'_{v_i}) - \epsilon \cdot (\log n - 1).$$

This implies Inequality (5), since  $v_{i+1}$  was chosen to minimize the value of  $\beta'_{v_{i+1}} - \alpha'_{v_{i+1}}$ . Inequality (5) implies that, with overwhelming probability,

$$\begin{aligned} |I_2| \cdot \epsilon \cdot (\log n - 1) &\leq \sum_{i \in I_2} (\beta'_{v_i} - \alpha'_{v_i}) - (\beta'_{v_{i+1}} - \alpha'_{v_{i+1}}) \\ &\leq \sum_{i \in I_2} (\beta'_{v_i} - \alpha'_{v_i}) - (\beta'_{v_{i+1}} - \alpha'_{v_{i+1}}) + \\ &\quad \sum_{i \in I_1} (\beta'_{v_i} - \alpha'_{v_i}) - (\beta'_{v_{i+1}} - \alpha'_{v_{i+1}}) + 1 \\ &= (\beta'_{\text{root}} - \alpha'_{\text{root}}) - (\beta'_{v_n} - \alpha'_{v_n}) + 1 \leq 2, \end{aligned} \tag{6}$$

where the first inequality follows by Equation (5) and summing over all  $i \in I_2$ , the second inequality follows by Claim 15, and the last inequality follows by our assumption that  $\alpha'_{\text{root}} > c$  and  $\beta'_{\text{root}} < 1 - c$  together with Claim 19.

Note that Claim 18 implies that for every  $i \in [n]$ , it holds that  $X_i \geq -\epsilon$  and  $Y_i \leq \epsilon$ . This, together with Equation (6), implies that

$$\sum_{i \in I_2} X_i \geq -\epsilon \cdot |I_2| \geq -\frac{2}{\log n - 1} \quad \text{and} \quad \sum_{i \in I_2} Y_i \leq \epsilon \cdot |I_2| \leq \frac{2}{\log n - 1},$$

as desired. ◀

---

## References

- 1 Miklós Ajtai and Nathan Linial. The influence of large coalitions. *Combinatorica*, 13(2):129–145, 1993.
- 2 Bar Alon and Eran Omri. Almost-optimally fair multiparty coin-tossing with nearly three-quarters malicious. In *Theory of Cryptography - 14th International Conference, TCC 2016-B*, pages 307–335, 2016.
- 3 Noga Alon and Moni Naor. Coin-flipping games immune against linear-sized coalitions. *SIAM J. Comput.*, 22(2):403–417, 1993.
- 4 Noga Alon and Joel Spencer. *The Probabilistic Method*. John Wiley, third edition, 2008.
- 5 Amos Beimel, Iftach Haitner, Nikolaos Makriyannis, and Eran Omri. Tighter bounds on multi-party coin flipping, via augmented weak martingales and differentially private sampling. *Electronic Colloquium on Computational Complexity (ECCC)*, 24:168, 2017.
- 6 Amos Beimel, Eran Omri, and Ilan Orlov. Protocols for multiparty coin toss with a dishonest majority. *J. Cryptology*, 28(3):551–600, 2015.



- 7 Michael Ben-Or and Nathan Linial. Collective coin flipping. *Advances in Computing Research*, 5:91–115, 1989.
- 8 Ravi B. Boppana and Babu O. Narayanan. The biased coin problem. *SIAM J. Discrete Math.*, 9(1):29–36, 1996.
- 9 Niv Buchbinder, Iftach Haitner, Nissan Levi, and Eliad Tsfadia. Fair coin flipping: Tighter analysis and the many-party case. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 2580–2600. SIAM, 2017.
- 10 Richard Cleve. Limits on the security of coin flips when half the processors are faulty (extended abstract). In Juris Hartmanis, editor, *Proceedings of the 18th Annual ACM Symposium on Theory of Computing*, pages 364–369. ACM, 1986.
- 11 Richard Cleve and Russell Impagliazzo. Martingales, collective coin flipping and discrete control processes (extended abstract), 1993. Unpublished manuscript.
- 12 Yevgeniy Dodis. Impossibility of black-box reduction from non-adaptively to adaptively secure coin-flipping. *Electronic Colloquium on Computational Complexity (ECCC)*, 7(39), 2000.
- 13 Devdatt P. Dubhashi and Alessandro Panconesi. *Concentration of Measure for the Analysis of Randomized Algorithms*. Cambridge University Press, 2009. doi:10.1017/CB09780511581274.
- 14 Uriel Feige. Noncryptographic selection protocols. In *40th Annual Symposium on Foundations of Computer Science, FOCS*, pages 142–153, 1999.
- 15 Shafi Goldwasser, Yael Tauman Kalai, and Sunoo Park. Adaptively secure coin-flipping, revisited. In *42nd International Colloquium on Automata, Languages and Programming, ICALP*, pages 663–674, 2015.
- 16 Iftach Haitner and Eliad Tsfadia. An almost-optimally fair three-party coin-flipping protocol. *SIAM J. Comput.*, 46(2):479–542, 2017.
- 17 Jeff Kahn, Gil Kalai, and Nathan Linial. The influence of variables on boolean functions (extended abstract). In *29th Annual Symposium on Foundations of Computer Science, FOCS*, pages 68–80, 1988.
- 18 Yael Tauman Kalai and Ilan Komargodski. Compressing communication in distributed protocols. In *Distributed Computing - 29th International Symposium, DISC*, pages 467–479, 2015.
- 19 David Lichtenstein, Nathan Linial, and Michael E. Saks. Some extremal problems arising from discrete control processes. *Combinatorica*, 9(3):269–287, 1989.
- 20 Tal Moran, Moni Naor, and Gil Segev. An optimally fair coin toss. *J. Cryptology*, 29(3):491–513, 2016.
- 21 Alexander Russell, Michael E. Saks, and David Zuckerman. Lower bounds for leader election and collective coin-flipping in the perfect information model. *SIAM J. Comput.*, 31(6):1645–1662, 2002.
- 22 Michael E. Saks. A robust noncryptographic protocol for collective coin flipping. *SIAM J. Discrete Math.*, 2(2):240–244, 1989.

# Adapting Local Sequential Algorithms to the Distributed Setting

**Ken-ichi Kawarabayashi**

National Institute of Informatics, Tokyo, Japan  
k\_keniti@nii.ac.jp

**Gregory Schwartzman**

National Institute of Informatics, Tokyo, Japan  
greg@nii.ac.jp

---

## Abstract

It is a well known fact that sequential algorithms which exhibit a strong "local" nature can be adapted to the distributed setting given a legal graph coloring. The running time of the distributed algorithm will then be at least the number of colors. Surprisingly, this well known idea was never formally stated as a unified framework. In this paper we aim to define a robust family of local sequential algorithms which can be easily adapted to the distributed setting. We then develop new tools to further enhance these algorithms, achieving state of the art results for fundamental problems.

We define a simple class of greedy-like algorithms which we call *orderless-local* algorithms. We show that given a legal  $c$ -coloring of the graph, every algorithm in this family can be converted into a distributed algorithm running in  $O(c)$  communication rounds in the CONGEST model. We show that this family is indeed robust as both the method of conditional expectations and the unconstrained submodular maximization algorithm of Buchbinder *et al.* [10] can be expressed as orderless-local algorithms for *local utility functions* – Utility functions which have a strong local nature to them.

We use the above algorithms as a base for new distributed approximation algorithms for the weighted variants of some fundamental problems: Max  $k$ -Cut, Max-DiCut, Max 2-SAT and correlation clustering. We develop algorithms which have the same approximation guarantees as their sequential counterparts, up to a constant additive  $\epsilon$  factor, while achieving an  $O(\log^* n)$  running time for deterministic algorithms and  $O(\epsilon^{-1})$  running time for randomized ones. This improves exponentially upon the currently best known algorithms.

**2012 ACM Subject Classification** Theory of computation → Distributed algorithms

**Keywords and phrases** Distributed, Approximation Algorithms, Derandomization, Max-Cut

**Digital Object Identifier** 10.4230/LIPIcs.DISC.2018.35

**Related Version** A full version of the paper is available at <https://arxiv.org/abs/1711.10155>.

**Funding** This work was supported by JST ERATO Grant Number JPMJER1201, Japan

**Acknowledgements** We would like to thank Ami Paz and Seri Khoury for many fruitful discussions and useful advice.



© Ken-ichi Kawarabayashi and Gregory Schwartzman;  
licensed under Creative Commons License CC-BY

32nd International Symposium on Distributed Computing (DISC 2018).

Editors: Ulrich Schmid and Josef Widder; Article No. 35; pp. 35:1–35:17

Leibniz International Proceedings in Informatics



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

■ **Table 1** Summary of our results for the CONGEST model ( $\tilde{O}$  hides factors polylogarithmic in  $\Delta$ ). (\*) General graphs, max-agree (†) Unweighted graphs, only Max-Cut ( $k = 2$ ).

Problem	Our Approx.	Our Time	Prev Approx.	Prev Time	Notes
Weighted Correlation-Clustering*	$1/2 - \epsilon$	$O(\log^* n)$	-	-	det.
Weighted Max $k$ -Cut	$1 - 1/k - \epsilon$	$O(\log^* n)$	$1/2$ [12]†	$\tilde{O}(\Delta + \log^* n)$	det.
Weighted Max-Dicut	$1/3 - \epsilon$	$O(\log^* n)$	$1/3$ [12]†	$\tilde{O}(\Delta + \log^* n)$	det.
Weighted Max-Dicut	$1/2 - \epsilon$	$O(\epsilon^{-1})$	$1/2$ [12]†	$\tilde{O}(\Delta + \log^* n)$	rand.
Weighted Max 2-SAT	$3/4 - \epsilon$	$O(\epsilon^{-1})$	-	-	rand.

## 1 Introduction

A large part of research in the distributed environment aims to develop fast distributed algorithms for problems which have already been studied in the sequential setting. Ideally, we would like to use the power of the distributed environment to achieve a substantial improvement in the running time over the sequential algorithm, and indeed, for many problems distributed algorithms achieve an exponential improvement over the sequential case. One approach to designing distributed algorithms is using the sequential algorithm as natural starting point [5–7, 12, 18], then certain adjustments are made for the distributed environment in order to achieve a faster running time.

There is a well known folklore in distributed computing, which roughly says that if a sequential graph algorithm works by traversing nodes in any order (perhaps adversarial), and for every node makes a local decision, then given a legal  $c$ -coloring of the graph, the algorithm can be adapted to the distributed setting by going over all color classes, and for each executing all nodes in the class simultaneously. Surprisingly, there is no formal framework describing the above. In this paper we provide such a framework for a specific class of algorithms (defined later).

We note that for general graphs a legal coloring may require at least  $\Delta + 1$  colors, where  $\Delta$  is the maximal degree of the graph. Using the above framework we aim to answer the following question: Are there certain classes of algorithms where using the above can result in a running time sublinear in  $\Delta$ ? We show that for certain approximation problems the answer is quite surprising, as we are able to achieve an almost constant running time!

More precisely, we show that for the problems of Max  $k$ -Cut, Max-DiCut, Max 2-SAT and correlation clustering we can adapt the sequential algorithm to these problems in such a way that the running time is  $O(\log^* n)$  rounds for deterministic algorithms and  $O(\epsilon^2)$  for randomized ones, while losing only an additive  $\epsilon$ -factor in the approximation ratio. For the problems of Max-Cut and Max-DiCut this greatly improves upon the previous best known results, which required a number of rounds linear in  $\Delta$ . A summary of our results appears in Table 1.

### 1.1 Tools and results

In this paper we focus our attention on approximation algorithms for unconstrained optimization problems on graphs. We are given some graph  $G(V, E)$ , where each vertex  $v$  is assigned a variable  $X_v$  taking values in some set  $A$ . We aim to maximize some utility function  $f$  over these variables (For a formal definition see Section 2). Our distributed model is the CONGEST model of distributed computation, where the network is represented by a graph, s.t nodes are computational units and edges are communication links. Nodes communicate in

synchronous communication rounds, where at each round a node sends and receives messages from all of its neighbors. In the CONGEST model the size of messages sent between nodes is limited to  $O(\log n)$  bits, where  $|V| = n$ . This is more restrictive than the LOCAL model, where message size is unbounded. Our complexity measure is the number of communication rounds of the algorithm.

Adapting a sequential algorithm of the type we describe above to the distributed setting, means we wish each node  $v$  in the communication graph to output an assignment to  $X_v$  such that the approximation guarantee is close to that of the sequential algorithm, while minimizing the number of communication rounds of the distributed algorithm. Our goal is to formally define a family of sequential algorithms which can be easily converted to distributed algorithms, and then develop tools to allow these algorithms to run exponentially faster, while achieving almost the same approximation ratio. To achieve this we focus our attention on a family of sequential algorithms which exhibit a very strong local nature.

We define a family of utility functions, which we call *local utility functions* (Formally defined in Section 2). We say that a utility function  $f$  is a local utility function, if the change to the value of the function upon setting one variable  $X_v$  can be computed locally. Intuitively, while optimizing a general utility function in the distributed setting might be difficult for global functions, the local nature of the family of local utility functions makes it a perfect candidate.

We focus on adapting a large family of, potentially randomized, local algorithms to the distributed setting. We consider *orderless-local* algorithms - algorithms that can traverse the variables in any order and in each iteration apply some local function to decide the value of the variable. By local we mean that the decision only depends on the local environment of the node in the graph, the variables of nodes adjacent to that variable and some randomness only used by that node. This is similar to the family of Priority algorithms first defined in [9]. The goal of [9] was to formally define the notion of a greedy algorithm, and then to explore the limits of these algorithms. Our definition is similar (and can be expressed as a special case of priority algorithms), but the goal is different. While [9] aims to prove lower bounds, we provide some sufficient conditions that allow us to easily transform local sequential algorithms into fast distributed algorithms.

Our definitions are also similar to the SLOCAL model [21], which also shows that sequential algorithms which traverse the graph vertices in any order and make local decisions can be adapted to the distributed LOCAL model in poly logarithmic rounds using randomization. While the results of [21] are much more broad, our transformation does not require any randomization and works in the CONGEST model. Finally, we should also mention the field of local computation algorithms [35] whose aim is developing efficient local *sequential* algorithms. We refer the reader to an excellent survey by Levi and Medina [30].

One might expect that due to the locality of this family of algorithms it can be distributed if the graph is provided with a legal coloring. The distributed algorithm goes over the color classes one after another and executes all nodes in the color class simultaneously. This solves any conflicts that may occur from executing two neighboring nodes, while the orderless property guarantees that this execution is valid. In a sense this argument was already used for specific algorithm (Coloring to MIS [32], MaxIS of [5], Max-Cut of [12]). We provide a more general result, using this classical argument. Specifically, we show that given a legal  $c$ -coloring, any orderless-local algorithm can be distributed in  $O(c)$ -communication rounds in the CONGEST model.

To show that this definition is indeed robust, we show two general applications. The first is adapting the method of conditional expectations (Formally defined in Section 2)

to the distributed setting. This method is inherently sequential, but we show that if the utility function optimized is a local utility function, then the algorithm is an orderless-local algorithm. A classical application of this technique is for Max  $k$ -cut, where an  $(1 - 1/k)$ -approximation is achieved when every node chooses a cut side at random. This can be derandomized using the method of conditional expectations, and adapted to the distributed setting, as the cut function is a local utility function. We note that the same exact approach results in a  $(1/2 - \epsilon)$ -approximation for max-agree correlation clustering on general graphs (see Section 2 for a definition). Because the tools used for Max-Cut directly translate to correlation clustering, we focus on Max-Cut for the rest of the paper, and only mention correlation clustering at the very end.

The second application is the unconstrained submodular maximization algorithms of [10], where a deterministic  $1/3$ -approximation and a randomized expected  $1/2$ -approximation algorithms are presented. We show that both are orderless-local algorithms when provided with a local utility function. This can be applied to the problem of Max-DiCut, as it is an unconstrained submodular function, and also a local utility function. The algorithms of [10] were already adapted to the distributed setting for the specific problem of Max-DiCut by [12] using similar ideas. The main benefit of our definition is the convenience and generality of adapting these algorithms without the need to consider their analysis or reprove correctness. We conclude that the family of orderless-local algorithms indeed contains robust algorithms for fundamental problems, and especially the method of conditional expectations.

At the time this paper was first made public, there was no distributed equivalent for the method of conditional expectations. We have since learned that, independently and simultaneously, an adaptation of the method of conditional expectations to the distributed setting was also presented in [20]. Their results show how the method of conditional expectations combined with a *legal* coloring can be used to convert any randomized LOCAL  $r$ -round algorithm for a locally checkable problem to a deterministic one, running in  $O(\Delta^{O(r)} + O(r \log^* n))$ .<sup>1</sup> This is done via a transformation to an SLOCAL algorithm, where the derandomization is applied and then transforming back to a LOCAL algorithm.

Although not stated for the CONGEST model, we believe it to be the case that when  $r = 1$  their application of the method of conditional expectations works in the CONGEST, and is equivalent to our results. Another difference apart from the different model of communication, is that they focus on derandomizing locally checkable problems, while we focus on local utility functions. These two families of problems are different, as the approximation guaranteed for a certain local utility function need not be locally checkable. This last point highlights the different goal of the two papers. While [20] skillfully show that a large family of LOCAL algorithm can be derandomized, we aim to adapt sequential algorithm to the distributed setting while achieving as *fast* of a running time as possible in the more restrictive CONGEST model – hence we focus on local utility function which capture the locality of the optimization process.

Next, we wish to consider the running time of these algorithms. Recall that we expressed the running time of orderless local algorithms in terms of the colors of some legal coloring for the graph. For a general graph, we cannot hope for a legal coloring using less than  $\Delta + 1$ , where  $\Delta$  is the maximum degree in the graph. This means that using the distributed version of an orderless-local algorithm unchanged will have a running time linear in  $\Delta$ . We show how

---

<sup>1</sup> They actually show that the running time is either  $O(\Delta^{O(r)} + O(r \log^* n))$  or  $r \cdot 2^{O(\sqrt{\log m})}$ , achieving the latter via network decomposition. We focus on the first bound, as the second is less relevant for the comparison which follows.

to overcome this obstacle for Max  $k$ -Cut and Max-DiCut. The general idea is to compute a *defective* coloring of the graph which uses few colors, drop all monochromatic edges, and call the algorithm for the new graph which now has a legal coloring.

A key tool in our algorithms is a new type of defective coloring we call a *weighted  $\epsilon$ -defective coloring*. The classical defective coloring allows each vertex to have at most  $d$  monochromatic edges, for some defect parameter  $d$ . We consider positively edge weighted graphs and require a weighted fractional defect - for every vertex the total weight of monochromatic edges is at most an  $\epsilon$ -fraction of the total weight of all edges of that vertex. We show that a weighted  $\epsilon$ -defective coloring using  $O(\epsilon^{-2})$  colors can be computed *deterministically* in  $O(\log^* n)$  rounds using the defective coloring algorithm of [29]. The classical algorithm of Kuhn was found useful in the adaptation of sequential algorithms to the distributed setting [17, 21], thus its effectiveness for weighted  $\epsilon$ -defective coloring might be of further use.

Although we cannot guarantee a legal coloring with a small number of colors for any graph  $G(V, E, w)$ , we may remove some subset of  $E$  which will result in a new graph  $G'$  with a low chromatic number. We wish to do so while not decreasing the total sum of edge weights in  $G'$ , which we prove guarantees the approximation will only be mildly affected for our cut problems. Formally, we show that if we only decrease the total edge weight by an  $\epsilon$ -fraction, we will incur an additive  $\epsilon$ -loss in the approximation ratio of the cut algorithms for  $G$ . For the randomized algorithm this is easy, simply color each vertex randomly with a color in  $[\lceil \epsilon^{-1} \rceil]$  and drop all monochromatic edges. For the deterministic case, we execute our weighted  $\epsilon$ -defective coloring algorithm, and then remove all monochromatic edges. We then execute the relevant cut algorithm on the resulting graph  $G'$  which now has a legal coloring, using a small number of colors. The above results in extremely fast approximation algorithms for weighted Max  $k$ -Cut and weighted Max-DiCut, while having almost the same approximation ratio as their sequential counterpart.

Finally, our techniques can also be applied to the problem of weighted Max 2-SAT. To do so we may use the randomized expected  $3/4$ -approximation algorithm presented in [34]. It is based on the algorithm of [10], and thus is almost identical to the unconstrained submodular maximization algorithm. Because the techniques we use are very similar to the above, we defer the entire proof to the full version of the paper.<sup>2</sup>

## 1.2 Previous research

**Cut problems:** An excellent overview of the Max-Cut and Max-DiCut problems appears in [12], which we follow in this section. Computing Max-Cut exactly is NP-hard as shown by Karp [27] for the weighted version, and by [19] for the unweighted case. As for approximations, it is impossible to improve upon a  $16/17$ -approximation for Max-Cut and a  $12/13$ -approximation for Max-DiCut unless  $P = NP$  [24, 38]. If every node chooses a cut side randomly, an expected  $1/2$ -approximation for Max-Cut, a  $1/4$ -approximation for Max-DiCut and a  $(1 - 1/k)$ -approximation is achieved. This can be derandomized using the method of conditional expectations. In the breakthrough paper of Goemans and Williamson [23] a  $0.878$ -approximation is achieved using semidefinite programming. This is optimal under the unique games conjecture [28]. In the same paper a  $0.796$ -approximation for Max-DiCut was presented. This was later improved to  $0.863$  in [MatuuraM01]. Other results using different techniques are presented in [26, 37].

<sup>2</sup> The full version can be found here: <https://arxiv.org/abs/1711.10155>.



In the distributed setting the problem has not received much attention. A node may choose a cut side at random, achieving the same guarantees as above in constant time. In [25] a distributed algorithm for  $d$ -regular triangle free graphs which achieves a  $(1/2 + 0.28125/\sqrt{d})$ -approximation ratio in a single communication round is presented. The only results for general graphs in the distributed setting is due to [12]. In the CONGEST model they present a deterministic  $1/2$ -approximation for Max-Cut, a deterministic  $1/3$ -approximation for Max-DiCut, and a randomized expected  $1/2$  approximation for Max-DiCut running in  $\tilde{O}(\Delta + \log^* n)$  communication rounds. The results for Max-DiCut follow from adapting the unconstrained submodular maximization algorithm of [10] to the distributed setting. Better results are presented for the LOCAL model; we refer the reader to [12] for the full details. Recently, a lower bound of  $O(1/\epsilon)$ -rounds in the LOCAL model for any (even randomized)  $(1 - \epsilon)$ -approximation algorithm for Max-Cut and Max-DiCut was presented in [8].

**Max 2-SAT:** The decision version of Max 2-SAT is NP-complete [19], and there exist several approximation algorithms [16, 23, 31, 33], of which currently the best known approximation ratio is 0.9401 [31]. In [3] it is shown that assuming the unique games conjecture, the approximation factor of [31] cannot be improved. Assuming only that  $P \neq NP$  it cannot be approximated to within a  $21/22$ -factor [24]. To the best of our knowledge the problem of Max 2-SAT (or Max-SAT) was not studied in the distributed model.

**Correlation clustering:** An excellent overview of correlation clustering (see Section 2 for a definition) appears in [1], which we follow in this section. Correlation clustering was first defined by [4]. Solving the problem exactly is NP-Hard, thus we are left with designing approximation algorithms for the problem, here one can try to approximate *max-agree* or *min-disagree*. If the graph is a clique, there exists a PTAS for max-agree [4, 22], and a 2.06-approximation for max-disagree [14]. For general (even weighted) graphs there exists a 0.7666-approximation for max-agree [13, 36], and a  $O(\log n)$ -approximation for min-disagree [15]. A trivial  $1/2$ -approximation for max-agree on general graphs can be achieved by considering putting every node in a separate cluster, then considering putting all nodes in a single cluster, and taking the more profitable of the two.

In the distributed setting little is known about correlation clustering. In [11] a dynamic distributed MIS algorithm is provided, it is stated that this achieves a 3-approximation for min-disagree correlation clustering as it simulates the classical algorithm of Ailon *et al.* [2]. We note that the algorithm of Ailon *et al.* assumes the graph to be a clique, thus the above result is limited to complete graphs where the edges of the communication graph are taken to be the positive edges, and the non-edges are taken as the negative edges (as indeed for general graphs, the problem is APX-Hard, and difficult to approximate better than  $\Theta(\log n)$  [15]). We also note that using only two clusters, where each node chooses a cluster at random, guarantees an expected  $1/2$ -approximation for max-agree on weighted general graphs. We derandomize this approach in this paper.

## 2 Preliminaries

**Sequential algorithms:** The main goal of this paper is converting (local) sequential graph algorithms for unconstrained maximization (or minimization) to distributed graph algorithms. Let us first define formally this family of algorithms. The sequential algorithm receives as input a graph  $G = (V, E)$ , we associate each vertex  $v \in V$  with a variable  $X_v$  taking values in some finite set  $A$ . The algorithm outputs a set of assignments  $\bar{X} = \{X_v = \alpha_v\}$ . The goal of



the algorithms is to maximize some utility function  $f(G, \bar{X})$  taking in a graph and the set of assignments and outputting some value in  $\mathbb{R}$ . For simplicity we assume that the order of the variables in  $\bar{X}$  does not affect  $f$ , so we use a set notation instead of a vector notation. We somewhat abuse notation, and when assigning a variable we write  $\bar{X} \cup \{X_v = \alpha\}$ , meaning that any other assignment to  $X_v$  is removed from the set  $\bar{X}$ . We also omit  $G$  as a parameter when it is clear from context.

When considering randomized algorithms we assume the algorithm takes in a vector of random bits denoted by  $\vec{r}$ . This way of representing random algorithms is identical to having the algorithm generate random coins, and we use these two definitions interchangeably. The randomized algorithm aims to maximize the expectation of  $f$ , where the expectation is taken over the random bits of the algorithm.

**Max  $k$ -Cut, Max-DiCut:** In this paper we provide fast distributed approximation algorithms to some fundamental problems, which we now define formally. In the Max  $k$ -Cut problem we wish to divide the vertices into  $k$  disjoint sets, such that the weight of edges between different sets is maximized. In the Max-DiCut problem the edges are directed and we wish to divide the nodes into two disjoint sets, denoted  $A, B$ , such that the weight of edges directed from  $A$  to  $B$  is maximized.

**Max 2-SAT:** In the Max 2-SAT problem we are given a set of unique weighted clauses over some set of variables, where each clause contains *at most* two literals. Our goal is to maximize the weight of satisfied clauses. This problem is more general than the cut problems, so we must define what it means in the distributed context. First, the variables will be node variables as defined before. Second, each node knows all of the clauses it appears in as a literal.

**Correlation clustering:** We are given an edge weighted graph  $G(V, E, w)$ , such that each edge is also assigned a value from  $\{+, -\}$  (referred to positive and negative edges). Given some partition,  $C$ , of the graph into disjoint clusters, we say that an edge *agrees* with  $C$  if it is positive and both endpoints are in the same cluster, or it is negative, and its endpoints are in different clusters. Otherwise we say it *disagrees* with  $C$ . We aim to find a partition  $C$ , using any number of clusters, such that the weight of edges that agree with  $C$  (agreements) is maximized (max-agree), or equivalently the weight of edges that disagree with  $C$  is minimized (min-disagree).

The problem is usually expressed as an LP using edge variables, where each variable indicates whether the nodes are in the same cluster. This allows a solution to use any number of clusters. In this paper we only aim to achieve a  $(1/2 - \epsilon)$ -approximation for the problem. This can be done rather simply without employing the full power of correlation clustering. Specifically, two clusters are enough for our case as we show that we can deterministically achieve  $(1/2 - \epsilon)|E|$  agreements which results in the desired approximation ratio.

**Local utility functions:** We are interested in a type of utility function which we call a *local utility function*. Before we continue with the definition let us define an operator on assignments  $\bar{X}$ , we define  $L_v[\bar{X}] = \{\{X_u = \alpha_u\} \in \bar{X} \mid u \in N(v)\}$ . For convenience, when we pass  $L_v[\bar{X}]$  as parameter to a function, we assume that the function also receives the 1-hop neighborhood of  $v$  which we do not write explicitly. We say that a utility function  $f$ , as defined above, is a local utility function if for every  $v$  there exists a function  $g_v$  s.t  $f(\bar{X} \cup \{X_v = \alpha\}) - f(\bar{X} \cup \{X_v = \alpha'\}) = g_v(L_v[\bar{X}], \alpha, \alpha')$ . That is, to compute the change

in the utility function which is caused by changing  $X_v$  from  $\alpha'$  to  $\alpha$ , we only need to know the immediate neighborhood of  $v$ , and the assignment to neighboring node variables. We note that for the cut problems considered in this paper the utility functions are indeed local utility functions. This is proven in the following Lemma:

► **Lemma 1.** *The utility functions for Max  $k$ -Cut, Max-DiCut and max-agree correlation clustering with 2 clusters are local utility functions.*

**Proof.** The utility functions for Max  $k$ -Cut is given by  $f(\bar{X}) = \sum_{e=(w,u) \in E} w(e) \cdot X_w \oplus X_u$  where  $X_w \oplus X_u = 0$  if  $X_w = X_u$  and 1 otherwise. Thus, if we fix some  $v$  it holds that

$$\begin{aligned} & f(\bar{X} \cup \{X_v = \alpha'\}) - f(\bar{X} \cup \{X_v = \alpha\}) \\ &= \sum_{e=(v,u) \in E} w(e) \cdot \alpha' \oplus X_u - \sum_{e=(v,u) \in E} w(e) \cdot \alpha \oplus X_u \\ &= \sum_{e=(v,u) \in E} w(e) \cdot (\alpha' \oplus X_u - \alpha \oplus X_u) \triangleq g_v(L_v[\bar{X}], \alpha', \alpha) \end{aligned}$$

Because the final sum only depends on vertices  $u \in N(v)$ , the last equality defines the local function equivalent to the difference, and we are done.

For the problem of Max-DiCut the utility functions is given by  $f(\bar{X}) = \sum_{e=(v \rightarrow u) \in E} w(e) \cdot X_v \wedge (1 - X_u)$ , and for max-agree correlation clustering with 2 clusters the utility function is given by  $f(\bar{X}) = \sum_{e=(v,u) \in E^+} w(e) \cdot (1 - X_v \oplus X_u) + \sum_{e=(v,u) \in E^-} w(e) \cdot X_v \oplus X_u$  ( $E^+, E^-$  are the positive and negative edges, respectively), and the proof is exactly the same. ◀

**Submodular functions:** A family of functions that will be of interest in this paper is the family of *submodular functions*. A function  $f : \{0, 1\}^\Omega \rightarrow \mathbb{R}$  is called a set function, with ground set  $\Omega$ . It is said to be submodular if for every  $S, T \subseteq \Omega$  it holds that  $f(S) + f(T) \geq f(S \cup T) + f(S \cap T)$ . The functions we are interested in have  $V$  as their ground set, thus we remain with our original notation, setting  $A = \{0, 1\}$  and having  $f$  take in a set of binary assignments  $\bar{X}$  as a parameter.

**The method of conditional expectations:** Next, we consider the method of conditional expectations. Let  $A$  be some set and  $f : A^n \rightarrow \mathbb{R}$ , next let  $\bar{X} = (X_1, \dots, X_n)$  be a vector of random variables taking values in  $A$ . We wish to be consistent with the previous notation, thus we treat  $\bar{X}$  as a set of assignments. If  $E[f(\bar{X})] \geq \beta$ , then there is an assignment of values  $\bar{Z} = \{X_i = \alpha_i\}_{i=1}^n$  such that  $f(\bar{Z}) \geq \beta$ . We describe how to *find* the vector  $\bar{Z}$ . We first note that from the law of total expectation it holds that  $E[f(\bar{X})] = \sum_{\alpha \in A} E[f(\bar{X}) \mid X_1 = \alpha] Pr[X_1 = \alpha]$ , and therefore for at least some  $\alpha \in A$  it holds that  $E[f(\bar{X}) \mid X_1 = \alpha] \geq \beta$ . We set this value to be  $\alpha_1$ . We then repeat this process for the rest of the values in  $\bar{X}$ , which results in the set  $\bar{Z}$ . In order for this method to work we need it to be possible to *compute*<sup>3</sup> the conditional expectation of  $f(\bar{X})$ .

**Graph coloring:** A  $c$ -coloring for  $G(V, E)$  is defined as a function  $\varphi : V \rightarrow \mathcal{C}$ . For simplicity we treat any set  $\mathcal{C}$  of size  $c$  with some ordering as the set of integers  $[c]$ . This simplifies things as we can always consider  $\varphi(v) \pm 1$ , which is very convenient. We say that a coloring

<sup>3</sup> This point is critical, and this computation is not simple in many cases. In our case we also need this computation to be done *locally* at every nodes. We apply this technique to Max-Cut, which meets all of these demands.

---

**Algorithm 1:**  $OL(G, \vec{r}, \pi)$ .
 

---

- 1  $\forall v \in V, X_v = \text{init}(L_v[\vec{X}])$
  - 2 Order the variables according to  $\pi: v_1, v_2, \dots, v_n$
  - 3 **for**  $i$  from 1 to  $n$  **do**  $X_{v_i} = \text{decide}(L_{v_i}[\vec{X}], r_i)$
  - 4 Return  $\vec{X}$
- 

is a legal coloring if  $\forall v, u$  s.t.  $(v, u) \in E$  it holds that  $\varphi(v) \neq \varphi(u)$ . An important tool in this paper is *defective coloring*. Let us fix some  $c$ -coloring function  $\varphi : V \rightarrow [c]$ . We define the defect of a vertex to be the number of monochromatic edges it has. Formally,  $\text{defect}(v) = \text{size}\{u \in N(v) \mid \varphi(v) = \varphi(u)\}$ . We call  $\varphi$  a  $c$ -coloring with defect  $d$  if it holds that  $\forall v \in V, \text{defect}(v) \leq d$ . A classic result by Kuhn [29] states that for all  $d \in \{1, 2, \dots, \Delta\}$  an  $O(\Delta^2/d^2)$ -coloring with defect  $d$  can be computed deterministically in  $O(\log^* n)$  rounds in the CONGEST model.

In this paper we define a new kind of defective coloring which we call a *weighted  $\epsilon$ -defective coloring*. Given a positively edge weighted graph and any coloring, for every vertex we denote by  $E_m(v)$  its monochromatic edges. Define its weighted defect as  $\text{defect}_w(v) = \sum_{e=(u,v) \in E_m(v)} w(e)$ . We aim to find a coloring s.t the defect for every  $v$  is below  $\epsilon w(v) = \epsilon \sum_{v \in e} w(e)$ . We show that the algorithm of Kuhn actually computes a weighted  $\epsilon$ -defective  $O(\epsilon^{-2})$ -coloring. We state the following theorem (As the analysis is rather similar to the original analysis of Kuhn, the proof is deferred to the full version):

► **Theorem 2.** *For any constant  $\epsilon \in (0, 1/e)$  a weighted  $\epsilon$ -defective  $O(\epsilon^{-2})$ -coloring can be computed deterministically in  $O(\log^* n)$  rounds in the CONGEST model.*

### 3 Orderless-local algorithms

Next we turn our attention to a large family of (potentially randomized) greedy algorithms. We limit ourselves to graph algorithms s.t every node  $v$  has a variable  $X_v$  taking values in some set  $A$ . We aim to maximize some global utility function  $f(\vec{X})$ . We focus on a class of algorithms we call *orderless-local* algorithms. These are greedy-like algorithms which may traverse the vertices in any order, and at each step decide upon a value for  $X_v$ . This decision is local, meaning that it only depends on the 1-hop topology of  $v$  and the values of neighboring variables. The decision may be random, but each variable has its own random bits, keeping the decision process local.

The code for a generic algorithm of this family is given in Algorithm 1. The algorithm first initiates the vertex variables. Next it traverses the variables in some order  $\pi : V \rightarrow [n]$ . Each  $X_{v_i}$  is assigned a value according to some function *decide*, which only depends on  $L_{v_i}[\vec{X}]$  at the time of the assignment and some random bits  $\vec{r}_i$  which are only used to set the value for that variable. Finally the assignment to the variables is returned. We are guaranteed that the expected value of  $f$  is at least  $\beta(G)$  for any, potentially adversarial, ordering  $\pi$  of the variables. Formally,  $E_{\vec{r}}[f(OL(G, \vec{r}, \pi))] \geq \beta(G)$ .

We show that this family of algorithms can be easily distributed using coloring, s.t the running time of the distributed version depends on the number of colors. The distributed version, **OLDist**, is presented as Algorithm 2. The variables are all initiated as in the sequential version, and then the color classes are executed sequentially, while in each color class the nodes execute *decide* simultaneously, and send the newly assigned value to all neighbors. *Decide* does not communicate with the neighbors, so the algorithm finishes in  $O(c)$  rounds.

---

**Algorithm 2:**  $OLDist(G, \vec{r}, \varphi)$ .

---

```

1  $\forall v \in V, X_v = init(L_v[\bar{X}])$ 
2 for  $i$  from 1 to  $c$  do
3   foreach  $v$  s.t.  $\varphi(v) = i$  simultaneously do
4      $X_{v_i} = decide(L_v[\bar{X}], r_i)$ 
5     Send  $X_{v_i}$  to neighbors
6   end
7 end
8 return  $\bar{X}$ 

```

---

It is easy to see that given the same randomness both the sequential and distributed algorithms output the same result, this is because all decisions of the distributed algorithm only depend on the 1-hop environment of a vertex, and we are provided with a *legal* coloring. Thus, one round of the distributed algorithm is equivalent to many steps of the sequential algorithm. We prove the following lemma:

► **Lemma 3.** *For any graph  $G$  with a legal coloring  $\varphi$ , there exists an order  $\pi$  on the variables s.t it holds that  $OL(G, \vec{r}, \pi) = OLDist(G, \vec{r}, \pi)$  for any  $\vec{r}$ .*

**Proof.** We prove the claim by induction on the executions of color classes by the distributed algorithm. We note that the execution of the distributed algorithm defines an order on the variables. Let us consider the  $i$ -th color class. Let us denote these variables as  $\{X_{v_j}\}_{j=1}^k$ , assigning some arbitrary order within the class. The ordering we analyze for the sequential algorithm would be  $\pi(v_j) = (\varphi(v), j)$ . Now both the distributed and sequential algorithms follow the same order of color classes, thus we allow ourselves to talk about the sequential algorithm finishing an execution of a color class.

Let  $Y_i$  be the assignments to all variables of the distributed algorithm after the  $i$ -th color class finishes execution. And let  $Y'_i$  be the assignments made by the sequential algorithm following  $\pi$  until all variable in the  $i$ -th color class are assigned. Both algorithms initiate the variables identically, so it holds that  $Y_0 = Y'_0$ . Assume that it holds that  $Y_{i-1} = Y'_{i-1}$ . The coloring is legal, so for any  $X_u, X_v$ , s.t  $\varphi(u) = \varphi(v) = i$  it holds that  $N(v) \cap u = \emptyset$ . Thus, when assigning  $v$ , its neighborhood is not affected by any other assignments done in the color class, so the randomness is identical for both algorithms, and using the induction hypothesis all assignments up until this color class were identical. Thus, for all variables in this color class *decide* will be executed with the same parameters for both the distributed and sequential algorithms, and all assignments will be identical. ◀

Finally we show that for any graph  $G$  with a legal coloring  $\varphi$ , it holds that

$$E_{\vec{r}}[f(OLDist(G, \vec{r}, \varphi))] \geq \beta(G).$$

We know from Lemma 3 that for any coloring  $\varphi$  there exists an ordering  $\pi$  s.t  $OL(G, \vec{r}, \pi) = OLDist(G, \vec{r}, \varphi)$  for any  $\vec{r}$ . The proof is direct from here:

$$\begin{aligned}
E_{\vec{r}}[f(OLDist(G, \vec{r}, \varphi))] &= \sum_{\vec{r}} Pr[\vec{r}] f(OLDist(G, \vec{r}, \varphi)) \\
&= \sum_{\vec{r}} Pr[\vec{r}] f(OL(G, \vec{r}, \pi)) = E_{\vec{r}}[f(OL(G, \vec{r}, \pi))] \geq \beta(G)
\end{aligned}$$

---

**Algorithm 3:** CondExpSeq( $G$ ).
 

---

- 1  $\forall v \in V, X_v = \emptyset$
  - 2 Order the variables according to any order:  $v_1, v_2, \dots, v_n$
  - 3 **for**  $i$  from 1 to  $n$  **do**  $X_{v_i} = \operatorname{argmax}_{\alpha} E[f(\bar{X}) \mid Y, X_{v_i} = \alpha] - E[f(\bar{X}) \mid Y]$
  - 4 Return  $\bar{X}$
- 

We conclude that any orderless-local algorithm can be distributed, achieving the same performance guarantee on  $f$ , and requiring  $O(c)$  communication rounds to finish, given a legal  $c$ -coloring. We state the following theorem:

► **Theorem 4.** *Given some utility function  $f$ , any sequential orderless-local algorithm for which it holds that  $E_{\vec{r}}[f(OL(G, \vec{r}, \pi))] \geq \beta(G)$ , can be converted into a distributed algorithm for which it holds that  $E_{\vec{r}}[f(OLDist(G, \vec{r}, \varphi))] \geq \beta(G)$ , where  $\varphi$  is a legal  $c$ -coloring of the graph. The running time of the distributed algorithm is  $O(c)$  communication rounds.*

### 3.1 Distributed derandomization

We consider the method of conditional expectations in the distributed case for some local utility function  $f(G, \bar{X})$ , as defined in the preliminaries. Assume that the value of every  $X_v$  is set independently at random according to some distribution on  $A$  which depends only on the 1-hop neighborhood of  $v$ . We are guaranteed that  $\forall G, E[f(G, \bar{X})] \geq \beta(G)$ . Thus in the sequential setting we may use the method of conditional expectations to compute a deterministic assignment to the variables with the same guarantee. We show that because  $f$  is a local utility function, the method of conditional expectations applied on  $f$  is an orderless-local algorithm, and thus can be distributed.

Initially all variables are initiated to some value  $\emptyset \notin A$ , meaning the variable is unassigned. Let  $Y = \{X_u = \alpha_u \mid u \in U \subseteq V\}$  be some partial assignment to the variables. The method of conditional expectations goes over the variables in any order, and in each iteration sets  $X_{v_i} = \operatorname{argmax}_{\alpha} E[f(\bar{X}) \mid Y, X_{v_i} = \alpha]$ . This is equivalent to  $\operatorname{argmax}_{\alpha} \{E[f(\bar{X}) \mid Y, X_{v_i} = \alpha] - E[f(\bar{X}) \mid Y]\}$ , as the subtracted term is just a constant. With this in mind, we present the pseudo code for the method of conditional expectations in Algorithm 3.

To show that Algorithm 3 is an orderless-local algorithm we only need to show that  $\operatorname{argmax}_{\alpha} E[f(\bar{X}) \mid Y, X_v = \alpha] - E[f(\bar{X}) \mid Y]$  can be computed locally for any  $v$ . We state the following lemma, followed by the main theorem for this section.

► **Lemma 5.** *The value  $\operatorname{argmax}_{\alpha} E[f(\bar{X}) \mid Y, X_v = \alpha] - E[f(\bar{X}) \mid Y]$  can be computed locally.*

**Proof.** It holds that:

$$\begin{aligned}
 & E[f(\bar{X}) \mid Y, X_v = \alpha_v] - E[f(\bar{X}) \mid Y] \\
 &= \sum_{\alpha \in A} E[f(\bar{X}) \mid Y, X_v = \alpha] Pr[X_v = \alpha] - \sum_{\alpha \in A} E[f(\bar{X}) \mid Y, X_v = \alpha] Pr[X_v = \alpha] \\
 &= \sum_{\alpha \in A} Pr[X_v = \alpha] (E[f(\bar{X}) \mid Y, X_v = \alpha_v] - E[f(\bar{X}) \mid Y, X_v = \alpha])
 \end{aligned}$$

Where the first equality is due to the law of total expectation and the fact that  $\sum_{\alpha \in A} Pr[X_v = \alpha] = 1$ . The probability of assigning  $X_v$  to some value can be computed locally, so we are only

left with the difference between the expectations. To show that this is indeed a local quantity we use the definition of expectation as a weighted summation over all possible assignments to unassigned variables. Let  $U_v$  be the set of all possible assignments to unassigned variables in  $N(v)$  and let  $U$  be the set of all possible assignments to the rest of the unassigned variables. It holds that:

$$\begin{aligned}
 & E[f(\bar{X}) \mid Y, X_v = \alpha_v] - E[f(\bar{X}) \mid Y, X_v = \alpha] \\
 &= \sum_{Z_v \in U_v} \sum_{Z \in U} Pr[Z_v] Pr[Z] f(\bar{X} \cup Z_v \cup Z \cup \{X_v = \alpha_v\}) - f(\bar{X} \cup Z_v \cup Z \cup \{X_v = \alpha\}) \\
 &= \sum_{Z_v \in U_v} \sum_{Z \in U} Pr[Z_v] Pr[Z] g_v(L_v[\bar{X} \cup Z_v \cup Z], \alpha, \alpha_v) \\
 &= \sum_{Z_v \in U_v} \sum_{Z \in U} Pr[Z_v] Pr[Z] g_v(L_v[\bar{X} \cup Z_v], \alpha, \alpha_v) \\
 &= \sum_{Z_v \in U_v} Pr[Z_v] g_v(L_v[\bar{X} \cup Z_v], \alpha, \alpha_v),
 \end{aligned}$$

where in the first equality we use the definition of expectations and the fact that the variables are set independently of each other. Then we use the definition of a local utility function, and finally the dependence on  $U$  disappears due to the law of total probability. The final sum can be computed locally, as the probabilities for assigning variables in  $Z_v$  are known and  $g_v$  is local. ◀

► **Theorem 6.** *Let  $G$  be any graph and  $f$  a local utility function for which it holds that  $E[f(\bar{X})] \geq \beta$ , where the random assignments to the variables are independent of each other, and depend only on the immediate neighborhood of the node. There exists a distributed algorithm achieving the same value as the expected value for  $f$ , running in  $O(c)$  communication rounds in the CONGEST model, given a legal  $c$ -coloring.*

## 3.2 Submodular Maximization

In this section we consider the problem of unconstrained submodular function maximization. Given an submodular function  $f$  (as defined in Section 2), we aim to find an input s.t the function is maximized. There are no constraints on the input set we pass to the function, hence it is 'unconstrained'. We are interested in finding an approximate solution to the problem, to this end, we consider both the deterministic and randomized algorithms of [10], achieving  $1/3$  and  $1/2$  approximation ratios for unconstrained submodular maximization. We show that both can be expressed as orderless-local algorithms for any local utility function. As the deterministic and randomized algorithms of [10] are almost identical, we focus on the randomized algorithm achieving a  $1/2$ -approximation in expectation (Algorithm 5), as it is a bit more involved (The deterministic algorithm appears as Algorithm 4). The algorithms of [10] are defined for any submodular function, but as we are interested only in the case where the ground set is  $V$ , we will present it as such.

The algorithm maintains two variable assignment  $Z_i, Y_i$ , initially  $Z_0 = \{X_v = 0 \mid v \in V\}$ ,  $Y_0 = \{X_v = 1 \mid v \in V\}$ . It iterates over the variables in any order, at each iteration it considers two nonnegative quantities  $a_i, b_i$ . These quantities represent the gain of either setting  $X_{v_i} = 1$  in  $Z_{i-1}$  or setting  $X_{v_i} = 0$  in  $Y_{i-1}$ . Next a coin is flipped with probability  $p = a_i / (a_i + b_i)$ , if  $a_i = b_i = 0$  we set  $p = 1$ . If we get heads we set  $X_{v_i} = 1$  in  $Z_i$  and otherwise we set it to 0 in  $Y_i$ . When the algorithm ends it holds that  $Z_n = Y_n$ , and this is our solution. The deterministic algorithm is almost identical, only that it allows  $a_i, b_i$  to take

**Algorithm 4:**  $\text{det-usm}(f)$ .

---

```

1  $Z_0 = \{X_v = 0 \mid v \in V\}, Y_0 = \{X_v = 1 \mid v \in V\}$ 
2 for  $i$  from 1 to  $n$  do
3    $a_i = f(Z_{i-1} \cup \{X_{v_i} = 1\}) - f(Z_{i-1})$ 
4    $b_i = f(Y_{i-1} \cup \{X_{v_i} = 0\}) - f(Y_{i-1})$ 
5   if  $a_i \geq b_i$  then
6      $Z_i = Z_{i-1} \cup \{X_{v_i} = 1\}$ 
7      $Y_i = Y_{i-1}$ 
8   end
9   else
10     $Z_i = Z_{i-1}$ 
11     $Y_i = Y_{i-1} \cup \{X_{v_i} = 0\}$ 
12  end
13 end
14 return  $Z_n$ 

```

---

**Algorithm 5:**  $\text{rand-usm}(f)$ .

---

```

1  $Z_0 = \{X_v = 0 \mid v \in V\}, Y_0 = \{X_v = 1 \mid v \in V\}$ 
2 Order the variables in any order  $v_1, \dots, v_n$ 
3 for  $i$  from 1 to  $n$  do
4    $a_i = \max \{f(Z_{i-1} \cup \{X_{v_i} = 1\}) - f(Z_{i-1}), 0\}$ 
5    $b_i = \max \{f(Y_{i-1} \cup \{X_{v_i} = 0\}) - f(Y_{i-1}), 0\}$ 
6   if  $a_i + b_i = 0$  then  $p = 1$  else  $p = a_i / (a_i + b_i)$ 
7    $Y_i = Y_{i-1}, Z_i = Z_{i-1}$ 
8   Flip a coin with probability  $p$ , if heads  $Z_i = Z_{i-1} \cup \{X_{v_i} = 1\}$ , else
    $Y_i = Y_{i-1} \cup \{X_{v_i} = 0\}$ 
9 end
10 return  $Z_n$ 

```

---

negative values, and instead of flipping a coin it makes the decision greedily by comparing  $a_i, b_i$ .

We first note that the algorithm does not directly fit into our mold, as each vertex has two variables. We can overcome this, by taking  $X_v$  to be a binary tuple, the first coordinate stores its value for  $Z_i$ , and the other for  $Y_i$ . Initially it holds that  $\forall v \in V, X_v = (0, 1)$ , and our final goal function will only take the first coordinate of the variable. We note that because  $f$  is a local utility function the values  $a_i, b_i$  can be computed locally, this results directly from the definition of a local utility function, as we are interested in the change in  $f$  caused by flipping a single variable. Now we may rewrite the algorithm as an orderless-local algorithm, the pseudocode as Algorithm 6.

Using Theorem 4 we state our main result:

► **Theorem 7.** *For any graph  $G$  and a local unconstrained submodular function  $f$  with  $V$  as its ground set, there exists a randomized distributed  $1/2$ -approximation, and a deterministic  $1/3$ -approximation algorithms running in  $O(c)$  communication rounds in the CONGEST model, given a legal  $c$ -coloring.*



---

**Algorithm 6:**  $\text{rand-usm}(G, \vec{r}, \pi)$ .

---

```

1  $\forall v \in V, X_v = (0, 1)$ 
2 Order the vertices according to  $\pi$ 
3 for  $i$  from 1 to  $n$  do
4    $X_u = \text{decide}(L_v[\vec{X}], \vec{r}_i)$ 
5 end
6 return  $X_n$ 

```

---



---

**Algorithm 7:**  $\text{decide}(L_v[\vec{X}], r)$ .

---

```

1  $Z = \{X_u = \alpha_{u,1} \mid \{X_u = (\alpha_{u,1}, \alpha_{u,2})\} \in L_v[\vec{X}]\}$ 
2  $Y = \{X_u = \alpha_{u,2} \mid \{X_u = (\alpha_{u,1}, \alpha_{u,2})\} \in L_v[\vec{X}]\}$ 
3  $a = \max\{g_v(Z, 0, 1), 0\}$ 
4  $b = \max\{g_v(Z, 1, 0), 0\}$ 
5 if  $a + b = 0$  then  $p = 1$ 
6 else  $p = a/(a + b)$ 
7 Flip coin with probability  $p$ 
8 if heads then return (1,1)
9 else return (0,0)

```

---

### 3.3 Fast approximations for cut functions

Using the results of the previous sections we can provide fast and simple approximation algorithms for Max-DiCut and Max  $k$ -Cut. Lemma 1 guarantees that the utility functions for these problems are indeed local utility functions. For Max-DiCut we use the algorithms of Buchbinder *et al.*, as this is an unconstrained submodular function. For Max  $k$ -Cut each node choosing a side uniformly at random achieves a  $(1 - 1/k)$  approximation, thus we use the results of Section 3.1. Theorem 7 and Theorem 6 immediately guarantee distributed algorithms, running in  $O(c)$  communication rounds given a legal  $c$ -coloring.

Denote by  $\text{Cut}(G, \varphi)$  one of the cut algorithms guaranteed by Theorem 7 or Theorem 6. We present two algorithms, **approxCutDet**, a deterministic algorithm to be used when  $\text{Cut}(G, \varphi)$  is deterministic (Algorithm 8), and, **approxCutRand**, a randomized algorithm (Algorithm 9) for the case when  $\text{Cut}(G, \varphi)$  is randomized. **approxCutDet** works by coloring the graph  $G$  using a weighted  $\epsilon$ -defective coloring and then defining a new graph  $G'$  by dropping all of the monochromatic edges. This means that the coloring is a legal coloring for  $G'$ . Finally we call one of the deterministic cut functions. **approxCutRand** is identical, apart from the fact that nodes choose a color uniformly at random from  $[\epsilon^{-1}]$ .

For **approxCutDet**, the running time of the coloring is  $O(\log^* n)$  rounds, returning a weighted  $\epsilon$ -defective  $O(\epsilon^{-2})$ -coloring. The running time of the cut algorithms is the number of colors, thus the total running time of the algorithm is  $O(\epsilon^{-2} + \log^* n)$  rounds. Using the same reasoning, the running time of **approxCutRand** is  $O(\epsilon^{-1})$ . It is only left to prove the approximation ratio. We prove the following lemma:

► **Lemma 8.** *Let  $G(V, E, w)$  be any graph, and let  $G'(V, E', w)$  be a graph resulting from removing any subset of edges from  $G$  of total weight at most  $\epsilon \sum_{e \in E} w(e)$ . Then for any constant  $p$ , any  $p$ -approximation for Max-DiCut or Max  $k$ -Cut for  $G'$  is a  $p(1 - 4\epsilon)$ -approximation for  $G$ .*

---

**Algorithm 8:** `approxCutDet`( $G, \epsilon$ ).

---

- 1  $\varphi = \text{epsilonColor}(G, \epsilon)$
  - 2 Let  $G' = (V, E' = \{(v, u) \in E \mid \varphi(v) \neq \varphi(u)\})$
  - 3 `Cut`( $G', \varphi$ )
- 

---

**Algorithm 9:** `approxCutRand`( $G, \epsilon$ ).

---

- 1 Each vertex  $v$  chooses  $\varphi(v)$  uniformly at random from  $[\lceil \epsilon^{-1} \rceil]$
  - 2 Let  $G' = (V, E' = \{(v, u) \in E \mid \varphi(v) \neq \varphi(u)\})$
  - 3 `Cut`( $G', \varphi$ )
- 

**Proof.** Let  $OPT, OPT'$  be the size of optimal solutions for  $G, G'$ . It holds that  $OPT' \geq OPT - \epsilon \sum_{e \in E} w(e)$ , as any solution for  $G$  is also a solution for  $G'$  whose value differs by at most  $\epsilon \sum_{e \in E} w(e)$  (the weight of discarded edges). Assigning every node a cut side uniformly at random the expected cut weight is at least  $\sum_{e \in E} w(e)/4$  for Max-DiCut and Max  $k$ -Cut. Using the probabilistic method this implies that  $OPT \geq \sum_{e \in E} w(e)/4$ . Using all of the above we can say that given a  $p$ -approximate solution for  $OPT'$  it holds that:  $p \cdot OPT' \geq p(OPT - \epsilon \sum_{e \in E} w(e)) \geq p(OPT - 4\epsilon OPT) = p(1 - 4\epsilon)OPT$   $\blacktriangleleft$

Lemma 8 immediately guarantees the approximation ratio for the deterministic algorithm. As for the randomized algorithm, let the random variable  $\delta$  be the fraction of edges removed, let  $p$  be the approximation ratio guaranteed by one of the cut algorithms and let  $\rho$  be the approximation ratio achieved by `approxCutRand`. We know that  $E_\rho[\rho \mid \delta] = p(1 - 4\delta)$ . Applying the law of total expectations we get that  $E[\rho] = E_\delta[E_\rho[\rho \mid \delta]] = E_\delta[p(1 - 4\delta)] = p(1 - 4\epsilon)$ . We state our main theorems for this section.

► **Theorem 9.** *There exists a deterministic  $(1 - 1/k - \epsilon)$ -approximation algorithms for Weighted Max  $k$ -Cut running in  $O(\log^* n)$  communication rounds in the CONGEST model.*

► **Theorem 10.** *There exists a deterministic  $(1/3 - \epsilon)$ -approximation algorithm for Weighted Max-DiCut running in  $O(\log^* n)$  communication rounds in the CONGEST model.*

► **Theorem 11.** *There exists a randomized distributed expected  $(1/2 - \epsilon)$ -approximation for Weighted Max-DiCut running in  $O(\epsilon^{-1})$  communication rounds in the CONGEST model.*

### Correlation clustering

We note the same techniques used for Max-Cut work directly for max-agree correlation clustering on general graphs. Specifically, if we divide the nodes into two clusters, s.t each node selects a cluster uniformly at random, each edge has exactly probability  $1/2$  to agree with the clustering, thus the expected value of the clustering is  $\sum_{e \in E} w(e)/2$ , which is a  $1/2$ -approximation. The above can be derandomized exactly in the same manner as Max-Cut, meaning this is an orderless local algorithm. Finally, we apply the weighted  $\epsilon$ -defective coloring algorithm twice (note that we ignore the sign of the edge), discard all monochromatic edges and execute the deterministic algorithm guaranteed from Theorem 6 with a legal coloring. Because there must exist a clustering which has a value at least  $\sum_{e \in E} w(e)/2$ , a lemma identical to Lemma 8 can be proved and hence we are done. We state the following theorem:

► **Theorem 12.** *There exists a deterministic  $(1/2 - \epsilon)$ -approximation algorithms for weighted max-agree correlation clustering on general graphs, running in  $O(\log^* n)$  communication rounds in the CONGEST model.*

---

### References

- 1 Kook Jin Ahn, Graham Cormode, Sudipto Guha, Andrew McGregor, and Anthony Wirth. Correlation clustering in data streams. In *ICML*, volume 37 of *JMLR Workshop and Conference Proceedings*, pages 2237–2246. JMLR.org, 2015.
- 2 Nir Ailon, Moses Charikar, and Alantha Newman. Aggregating inconsistent information: Ranking and clustering. *J. ACM*, 55(5):23:1–23:27, 2008.
- 3 Per Austrin. Balanced max 2-sat might not be the hardest. In *STOC*, pages 189–197. ACM, 2007.
- 4 Nikhil Bansal, Avrim Blum, and Shuchi Chawla. Correlation clustering. In *FOCS*, page 238. IEEE Computer Society, 2002.
- 5 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.
- 6 Reuven Bar-Yehuda, Keren Censor-Hillel, and Gregory Schwartzman. A distributed  $(2 + \epsilon)$ -approximation for vertex cover in  $o(\log \Delta / \epsilon \log \log \Delta)$  rounds. *J. ACM*, 64(3):23:1–23:11, 2017.
- 7 Surender Baswana and Sandeep Sen. A simple and linear time randomized algorithm for computing sparse spanners in weighted graphs. *Random Struct. Algorithms*, 30(4):532–563, 2007.
- 8 Ran Ben-Basat, Ken-ichi Kawarabayashi, and Gregory Schwartzman. Parameterized distributed algorithms. *CoRR*, abs/1807.04900, 2018. [arXiv:1807.04900](https://arxiv.org/abs/1807.04900).
- 9 Allan Borodin, Morten N. Nielsen, and Charles Rackoff. (incremental) priority algorithms. In *SODA*, pages 752–761. ACM/SIAM, 2002.
- 10 Niv Buchbinder, Moran Feldman, Joseph Naor, and Roy Schwartz. A tight linear time  $(1/2)$ -approximation for unconstrained submodular maximization. *SIAM J. Comput.*, 44(5):1384–1402, 2015. [doi:10.1137/130929205](https://doi.org/10.1137/130929205).
- 11 Keren Censor-Hillel, Elad Haramaty, and Zohar S. Karnin. Optimal dynamic distributed MIS. In *PODC*, pages 217–226. ACM, 2016.
- 12 Keren Censor-Hillel, Rina Levy, and Hadas Shachnai. Fast distributed approximation for max-cut. In *Algorithms for Sensor Systems, 13th International Symposium on Algorithms and Experiments for Wireless Sensor Networks, ALGOSENSORS 2017, Vienna, Austria, September 7–8, 2017, Revised Selected Papers*, volume 10718 of *Lecture Notes in Computer Science*, pages 41–56. Springer, 2017.
- 13 Moses Charikar, Venkatesan Guruswami, and Anthony Wirth. Clustering with qualitative information. *J. Comput. Syst. Sci.*, 71(3):360–383, 2005.
- 14 Shuchi Chawla, Konstantin Makarychev, Tselil Schramm, and Grigory Yaroslavtsev. Near optimal LP rounding algorithm for correlationclustering on complete and complete k-partite graphs. In *STOC*, pages 219–228. ACM, 2015.
- 15 Erik D. Demaine, Dotan Emanuel, Amos Fiat, and Nicole Immorlica. Correlation clustering in general weighted graphs. *Theor. Comput. Sci.*, 361(2-3):172–187, 2006.
- 16 Uriel Feige and Michel X. Goemans. Aproximating the value of two prover proof systems, with applications to MAX 2sat and MAX DICUT. In *ISTCS*, pages 182–189. IEEE Computer Society, 1995.
- 17 Manuela Fischer, Mohsen Ghaffari, and Fabian Kuhn. Deterministic distributed edge-coloring via hypergraph maximal matching. In *FOCS*, pages 180–191. IEEE Computer Society, 2017.

- 18 Robert G. Gallager, Pierre A. Humblet, and Philip M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. Program. Lang. Syst.*, 5(1):66–77, 1983.
- 19 M. R. Garey, David S. Johnson, and Larry J. Stockmeyer. Some simplified np-complete graph problems. *Theor. Comput. Sci.*, 1(3):237–267, 1976. doi:10.1016/0304-3975(76)90059-1.
- 20 Mohsen Ghaffari, David G. Harris, and Fabian Kuhn. On derandomizing local distributed algorithms. *CoRR*, abs/1711.02194, 2017.
- 21 Mohsen Ghaffari, Fabian Kuhn, and Yannic Maus. On the complexity of local distributed graph problems. In *STOC*, pages 784–797. ACM, 2017.
- 22 Ioannis Giotis and Venkatesan Guruswami. Correlation clustering with a fixed number of clusters. *Theory of Computing*, 2(13):249–266, 2006.
- 23 Michel X. Goemans and David P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *J. ACM*, 42(6):1115–1145, 1995. doi:10.1145/227683.227684.
- 24 Johan Håstad. Some optimal inapproximability results. *J. ACM*, 48(4):798–859, 2001.
- 25 Juho Hirvonen, Joel Rybicki, Stefan Schmid, and Jukka Suomela. Large cuts with local algorithms on triangle-free graphs. *CoRR*, abs/1402.2543, 2014.
- 26 Satyen Kale and C. Seshadhri. Combinatorial approximation algorithms for maxcut using random walks. In *ICS*, pages 367–388. Tsinghua University Press, 2011.
- 27 Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972.
- 28 Subhash Khot, Guy Kindler, Elchanan Mossel, and Ryan O’Donnell. Optimal inapproximability results for MAX-CUT and other 2-variable csps? *SIAM J. Comput.*, 37(1):319–357, 2007.
- 29 Fabian Kuhn. Weak graph colorings: distributed algorithms and applications. In *SPAA*, pages 138–144. ACM, 2009.
- 30 Reut Levi and Moti Medina. A (centralized) local guide. *Bulletin of EATCS*, 2(122), 2017.
- 31 Michael Lewin, Dror Livnat, and Uri Zwick. Improved rounding techniques for the MAX 2-sat and MAX DI-CUT problems. In *IPCO*, volume 2337 of *Lecture Notes in Computer Science*, pages 67–82. Springer, 2002.
- 32 Nathan Linial. Locality in distributed graph algorithms. *SIAM J. Comput.*, 21(1):193–201, 1992.
- 33 Shiro Matuura and Tomomi Matsui. 0.863-approximation algorithm for MAX DICUT. In *RANDOM-APPROX*, volume 2129 of *Lecture Notes in Computer Science*, pages 138–146. Springer, 2001.
- 34 Matthias Poloczek, Georg Schnitger, David P. Williamson, and Anke van Zuylen. Greedy algorithms for the maximum satisfiability problem: Simple algorithms and inapproximability bounds. *SIAM J. Comput.*, 46(3):1029–1061, 2017.
- 35 Ronitt Rubinfeld, Gil Tamir, Shai Vardi, and Ning Xie. Fast local computation algorithms. In *ICS*, pages 223–238. Tsinghua University Press, 2011.
- 36 Chaitanya Swamy. Correlation clustering: maximizing agreements via semidefinite programming. In *SODA*, pages 526–527. SIAM, 2004.
- 37 Luca Trevisan. Max cut and the smallest eigenvalue. *SIAM J. Comput.*, 41(6):1769–1786, 2012.
- 38 Luca Trevisan, Gregory B. Sorkin, Madhu Sudan, and David P. Williamson. Gadgets, approximation, and linear programming. *SIAM J. Comput.*, 29(6):2074–2097, 2000.




# Strong Separations Between Broadcast and Authenticated Channels

**Julian Loss**

Ruhr University Bochum, Germany

julian.loss@rub.de

 <https://orcid.org/0000-0002-7979-3810>

**Ueli Maurer**


ETH Zurich, Switzerland

maurer@inf.ethz.ch

**Daniel Tschudi<sup>1</sup>**

Aarhus University, Denmark

tschudi@cs.au.dk

 <https://orcid.org/0000-0001-6188-1049>

---

## Abstract

In the theory of distributed systems and cryptography one considers a setting with  $n$  parties, (often) connected via authenticated bilateral channels, who want to achieve a certain goal even if some fraction of the parties is dishonest. A classical goal of this type is to construct a broadcast channel. A broadcast channel guarantees that all honest recipients get the same value  $v$  (consistency) and, if the sender is honest, that  $v$  is the sender's input (validity). Lamport et al. showed that it is possible to construct broadcast if and only if the fraction of cheaters is less than a third.

A natural question, first raised by Lamport, is whether there are weaker, still useful primitives achievable from authenticated channels. He proposed weak broadcast, where the validity condition must hold only if all parties are honest, and showed that it can be achieved with an unbounded number of protocol rounds, while broadcast cannot, suggesting that weak broadcast is in a certain sense weaker than broadcast.

The purpose of this paper is to deepen the investigation of the separation between broadcast and authenticated channels. This is achieved by proving the following results. First, we prove a stronger impossibility result for 3-party broadcast. Even if two of the parties can broadcast, one can not achieve broadcast for the third party. Second, we prove a strong separation between authenticated channels and broadcast by exhibiting a new primitive, called XOR-cast, which satisfies two conditions: (1) XOR-cast is strongly unachievable (even with small error probability) from authenticated channels (which is not true for weak broadcast), and (2) broadcast is strongly unachievable from XOR-cast (and authenticated channels). This demonstrates that the hierarchy of primitives has a more complex structure than previously known. Third, we prove a strong separation between weak broadcast and broadcast which is not implied by Lamport's results. The proofs of these results requires the generalization of known techniques for impossibility proofs.

**2012 ACM Subject Classification** Theory of computation → Cryptographic protocols

**Keywords and phrases** cryptography, multi-party computation, broadcast, impossibility

**Digital Object Identifier** 10.4230/LIPIcs.DISC.2018.36

---

<sup>1</sup> Author was supported by advanced ERC grant MPCPRO.



© Julian Loss, Ueli Maurer, and Daniel Tschudi;

licensed under Creative Commons License CC-BY

32nd International Symposium on Distributed Computing (DISC 2018).

Editors: Ulrich Schmid and Josef Widder; Article No. 36; pp. 36:1–36:17

Leibniz International Proceedings in Informatics



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

## 1 Introduction

### 1.1 Broadcast and Weaker Consistency Guarantees

In the theory of distributed systems and in cryptography one often considers a set of  $n$  parties which must securely perform a certain computation, even if some of the parties are dishonest. Broadcast, one of the most fundamental and widely used such primitives, allows one (possibly cheating) party to distribute a value  $m$  consistently to the other parties, in a context where only bilateral (authenticated) channels between parties are available. More formally, a broadcast protocol allows a sender to distribute a value  $v_s$  such that: *Consistency*: Every honest party outputs the same value  $v$ . *Validity*: If the sender is honest, the honest parties output the sender's value  $v = v_s$ . The seminal result of [12] and [10] states that given authenticated channels, broadcast can be achieved if and only if strictly less than  $\frac{n}{3}$  of the involved parties behave dishonestly, even if an error probability of less than  $\frac{1}{3}$  were tolerated.

In this work, consistency guarantees of a primitive, e.g. a broadcast channel, to which (potentially) each party has an input and receives an output, are modelled in a very general and natural manner, using so-called consistency specifications [14]. It captures, for every set  $H$  of (assumed) honest parties and for every tuple of input values of these honest parties, which tuples of output values are possible, no matter what the other parties do. In other words, a specification guarantees that no adversarial behavior can result in the honest parties' output values to be outside the specified set of tuples. Note that while this concept captures consistency guarantees in the most general form, it does (intentionally) not capture secrecy guarantees.

Broadcast guarantees a very strong form of consistency. The study of primitives with a weaker form of consistency guarantee is well-motivated for two different reasons described below.

First, as argued by Lamport in [11], there are settings of practical relevance where a weaker form of broadcast is sufficient. Specifically, in the *transaction commit problem*, a database transaction is coordinated by some (not necessarily honest) party  $P_1$  who decides whether a transaction should be committed or aborted. A single dishonest party  $P_i$  may be enough to cause the transaction to be aborted, but in this case, the honest parties must agree on whether to abort the transaction, or to commit to it. To formalize this setting, [11] introduced a weaker form of broadcast, which we will henceforth refer to as a *weak broadcast channel*. This channel behaves like a regular broadcast channel if all parties are honest, but requires the validity condition to hold *only if every party is honest*. Such a guarantee may be achievable even if broadcast is not achievable.

Second, such a weaker primitive  $P$  might be assumed to be available, and one can ask whether a stronger primitive (e.g. a broadcast channel) can be achieved by a protocol that not only can use authenticated channels, but also has access to  $P$ . A result of this type, proved in [4], is that broadcast is achievable up to  $\frac{n}{2}$  cheaters, assuming that each party can broadcast to any two other parties.

The ultimate goal of a theory in this field is a characterization of various levels of consistency guarantees as well as the hierarchy between them.

### 1.2 Contribution and Outline

In this work, we are concerned with refining the hierarchy between different types of consistency guarantees and placing weak broadcast in such a hierarchy. As is common for impossibility results in distributed computing, we first prove all of our results in the setting of three parties and then generalize them to the  $n$ -party setting.



In order to strengthen the known impossibility result of [12] one can investigate whether it still holds, even if certain primitives are available to the parties, in addition to bilateral authenticated communication. We prove (see Section 4.1) that even if two of the three parties can broadcast values, there is no protocol that would allow the third party to broadcast a value. The proof of this result requires the generalization of known techniques for impossibility proofs to a setting where additional primitives are given. This contribution, which is used throughout the paper, is of independent interest beyond the specific results of this paper.

In order to investigate the hierarchy of consistency primitives between authenticated channels and broadcast, we propose an intermediate level which we call XOR-cast (see Section 4.2). This channel takes a bit  $b_1$  from  $P_1$  and a bit  $b_2$  from  $P_2$  as input. If all parties behave correctly, the value of  $b_1 \oplus b_2$  should be output by all parties. If one of the parties  $P_1$  or  $P_2$  is dishonest, the honest parties must output the same value. If  $P_3$  is dishonest, the remaining parties must output  $b_1 \oplus b_2$ .

We demonstrate a strong separation between authenticated channels and broadcast by proving two strong impossibility results, where we call an impossibility *strong* if it holds even if a constant error probability is tolerated and even if an arbitrary number of communication rounds are allowed. First, it is strongly impossible to achieve XOR-cast from authenticated communication. Second, it is strongly impossible to achieve broadcast from XOR-cast and authenticated communication. This demonstrates that the hierarchy of primitives has a more complex structure than previously known.

The outline of our paper is as follows. In Section 2.1, we introduce the notion of consistency specifications and protocols. We also give some motivating examples of consistency specifications that will be used throughout our work. Here, we extend the work of [14] to case of probabilistic protocols.

In Section 3, we introduce the impossibility proof technique used in this work. In Section 4, we prove our main results, as explained above. Finally, in Section 5 we show how to generalize the results to the  $n$ -party case.

### 1.3 Related Work

Results on the possibility and impossibility of achieving broadcast when other primitives (stronger than authenticated communication) are available were proved in [4, 1, 6, 16, 13]. In a related line of work, [9, 15] derive combinatorial lower bounds on the number of partial broadcast channels among a set of parties needed in order to still be able to achieve broadcast. The general problem of constructing consistency primitives from assumed such primitives was proposed and formalized in [14].

In [11, 2] it is shown that there exists no perfectly secure protocol which constructs weak broadcast from authenticated channels in a finite number of rounds if  $\frac{n}{3}$  or more of the parties behave dishonestly. On the other hand, Lamport provides a protocol which achieves weak broadcast, but requires an infinite amount of runtime. This suggests that weak broadcast is in some sense weaker than broadcast; namely, the result in [12] implies that there exists no such approximation protocol for broadcast. However, in distributed computing or MPC one is mostly interested in protocols which run for a fixed number of rounds (or at least terminate eventually). Here, Lamport's results show that both weak broadcast and broadcast cannot be achieved with zero error probability given authenticated channels. If one allows protocols with an error probability negligible in the number of rounds, the impossibility for broadcast still holds. On the other hand, it was shown in [3] that weak broadcast can be achieved from authenticated channels with arbitrary small error probability. Moreover, [12, 11, 3] do not consider the relation between weak broadcast and broadcast. Especially, it is not shown whether broadcast can be achieved given weak broadcast.

Upper bounds for probabilistic broadcast and Byzantine agreement were also studied in [10, 5]. [10] gives an upper bound of  $\frac{2}{3}$  (for the success probability) for the fully synchronous, round-based setting. Somewhat surprisingly, [5] consider a synchronous model with a rushing adversary that can observe the inputs of all other parties in each round before deciding on its own input for the round. In this setting, [5] show the stronger bound of  $(\sqrt{5} - 1)/2$  and also give protocols that match this bound. Such a stronger bound is possible only because the guarantee is stronger and includes a secrecy guarantee: the adversary must not learn the output too early.

## 2 Preliminaries and Notation

Let  $\mathcal{P} = \{P_1, \dots, P_n\}$  be a set of  $n$  parties (also known as players or processors). For convenience, we will sometimes use  $i$  instead of  $P_i$ . We distinguish between the subset of honest parties,  $H \subseteq \mathcal{P}$ , and the dishonest parties in the complement,  $\mathcal{P} \setminus H$ . Honest parties will execute protocol instructions whereas dishonest parties can deviate arbitrarily from the protocol. For a set  $M$  and a subset  $S \subseteq \mathcal{P}$ , we denote by  $M^S$  the Cartesian product  $\times_{i \in S} M$ . Moreover we write  $[n]$  for the set  $\{1, \dots, n\}$ .

### 2.1 Consistency Specifications

Primitives, such as a broadcast channel, provide the honest parties with consistency guarantees. That is, for every set  $H$  of honest parties and every possible choice  $\vec{x}_H$  of inputs, the consistency guarantees restrict the set of possible outputs of the honest parties. In this manner, consistency guarantees limit the influence of dishonest parties on the possible outputs of honest parties. We thus model such primitives as functions called *consistency specifications* that map a set of honest parties along with their inputs to a non-empty set of possible outputs. A smaller set of possible outputs implies stronger guarantees offered by the consistency specification, as the uncertainty over the actual output is smaller. More formally, a consistency specification (introduced in [14]) with input domain  $\mathcal{D}$  and output domain  $\mathcal{R}$  is defined as follows.

► **Definition 1.** A *consistency specification* with input domain  $\mathcal{D}$  and output domain  $\mathcal{R}$  is a function which assigns to every non-empty subset  $H \subseteq \mathcal{P}$  and every input tuple  $\vec{x}_H \in \mathcal{D}^H$  a non-empty set  $\mathcal{C}(H, \vec{x}_H) \subseteq \mathcal{R}^H$  of output tuples and satisfies the following monotonicity constraint: For any non-empty subset  $H' \subseteq H \subseteq \mathcal{P}$

$$\mathcal{C}(H, \vec{x}_H)|_{H'} \subseteq \mathcal{C}(H', \vec{x}_{H|H'}). \quad (1)$$

The monotonicity constraint ensures that larger sets of honest parties do not have weaker consistency guarantees. It is therefore natural to require that  $\mathcal{C}(H, \vec{x}_H)$  is non-empty for any choice of  $H$  and  $\vec{x}_H$  as having no output is as good as has having an arbitrary output.

**Important Consistency Specifications.** We consider two important examples of consistency specifications that we will use throughout this work.

► **Definition 2.** A bit *broadcast channel*  $\text{BC}_i$  for sender  $P_i$  can be defined as the following consistency specification

$$\text{BC}_i(H, \vec{x}_H) = \left\{ \vec{y}_H \in \{0, 1\}^H \mid \begin{array}{l} \exists v \left( (\forall j \in H : \vec{y}_{H \setminus \{j\}} = v) \right) \\ \wedge (i \in H \Rightarrow v = \vec{x}_{H \setminus \{i\}}) \end{array} \right\}.$$

The top right line ensures consistency (all honest parties output the same bit) and the bottom right line ensures validity (if the sender is honest, the output bit is its input bit) condition.

► **Definition 3.** An *authenticated bit-channel*  $\text{AUTH}_{i,j}$  from  $P_i$  to  $P_j$  can be defined as the following consistency specification

$$\text{AUTH}_{i,j}(H, x_H) = \{ \vec{y}_H \in \{0,1\}^H \mid i, j \in H \Rightarrow \vec{y}_{H \setminus \{j\}} = \vec{x}_{H \setminus \{i\}} \}.$$

It guarantees that  $P_j$ 's output is equal to the input of  $P_i$  if both of them are honest.

In the above examples, the inputs of all (honest) parties except  $P_i$  have no influence on the consistency guarantee. Similarly for  $\text{AUTH}_{i,j}$ , the outputs of all (honest) parties except  $P_j$  provide no information (they are arbitrary). We say that such parties have no input, respectively no output. Formally, we define empty inputs and outputs as follows.

► **Definition 4.** Let  $\mathcal{C}$  be a consistency specification with input domain  $\mathcal{D}$  and output domain  $\mathcal{R}$ . A party  $P_i$  has *no input* if for every  $H$  with  $P_i \in H$  and all  $\vec{a}_H, \vec{b}_H \in \mathcal{D}^H$  with  $\vec{a}_H|_{H \setminus \{i\}} = \vec{b}_H|_{H \setminus \{i\}}$  it holds that  $\mathcal{C}(H, \vec{a}_H) = \mathcal{C}(H, \vec{b}_H)$ . A party  $P_i$  has *no output* if for every  $H$  with  $P_i \in H$  and all  $\vec{x}_H$  it holds that  $\mathcal{C}(H, \vec{x}_H)|_{\{i\}} = \mathcal{R}$ .

Finally, we note that the parallel composition of several consistency specifications once again forms a consistency specification. More formally, consider consistency specifications  $\mathcal{C}^{(1)}, \dots, \mathcal{C}^{(\ell)}$  where  $\mathcal{C}^{(j)}$  has input domain  $\mathcal{D}_j$  and output domain  $\mathcal{R}_j$   $j \in [\ell]$ . The parallel composition of  $\mathcal{C}^{(1)}, \dots, \mathcal{C}^{(\ell)}$  is defined as follows.

► **Definition 5.** The *parallel composition* of  $\mathcal{C}^{(1)}, \dots, \mathcal{C}^{(\ell)}$  is the  $(\mathcal{D}, \mathcal{R})$ -consistency specification  $[\mathcal{C}^{(1)}, \dots, \mathcal{C}^{(\ell)}]$  where  $\mathcal{D} = \prod_{j \in [\ell]} \mathcal{D}_j$ ,  $\mathcal{R} = \prod_{j \in [\ell]} \mathcal{R}_j$ , and for every  $H \subseteq \mathcal{P}$  and all  $\vec{x}_H = ((x_{ij})_{j \in [\ell]})_{i \in H} \in \mathcal{D}$  it holds that

$$[\mathcal{C}^{(1)}, \dots, \mathcal{C}^{(\ell)}](H, \vec{x}_H) = \left\{ \vec{y}_H \in \mathcal{R} \mid \begin{array}{l} \vec{y}_H = ((y_{ij})_{j \in [\ell]})_{i \in H} \\ \wedge \forall j (y_{ij})_{i \in H} \in \mathcal{C}^{(j)}(H, (x_{ij})_{i \in H}) \end{array} \right\}.$$

The complete network of authenticated channels can be seen as the parallel composition of authenticated channels.

► **Definition 6.** The complete network  $\text{AUTH}$  of authenticated bit-channels for parties  $\mathcal{P}$  is the parallel composition of the set  $\{\text{AUTH}_{i,j} \mid P_i, P_j \in \mathcal{P}\}$  of all authenticated bit-channels.

## 2.2 Protocols and Constructions.

Protocols are means to construct new consistency specifications from given consistency specifications. A protocol execution is round-based and proceeds as follows. In each round, a party computes an input to the consistency specification used in this round. This input may depend on its protocol input and outputs from previously invoked consistency specifications. At the end of the protocol execution, each party computes its protocol output as a function of its protocol input and all the outputs it received from invoked specifications over the course of the protocol.

**Deterministic Protocols.** A deterministic protocols runs for  $\ell \geq 0$  rounds. In each round  $r$ , party  $P_i$  uses the deterministic round function  $f_i^{(r)}$  to compute its input for the round specification  $\mathcal{C}^{(r)}$  which has input domain  $\mathcal{D}_r$  and output domain  $\mathcal{R}_r$ . At the end of the last round, party  $P_i$  uses its output function  $g_i$  to compute its protocol output. Denote by  $\vec{\mathcal{C}} = (\mathcal{C}^{(r)})_{r \in 1, \dots, \ell}$  the tuple of invoked specifications. Then we can define a deterministic protocol as follows.

► **Definition 7** ([14]). A deterministic  $\ell$ -round protocol  $\pi$  for tuple  $\vec{\mathcal{C}}$  with input domains  $\mathcal{D}$  and output domains  $\mathcal{R}$  consists of round functions

$$f_i^{(r)} : \mathcal{D} \times \mathcal{R}_1 \times \cdots \times \mathcal{R}_{(r-1)} \rightarrow \mathcal{D}_r \quad \forall i \in \mathcal{P} \quad \forall r \in [\ell]$$

and output functions

$$g_i : \mathcal{D} \times \mathcal{R}_1 \times \cdots \times \mathcal{R}_{(\ell)} \rightarrow \mathcal{R} \quad \forall i \in \mathcal{P}.$$

We explicitly allow zero-round protocols where no consistency specifications are invoked. By executing the protocol  $\pi$  using tuple  $\vec{\mathcal{C}}$ , the parties achieve a new consistency specification denoted by  $\pi\vec{\mathcal{C}}$ . The following definition formally defines how the output of  $\pi\vec{\mathcal{C}}$  is computed by iteratively applying the round functions of  $\pi$  to the input tuple  $\vec{x}_H$ .

► **Definition 8.** For a protocol  $\pi$  and the corresponding tuple  $\vec{\mathcal{C}}$  the *protocol specification*  $\pi\vec{\mathcal{C}}$  is the following consistency specification, such that for every  $H \subseteq \mathcal{P}$  and  $\vec{x}_H = (x_i)_{i \in H} \in \mathcal{D}^H$ , we have:

$$\pi\vec{\mathcal{C}}(H, \vec{x}_H) = \left\{ (y_i)_{i \in H} \in \mathcal{R}^H \left| \begin{array}{l} \forall r \in [\ell] \exists (x_{ir})_{i \in H} \in \mathcal{D}_r^H \exists (y_{ir})_{i \in H} \in \mathcal{R}_r^H \\ \forall i \in H: x_{ir} = f_i^{(r)}(x_i, y_{i1}, \dots, y_{ir-1}) \\ \wedge (y_{ir})_{i \in H} \in \mathcal{C}^{(r)}(H, (x_{ir})_{i \in H}) \\ \wedge \forall i \in H y_i = g_i(x_i, y_{i1}, \dots, y_{i\ell}) \end{array} \right. \right\}.$$

The goal of a protocol execution is to achieve a consistency specification whose guarantees are at least as strong as the guarantees of some target specification  $\mathcal{C}$ . As already argued, the consistency guarantee becomes stronger as the set of possible outputs becomes smaller. Therefore, we say that a protocol  $\pi$  *constructs* a consistency specification  $\mathcal{C}$  from the tuple  $\vec{\mathcal{C}}$ , if the set of possible outputs of the protocol specification  $\pi\vec{\mathcal{C}}(H, \vec{x}_H)$  for arbitrary inputs  $H, \vec{x}_H$  to  $\pi\vec{\mathcal{C}}$  is a subset of the corresponding set of possible outputs  $\mathcal{C}(H, \vec{x}_H)$  of the target specification  $\mathcal{C}$ . Formally:

► **Definition 9.** A protocol  $\pi$  *constructs* a specification  $\mathcal{C}$  from the tuple  $\vec{\mathcal{C}}$  if we have for all  $H \subseteq \mathcal{P}$  and all  $\vec{x}_H$   $\pi\vec{\mathcal{C}}(H, \vec{x}_H)$  that  $\subseteq \mathcal{C}(H, \vec{x}_H)$ .

Often, one is interested in a broader notion of construction where specifications from a set  $\mathfrak{C}$  may be invoked arbitrarily often during a protocol execution.

► **Definition 10.** A specification  $\mathcal{C}$  can be constructed from a set of specifications  $\mathfrak{C}$ , denoted by  $\mathfrak{C} \longrightarrow \mathcal{C}$ , if there exists a tuple  $\vec{\mathcal{C}}$  of specifications from  $\mathfrak{C}$  (including parallel compositions) which allows to construct  $\mathcal{C}$ .

The above definition naturally extends to a construction notion among sets of consistency specifications: A set of consistency specifications  $\mathfrak{C}'$  is constructible from  $\mathfrak{C}$ , denoted by  $\mathfrak{C} \longrightarrow \mathfrak{C}'$  if all  $\mathcal{C} \in \mathfrak{C}'$  can be constructed from  $\mathfrak{C}$ .

**Probabilistic Protocols.** In a probabilistic protocol, the parties may additionally use local randomness during the protocol execution. Formally, probabilistic protocols are modeled as distributions over deterministic protocols.

► **Definition 11.** A *probabilistic protocol*  $\ell$ -round  $\Pi$  for tuple  $\vec{\mathcal{C}}_\Pi$  with input domains  $\mathcal{D}$  and output domains  $\mathcal{R}$  is a random variable (for some distribution) over a set of deterministic protocols of at most  $\ell$ -rounds for tuple  $\vec{\mathcal{C}}_\Pi$  with input domains  $\mathcal{D}$  and output domains  $\mathcal{R}$ .

Note that our definition allows for protocols where parties have access to correlated randomness. We denote by  $\Pi\vec{\mathcal{C}}_{\Pi}$  the random variable over the protocol specifications for  $\Pi$  and  $\vec{\mathcal{C}}_{\Pi}$ . A protocol constructs a target specification  $\mathcal{C}$  within  $\epsilon$  if with probability strictly larger than  $1 - \epsilon$ ,  $\Pi\vec{\mathcal{C}}_{\Pi}$  provides better consistency guarantees than  $\mathcal{C}$ . Formally:

► **Definition 12.** A probabilistic protocol  $\Pi$  for tuple  $\vec{\mathcal{C}}_{\Pi}$  constructs  $\mathcal{C}$  within  $\epsilon$  if

$$\min_{H, \vec{x}_H} \mathbb{P}(\Pi\vec{\mathcal{C}}_{\Pi}(H, \vec{x}_H) \subseteq \mathcal{C}(H, \vec{x}_H)) > 1 - \epsilon.$$

A construction is called *perfect* if  $\epsilon = 0$ . A specification  $\mathcal{C}$  can be constructed within  $\epsilon$  from a set  $\mathfrak{C}$ , denoted by  $\mathfrak{C} \xrightarrow{\epsilon} \mathcal{C}$ , if there exists a tuple  $\vec{\mathcal{C}}_{\Pi}$  from  $\mathfrak{C}$  which allows to construct  $\mathcal{C}$  within  $\epsilon$ .

Note that any deterministic construction is a perfect construction.

### 3 Impossibility Proofs

In this section, we consider a generalized version of so called ‘scenario’-proofs (see e.g., [2]). This proof technique, a special type of proof by contradiction, is normally used to prove that a specification, e.g., broadcast, cannot be constructed from authenticated channels within some  $\epsilon$ . Here, we extend ‘scenario’-proofs to the setting where parties are given additional setup. This means that we want to prove statements of the form “There is no construction of a specification  $\mathcal{C}$  from given specifications  $\mathfrak{C}$  within  $\epsilon$ ” where  $\mathfrak{C}$  is arbitrary set of specification which contains the complete network of authenticated channels.

More formally, the technique allows to prove a *claim* of the form: “ $\mathcal{C}$  cannot be constructed from  $\mathfrak{C}$  within  $\epsilon = \frac{1}{k}$  where  $\text{AUTH} \in \mathfrak{C}$ .” The corresponding ‘scenario’-proof goes as follows (for a simple example of such a proof, see the proof of Lemma 14). Towards a contradiction, assume that there exists a protocol  $\Pi$  which allows to construct  $\mathcal{C}$  from  $\mathfrak{C}$  within  $\frac{1}{k}$ . This implies that for each party  $P_i$  and for each input  $x_i$ , there exists a corresponding (probabilistic) protocol system  $\Pi_i^{x_i}$  which executes the protocol part of  $P_i$  for input  $x_i$ <sup>2</sup>. For every other party  $P_j$ , the protocol system of a party  $P_i$  has an interface where one can connect it to  $P_j$ ’s protocol system. This models the assumption that parties are pair-wise connected via authenticated channels. If the parties are given additional specifications in  $\mathfrak{C}$  (e.g., broadcast channels for some parties) or some setup (e.g. shared randomness) during the protocol execution, this is modeled via a system  $R$  that provides the functionality of these specifications. In this case, all protocol systems have an additional interface where they expect to be connected to  $R$ .

The creative part of the proof is to build a *configuration* of connected protocol systems and  $R$ , which has impossible output guarantees. This implies that there is no construction of  $\mathcal{C}$  from  $\mathfrak{C}$  within  $\frac{1}{k}$ . More formally, we consider a configuration  $S$  and the output vector of selected protocol systems which we denote by the random variable  $\mathbf{Y}$ . To show that  $S$  has impossible output guarantees, we use the following technical lemma.

► **Lemma 13.** Let  $A_1, \dots, A_k$  be sets with non-empty union  $A = \bigcup_{i=1}^k A_i$  and let  $\mathbf{Y}$  be a random variable over some set  $U \supseteq A$  such that  $\mathbb{P}(\mathbf{Y} \in \bigcap_{i=1}^k A_i) = 0$ . Then  $\min_i \mathbb{P}(\mathbf{Y} \in A_i) \leq 1 - \frac{1}{k}$ .

<sup>2</sup> Such a system can be instantiated, for example, as an interactive Turing machine.

**Proof.** For convenience we denote for any set  $B$  by  $P(B)$  the probability  $P(\mathbf{Y} \in B)$ . We denote by  $\overline{B}$  the complement of  $B$  in  $U$ . Using elementary set operations and the union bound we get

$$\begin{aligned} P\left(\bigcap_{i=1}^k A_i\right) &= 1 - P\left(\bigcup_{i=1}^k \overline{A_i}\right) \geq 1 - \sum_{i=1}^k P(\overline{A_i}) \\ &= 1 - \sum_{i=1}^k (1 - P(A_i)) = 1 - k + \sum_{i=1}^k P(A_i). \end{aligned}$$

As the minimum overall  $P(\mathbf{Y} \in A_i)$  is smaller than the average we finally get

$$\begin{aligned} \min_i P(\mathbf{Y} \in A_i) &\leq \frac{1}{k} \sum_{i=1}^k P(A_i) \\ &\leq \frac{1}{k} \left(k - 1 + P\left(\bigcap_{i=1}^k A_i\right)\right) = 1 - \frac{1}{k}. \end{aligned} \quad \blacktriangleleft$$

To get to a contradiction, we thus need to show that there are  $k$  sets (of outputs)  $A_1, \dots, A_k$  with empty intersection, where  $\mathbf{Y} \in A_i$  with probability strictly greater than  $1 - \frac{1}{k}$  for any  $i$ . To do so, we use  $k$  so-called *scenarios*. Each scenario describes  $S$  as a protocol execution among three parties where exactly one of them is dishonest. With the exception of two systems (for the two honest parties), all parts of  $S$  are considered to be the ‘attack strategy’ of the dishonest party. The initial assumption implies that the outputs of the two honest parties in this scenario must satisfy some consistency guarantee with probability strictly greater than  $1 - \frac{1}{k}$ . This directly translates into a condition on  $\mathbf{Y}$ . Namely, for the  $i$ th scenario, there must exist a set of outputs  $A_i$  such that  $P(\mathbf{Y} \in A_i) > 1 - \frac{1}{k}$ . To arrive at the desired contradiction, the  $k$  scenarios are chosen such that the intersection of all  $A_i$ ’s is empty and therefore  $P(\mathbf{Y} \in \bigcap_{i=1}^k A_i) = 0$ . In this case, the above lemma implies that for at least one  $A_i$ , it must hold that  $P(\mathbf{Y} \in A_i) \leq 1 - \frac{1}{k}$ , thus contradicting the fact that for all  $i$ ,  $P(\mathbf{Y} \in A_i) > 1 - \frac{1}{k}$  (as required by the assumption of a construction within  $\epsilon = \frac{1}{k}$ ).

## 4 Results

In this section we consider specifications for party set  $\mathcal{P} = \{P_1, P_2, P_3\}$  where all inputs and outputs are bit-strings.

### 4.1 Strong Broadcast Impossibility

Here, we prove a strong impossibility for the construction of broadcast. That is, we show that broadcast channel, e.g.  $\text{BC}_1$ , cannot be constructed within  $\frac{1}{3}$  even if all other broadcast channels are available. This implies the well known result by Karlin and Yao [10] that broadcast cannot be constructed from authenticated channels within  $\frac{1}{3}$ .

As a warm up, we prove first the [10] statement using the impossibility techniques from above.

► **Lemma 14.**  $[10] \text{ AUTH} \not\stackrel{\frac{1}{3}}{\rightarrow} \text{BC}_1$ .

**Proof.** Towards a contradiction, let us assume that there exists a protocol  $\Pi$  such that  $\text{AUTH} \xrightarrow{\Pi, \frac{1}{3}} \text{BC}_1$ . Then there exist protocol systems  $\Pi_1^0, \Pi_1^1, \Pi_2, \Pi_3$ . Note that only the

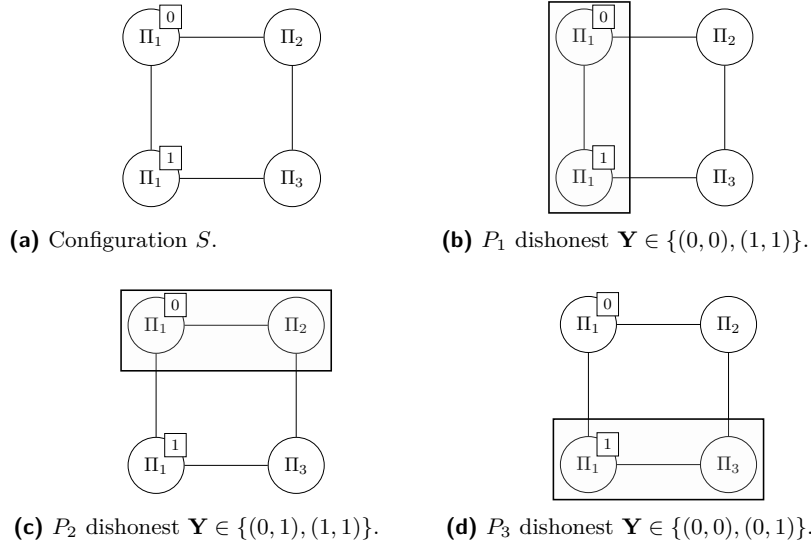


Figure 1 The configuration  $S$  and the three scenarios.

system of  $P_1$  has an input. Each of these systems has two interfaces where it expects to be connected to the systems of the other two parties.

We consider the configuration  $S$  in Figure 1a where all four systems are arranged in a circle. The random variable  $\mathbf{Y}$  describes the output behavior of systems  $\Pi_2$  and  $\Pi_3$ . This means that  $\mathbf{Y}$  maps to bit-tuples where the first component represents the output of  $\Pi_2$ .

We examine the distribution of  $\mathbf{Y}$  using different protocol execution scenarios. First, we consider the scenario where  $P_2$  and  $P_3$  are honest while  $P_1$  is dishonest, i.e.,  $H = \{P_2, P_3\}$ . In this scenario, consistency of broadcast ensures that the outputs of  $P_2$  and  $P_3$  are with probability strictly larger than  $1 - \frac{1}{3}$  the same (independently of the behavior of  $P_1$ ). In the configuration  $S$ , this corresponds to the scenario where the system of  $P_1$  consists of the two left-most systems (cf. Figure 1b). This implies that  $\mathbf{Y}$  is in  $A_1 = \{(0, 0), (1, 1)\}$  with probability strictly larger than  $1 - \frac{1}{3}$ . Next, we consider the scenario where  $P_1$  and  $P_3$  are honest ( $H = \{P_1, P_3\}$ ) and  $P_1$  has input 1. In our configuration  $S$ , we can perceive the two systems on the top as the system of the dishonest  $P_2$  (cf. Figure 1c). This implies (validity of broadcast) that  $\mathbb{P}(\mathbf{Y} \in A_2) > 1 - \frac{1}{3}$  for  $A_2 = \{(0, 1), (1, 1)\}$ . Finally, we consider the case  $H = \{P_1, P_3\}$  where  $P_1$  has input 0. In our configuration  $S$ , we can perceive the two systems at the bottom as the system of the dishonest  $P_3$  (cf. Figure 1d). This implies (validity of broadcast) that  $\mathbb{P}(\mathbf{Y} \in A_3) > 1 - \frac{1}{3}$  for  $A_3 = \{(0, 0), (0, 1)\}$ .

We observe that  $A_1 \cap A_2 \cap A_3 = \emptyset$  and thus  $\mathbb{P}(\mathbf{Y} \in \bigcap_{i=1}^3 A_i) = 0$ . This implies with Lemma 13 that for at least one  $A_i$ ,  $\mathbb{P}(\mathbf{Y} \in A_i) \leq 1 - \frac{1}{3}$ . This is a contradiction to the fact that  $\mathbb{P}(\mathbf{Y} \in A_i) > 1 - \frac{1}{3}$  for all  $A_i$ , as required by the definition of a construction within  $\epsilon = \frac{1}{3}$ . Thus, there exists no  $\epsilon$ -construction of broadcast for  $\epsilon \leq \frac{1}{3}$ . ◀

► **Theorem 15.**  $\{\text{AUTH}, \text{BC}_2, \text{BC}_3\} \not\stackrel{\frac{1}{3}}{\rightarrow} \text{BC}_1$ .

**Proof.** To prove this result we use the ‘scenario’-proof technique from Section 3. Assume therefore that there exists a probabilistic protocol  $\Pi$  which allows to construct  $\text{BC}_1$  from  $\{\text{AUTH}, \text{BC}_2, \text{BC}_3\}$  within  $\epsilon = \frac{1}{3}$ . Thus, there exist protocol systems  $\Pi_1^0, \Pi_1^1, \Pi_2, \Pi_3$  where the bit on top of  $\Pi_1$  denotes the input of sender  $P_1$ . Additionally there exists a system  $[\text{BC}_2, \text{BC}_3]$  which corresponds to the given broadcast channels for  $P_2$  and  $P_3$ .



We first show how to construct a system  $\overline{BC}$  from the system  $[BC_2, BC_3]$ . This system  $\overline{BC}$  will be used to build the configuration  $S$ , rather than  $[BC_2, BC_3]$  directly. Thus,  $\overline{BC}$  corresponds to the system  $R$  in our informal description from Section 3. System  $\overline{BC}$  is essentially the same as  $[BC_2, BC_3]$  except that the interface of  $P_1$  is cloned. More precisely,  $\overline{BC}$  has four interfaces. The two interfaces for parties  $P_2$  and  $P_3$  have the same input/output behavior as in  $[BC_2, BC_3]$ . However, the interface for  $P_1$  appears twice in  $\overline{BC}$ , where both copies deliver the same output. Note that this completely describes the behaviour of  $\overline{BC}$ , since  $P_1$ 's interface does not take input in  $[BC_2, BC_3]$  (and thus, it also does not take an input in  $\overline{BC}$ ).

System  $\overline{BC}$  can be built from  $[BC_2, BC_3]$  in three different ways. First, one can build it by adding a system  $e_1$  to the  $P_1$ -interface of  $[BC_2, BC_3]$  which relays the outputs of this interface to the two  $P_1$ -interfaces of  $\overline{BC}$ . Second, one can build  $\overline{BC}$  from  $[BC_2, BC_3]$  by adding a system  $e_2$  to the  $P_2$ -interface of  $[BC_2, BC_3]$ . System  $e_2$  relays any input at the  $\overline{BC}$   $P_2$ -interface to  $[BC_2, BC_3]$ . Any output at the  $P_2$ -interface of  $[BC_2, BC_3]$  is relayed to the  $\overline{BC}$   $P_1$ -interface and the  $\overline{BC}$   $P_2$ -interfaces of  $e_2$ , respectively. Note that adding system  $e_2$  in this way achieves the same as adding  $e_1$ . This is true, because in  $[BC_2, BC_3]$ , the outputs at any interface are always identical, due to the consistency guarantees of  $BC_2$  and  $BC_3$ . Analogously, one can build  $\overline{BC}$  from  $[BC_2, BC_3]$  by adding a system  $e_3$  to the  $P_3$ -interface of  $[BC_2, BC_3]$ . In summary we have that the systems  $\overline{BC}$ ,  $e_1[BC_2, BC_3]$ ,  $e_2[BC_2, BC_3]$ , and  $e_3[BC_2, BC_3]$  have the same input/output behavior.

We consider now the configuration  $S$  in Figure 2a and the output  $\mathbf{Y}$  of systems  $\Pi_2$  and  $\Pi_3$ . It follows from the above argumentation that the configurations seen in Figures 2b-2d have the same output behavior  $\mathbf{Y}$  as  $S$ .

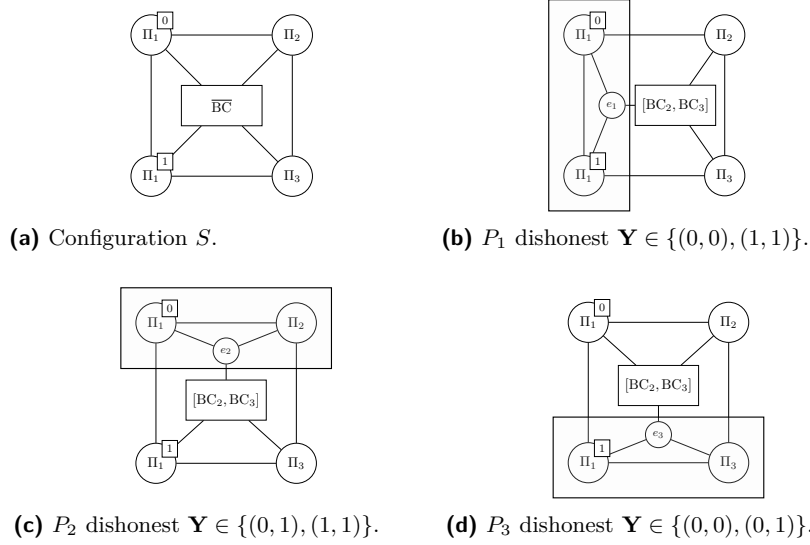
We examine the distribution of  $\mathbf{Y}$  using different protocol execution scenarios. First, we consider the scenario where  $P_1$  is dishonest, i.e,  $H = \{P_2, P_3\}$ . The consistency of  $BC_1$  implies that with probability strictly larger than  $1 - \frac{1}{3}$ , the outputs of  $P_2$  and  $P_3$  are the same. In this scenario, the adversarial  $P_1$  could control a system consisting of the three left-most systems in Figure 2b. The consistency of broadcast thus implies for  $S$  that  $P(\mathbf{Y} \in A_1) > 1 - \frac{1}{3}$ , where  $A_1 = \{(0, 0), (1, 1)\}$ . Next, we consider the scenario  $H = \{P_1, P_3\}$  where  $P_1$  has input 1. Here, dishonest  $P_2$  could run the top-three systems in Figure 2c. The validity condition of  $BC_1$  implies that  $P(\mathbf{Y} \in A_2) > 1 - \frac{1}{3}$  for  $A_2 = \{(0, 1), (1, 1)\}$ . Finally, we consider the scenario  $H = \{P_1, P_2\}$  where  $P_1$  has input 0. Here, dishonest  $P_3$  could run the bottom-three systems in Figure 2d. The validity condition of  $BC_1$  implies that  $P(\mathbf{Y} \in A_3) > 1 - \frac{1}{3}$  for  $A_3 = \{(0, 0), (0, 1)\}$ . The intersection  $A_1 \cap A_2 \cap A_3$  is empty and hence  $P(\mathbf{Y} \in A_1 \cap A_2 \cap A_3) = 0$ . Now, Lemma 13 implies that for at least one  $A_i$ ,  $P(\mathbf{Y} \in A_i) \leq 1 - \frac{1}{3}$ . This is a contradiction to the fact that  $P(\mathbf{Y} \in A_i) > 1 - \frac{1}{3}$  for all  $A_i$  as required by the definition of a construction within  $\epsilon = \frac{1}{3}$ . Thus no construction of broadcast  $BC_1$  from  $\{\text{AUTH}, BC_2, BC_3\}$  exists within  $\epsilon = \frac{1}{3}$ . ◀

► **Corollary 16.** *In particular, for every protocol  $\Pi$  which constructs broadcast  $BC_1$  from  $\{\text{AUTH}, BC_2, BC_3\}$ , there exists  $H \subseteq \mathcal{P}$  of size two such that*

$$P(\Pi(\text{AUTH}, BC_2, BC_3)(H, \vec{x}_H) \subseteq BC_1(H, \vec{x}_H)) \leq 1 - \frac{1}{3}.$$

## 4.2 Strong Separation of Broadcast and Authenticated Channels

In this section, we prove a strong separation between broadcast and authenticated channels. That is, we present a specification, called XOR-cast, which neither can be constructed from authenticated channels within a constant  $\epsilon$ , nor is sufficient to construct broadcast within



■ **Figure 2** The configuration  $S$  and the three scenarios. A line between two systems means that they are connected. In the case of protocol systems this corresponds to the fact that parties can communicate over authenticated channels.

a constant  $\epsilon$ . XOR-cast takes a bit  $b_i$  from  $P_i$  and a bit  $b_j$  from  $P_j$  as input. If all parties behave correctly, the value of  $b_i \oplus b_j$  should be output by all parties. If one of the parties  $P_i, P_j$  is dishonest, the honest parties should output the same value. If the third party  $P_k$  is dishonest, the remaining parties should output  $b_i \oplus b_j$ .

► **Definition 17.** Let  $P_i, P_j \in \mathcal{P}$  be distinct parties. The *XOR-cast*  $\text{XC}_{i,j}$  for  $P_i$  and  $P_j$  is defined as follows.

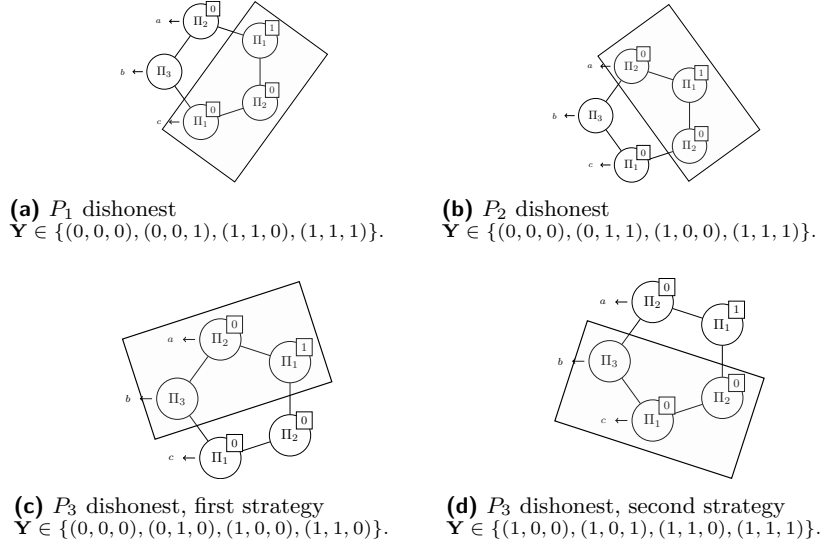
$$\text{XC}_{i,j}(H, \vec{x}_H) = \left\{ \vec{y} \in \{0, 1\}^H \mid \begin{array}{l} \exists v \left( (\forall \ell \in H : \vec{y}_{H \setminus \{\ell\}} = v) \right. \\ \left. \wedge (i, j \in H \Rightarrow v = \vec{x}_{H \setminus \{i\}} \oplus \vec{x}_{H \setminus \{j\}}) \right) \end{array} \right\}.$$

The top right line in the equation ensures that all honest parties output the same value. The bottom right line ensures for honest  $P_i$  and  $P_j$  that the output is the XOR of their input-bits.

We first prove that XOR-cast, e.g.,  $\text{XC}_{1,2}$ , cannot be constructed from the network of authenticated channels.

► **Lemma 18.**  $\{\text{AUTH}\} \not\stackrel{\frac{1}{4}}{\rightarrow} \text{XC}_{1,2}$ .

**Proof.** We again use the ‘scenario’-proof technique. Towards a contradiction, assume that there exists a protocol allowing to construct  $\text{XC}_{1,2}$  from  $\{\text{AUTH}\}$  within  $\frac{1}{4}$ . Then there exist protocol systems  $\Pi_1^{x_1}, \Pi_2^{x_2}, \Pi_3$  for parties  $P_1, P_2, P_3$  where  $x_1$  denotes the input bit of  $P_1$  and  $x_2$  denotes the input bit of  $P_2$ . Consider the pentagon configuration  $S$  in Figure 3. Let  $\mathbf{Y}$  be the random variable over the output  $(a, b, c)$  of the three left-most systems, i.e., where  $a$  is the output of  $\Pi_2^0$  (top left),  $b$  the output of  $\Pi_3$  (middle left), and  $c$  the output of  $\Pi_1^0$  (bottom left). We examine the distribution of  $\mathbf{Y}$  using four different protocol execution scenarios. First, we consider the scenario where  $P_2$  and  $P_3$  are honest ( $H = \{P_2, P_3\}$ ) and  $P_1$  has input 0. In this scenario, the dishonest  $P_1$  could run the three systems in the bottom-left in Figure 3a. The outputs of  $P_2$  and  $P_3$  must be the same. This implies  $\text{P}(\mathbf{Y} \in A_1) > 1 - \frac{1}{4}$



■ **Figure 3** The configuration  $S$  and the four scenarios.

for  $A_1 = \{(0, 0, 0), (0, 0, 1), (1, 1, 0), (1, 1, 1)\}$ . Next, we consider the scenario  $H = \{P_1, P_3\}$  where  $P_1$  has input 0 (cf. Figure 3b). Here, the outputs of  $P_1$  and  $P_3$  must be the same. This implies that  $\mathbf{P}(\mathbf{Y} \in A_2) > 1 - \frac{1}{4}$  for  $A_2 = \{(0, 0, 0), (0, 1, 1), (1, 0, 0), (1, 1, 1)\}$ . Next, we consider the scenario  $H = \{P_1, P_2\}$  where both  $P_1$  and  $P_2$  have input 0 (cf. Figure 3c). Here, the output of  $P_1$  must be  $0 = 0 \oplus 0$ . This implies that  $\mathbf{P}(\mathbf{Y} \in A_3) > 1 - \frac{1}{4}$  for  $A_3 = \{(0, 0, 0), (0, 1, 0), (1, 0, 0), (1, 1, 0)\}$ . Finally, we consider the scenario  $H = \{P_1, P_2\}$  where  $P_1$  has input 1 and  $P_2$  has input 0 (cf. Figure 3d). Here, the output of  $P_2$  must be  $1 = 1 \oplus 0$ . This implies that  $\mathbf{P}(\mathbf{Y} \in A_4) > 1 - \frac{1}{4}$  for  $A_4 = \{(1, 0, 0), (1, 0, 1), (1, 1, 0), (1, 1, 1)\}$ .

We observe that the intersection  $A_1 \cap A_2 \cap A_3 \cap A_4$  is empty and hence  $\mathbf{P}(\mathbf{Y} \in \bigcap_{i=1}^4 A_i) = 0$ . This implies with Lemma 13 that for at least one  $A_i$ ,  $\mathbf{P}(\mathbf{Y} \in A_i) \leq 1 - \frac{1}{4}$ . This is a contradiction to the fact that  $\mathbf{P}(\mathbf{Y} \in A_i) > 1 - \frac{1}{4}$  for all  $A_i$  as required by the definition of a construction within  $\epsilon = \frac{1}{4}$ . Thus no construction of  $\text{XC}_{1,2}$  from  $\text{AUTH}$  exists within  $\frac{1}{4}$ . ◀

► **Corollary 19.** *In particular, for every protocol  $\Pi$ , there exists  $H \subset \mathcal{P}$ ,  $|H| = 2$ , such that*

$$\mathbf{P}(\Pi(\text{AUTH})(H, \vec{x}_H) \subseteq \text{XC}_{1,2}(H, \vec{x}_H)) \leq 1 - \frac{1}{4}.$$

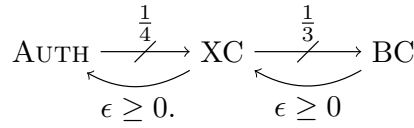
Next, we show that one can perfectly construct  $\text{XC}_{i,j}$  given the complete network of authenticated channels and a broadcast channel for  $P_i$  or  $P_j$ .

► **Lemma 20.** *For all  $i \neq j \in \{1, 2, 3\}$   $\{\text{AUTH}, \text{BC}_i\} \rightarrow \text{XC}_{i,j}$ .*

**Proof.** Let  $b_i$  be the input of  $P_i$  and let  $b_j$  be the input of  $P_j$  and denote by  $P_k$  the third party. Consider the following protocol.

1.  $P_j$  sends  $b_j$  to  $P_i$ . Denote by  $\hat{b}_j$  the bit received by  $P_i$ .
2.  $P_i$  broadcasts  $b_k := b_i \oplus \hat{b}_j$  using  $\text{BC}_i$ . Denote by  $\hat{b}_k$  the bit received by  $P_j$  and  $P_k$ .
3.  $P_i$  outputs  $b_k$ ,  $P_j$  and  $P_k$  both output  $\hat{b}_k$ .

If at least  $P_i$  and  $P_j$  are honest we have  $\hat{b}_j = b_j$  and  $\hat{b}_k = b_k$ . All honest parties will output  $b_k = b_i \oplus b_j$  as required by  $\text{XC}_{i,j}$ . On the other hand if  $H = \{P_j, P_k\}$  both honest parties will output  $\hat{b}_k$  as required by  $\text{XC}_{i,j}$ . If  $H = \{P_i, P_k\}$  we have  $\hat{b}_k = b_k$ . Both honest parties will output  $b_k$  as required by  $\text{XC}_{i,j}$ . If at most one party is honest any output is fine, thus the protocol achieves the construction also in those cases. ◀



■ **Figure 4** XOR-cast strongly separates AUTH and BC.

Finally, we show that XOR-cast is strictly weaker than broadcast. Even given all three XOR-casts, one cannot construct a single broadcast channel. Without loss of generality, we show that one cannot construct  $\text{BC}_1$  given all XOR-casts within  $\epsilon \leq \frac{1}{3}$ .

► **Lemma 21.**  $\{\text{XC}_{1,2}, \text{XC}_{1,3}, \text{XC}_{2,3}, \text{AUTH}\} \not\rightarrow^{\frac{1}{3}} \text{BC}_1$ .

**Proof.** Towards a contradiction, let us assume that one can construct  $\text{BC}_1$  given the XOR-casts, i.e.,  $\{\text{XC}_{1,2}, \text{XC}_{1,3}, \text{XC}_{2,3}, \text{AUTH}\} \rightarrow \text{BC}_1$  within  $\epsilon \leq \frac{1}{3}$ . Lemma 20 implies that one can perfectly construct all XOR-casts given broadcast channels  $\text{BC}_2, \text{BC}_3$ . This implies that one can construct  $\text{BC}_1$  from  $\{\text{BC}_2, \text{BC}_3, \text{AUTH}\}$  within  $\epsilon \leq \frac{1}{3}$ , a contradiction to Lemma 15. ◀

The above lemmas directly imply the following theorem.

► **Theorem 22.** *Authenticated channels and broadcast are strongly separated by XOR-cast.*

### 4.3 Weak Broadcast

For comparison, we consider *weak broadcast* which was introduced in [11]. This specification provides the same consistency guarantees as broadcast except that validity only holds if all parties are honest.

► **Definition 23.** Let  $P_s \in \mathcal{P}$ . A *weak broadcast-channel*  $\text{wBC}_s$  for sender  $P_s$  is defined to be a  $(\{0, 1\}, \{0, 1\})$ -consistency specification where for every  $H \subseteq \mathcal{P}$  and all  $\vec{x}_H \in \{0, 1\}^H$  it holds that

$$\begin{aligned} & \text{wBC}_s(H, \vec{x}_H) \\ &= \left\{ \vec{y}_H \in \{0, 1\}^H \mid \exists v \left( (\forall j \in H : \vec{y}_{H \setminus \{j\}} = v) \right) \right. \\ & \quad \left. \wedge (H = \mathcal{P} \Rightarrow v = \vec{x}_{H \setminus \{s}}) \right\}. \end{aligned}$$

It was shown in [11] that weak broadcast cannot be constructed from authenticated channels using a deterministic protocol.

► **Lemma 24.** [11] *There exists no deterministic  $r$ -round protocol  $\Pi$  which allows for  $\{\text{AUTH}\} \rightarrow \text{wBC}_i$ .*

**Proof.** Without loss of generality, let  $P_1$  be the sender. Suppose there exists a deterministic  $r$ -round protocol  $\Pi$  which allows to construct  $\text{wBC}_1$  from AUTH. Then, there exist protocol systems  $\Pi_1^x, \Pi_2, \Pi_3$  for parties  $P_1, P_2, P_3$ , where  $x$  denotes the input of  $P_1$ . Choose  $k > r + 1$  as a multiple of 3 and arrange  $4k$  such systems in a ring as follows: Start with a system  $\Pi_1^0$  and continue with systems  $\Pi_2, \Pi_3$ ; each system is connected via authenticated channels to its predecessor and successor. Now repeat this pattern going clockwise, until  $2k$  systems

## 36:14 Strong Separations Between Broadcast and Authenticated Channels

have been connected in this manner. Because  $k$  is a multiple of three, the last system in this arrangement will be a system  $\Pi_3$ . Now, restart the pattern from the end of this arrangement, but instead of  $\Pi_1^0$ , use  $\Pi_1^1$ . Arrange another  $2k$  nodes in this manner, thereby closing the ring.

Consider the system  $\Pi_1^0$  at “the top” of the ring. As all systems in the ring are deterministic the view of  $\Pi_1^0$  after  $r$  rounds is the same as if the system were run in a triangular configuration (where the triangle consists of  $\Pi_1^0, \Pi_2, \Pi_3$ ). The validity of weak broadcast implies that the system  $\Pi_1^0$  must output 0. Similarly, the system  $\Pi_1^1$  at “the bottom” of the ring must output 1. Now, consider any two adjacent systems in the ring. One can view the rest of the ring as an attack strategy of a corrupted party. Thus by consistency of weak broadcast any two adjacent systems must output the same value. We thus arrive at a contradiction. ◀

On the other hand, the results of [3] imply that weak broadcast can be achieved from authenticated channels for any  $\epsilon > 0$ .

► **Lemma 25.** [3] For any  $\epsilon > 0$   $\{\text{AUTH}\} \xrightarrow{\epsilon} \text{wBC}_i$ .

Finally, we show that weak broadcast is separated from broadcast. More precisely, we show that broadcast allows to construct weak broadcast while on the other hand broadcast cannot be constructed from weak broadcast within  $\epsilon \leq \frac{1}{3}$ .

► **Theorem 26.** Weak broadcast and broadcast are strongly separated.

The theorem follows from the following two lemmata.

► **Lemma 27.** For all  $i \in \{1, 2, 3\}$   $\{\text{BC}_i\} \longrightarrow \text{wBC}_i$ .

**Proof.** For all  $H$  and all  $\vec{x}_H$  it holds that  $\text{BC}_i(H, \vec{x}_H) \subseteq \text{wBC}_i(H, \vec{x}_H)$ . This directly implies  $\{\text{BC}_i\} \longrightarrow \text{wBC}_i$ . ◀

► **Lemma 28.** For all  $i \in \{1, 2, 3\}$   $\{\text{wBC}_i, \text{AUTH}\} \not\xrightarrow{\frac{1}{3}} \text{BC}_i$ .

**Proof.** We first show that  $\text{XC}_{i,j}$  for  $j \neq i$  is enough to construct  $\text{wBC}_i$ . The following protocol  $\pi$  allows  $P_i$  to weak broadcast its bit  $b$  using  $\text{XC}_{i,j}$ .

1.  $\text{XC}_{i,j}$  is invoked where  $P_i$  inputs  $b$  and  $P_j$  inputs 0. Denote by  $b_i, b_j, b_k$  the bits the parties  $P_i, P_j, P_k$  receive as output from  $\text{XC}_{i,j}$ .
2.  $P_i$  outputs  $b_i$ ,  $P_j$  outputs  $b_j$  and  $P_k$  outputs  $b_k$ .

The properties of  $\text{XC}_{i,j}$  ensure that honest parties will always output the same bit, as required by the consistency of  $\text{wBC}_i$ . If at least  $P_i$  and  $P_j$  are honest, the output of  $\text{XC}_{i,j}$  is  $b = b \oplus 0$ . The protocol thus achieves the validity condition required by  $\text{wBC}_i$ .

From Lemma 21, we know that  $\{\text{XC}_{i,j}, \text{AUTH}\} \not\xrightarrow{\frac{1}{3}} \text{BC}_i$ . This implies that  $\text{BC}_i$  cannot be constructed from  $\{\text{wBC}_i, \text{AUTH}\}$  within  $\epsilon \leq \frac{1}{3}$ . ◀

In summary, considering constructions for  $\epsilon > 0$ , weak broadcast is not stronger than authenticated channels. It is only when considering perfect constructions that weak broadcast provides strictly stronger guarantees. This is in contrast to XOR-cast which is stronger than authenticated channels for any  $\epsilon \geq 0$ .

## 5 Extension to the n-Party Case

In this section, we show how our theorems can be generalized to the  $n$ -party case. Note that our formal definition of XOR-cast can be used without modification for the setting with  $n$  parties. An informal explanation of the resulting specification is as follows. Again, parties  $P_i$  and  $P_j$  each input bits  $b_i$  and  $b_j$ . As in the three-party setting, if all parties behave correctly, the value of  $b_i \oplus b_j$  should be output by all parties. If one or both of the parties  $P_i, P_j$  is dishonest, the honest parties should output the same value. In any other case, the remaining honest parties should output  $b_i \oplus b_j$ .

We begin by proving an  $n$ -party analogon of Theorem 15. Informally, we prove that, given any set of at most  $\frac{2n}{3}$  distinct broadcast channels, no further broadcast channels can be achieved.

► **Theorem 29.** *Let  $\mathcal{B} = \{\text{BC}_{\frac{n}{3}+1}, \dots, \text{BC}_n\}$ . Then  $\{\text{AUTH}\} \cup \mathcal{B} \not\stackrel{\frac{1}{3}}{\rightarrow} \text{BC}_1$ .*

**Proof.** We show that the existence of such a protocol would contradict Corollary 16. Thus, assume that there exists a protocol  $\Pi$  which allows to construct  $\text{BC}_k$  from  $\{\text{AUTH}\} \cup \mathcal{B}$  within  $\epsilon = \frac{1}{3}$ . In particular,  $\forall H' \subseteq \mathcal{P}$  of size  $\frac{2n}{3}$  we have that

$$\mathbb{P}(\Pi(\text{AUTH}, \mathcal{B})(H, \vec{x}_{H'}) \subseteq \text{BC}_1(H', \vec{x}_{H'})) > 1 - \epsilon. \quad (2)$$

We show now that this implies the existence of a protocol  $\Pi'$  which allows to construct  $\text{BC}_1$  within  $\frac{1}{3}$  in the three-party setting. In particular, for protocol  $\Pi'$  it will hold that  $\forall H \subseteq \{P_1, P_2, P_3\}$  of size two that  $\mathbb{P}(\Pi'(\text{AUTH}, \text{BC}_2, \text{BC}_3)(H, \vec{x}_H) \subseteq \text{BC}_1(H, \vec{x}_H)) > 1 - \epsilon$ , which is a direct contradiction of Corollary 16.

The idea of  $\Pi'$  is to execute protocol  $\Pi$  where each of the three parties  $P_1, P_2, P_3$  emulates  $\frac{n}{3}$  of the  $n$  parties. Concretely, party  $P_1$  emulates virtual parties  $P_1, \dots, P_{\frac{2n}{3}}$ , party  $P_2$  emulates  $P_{\frac{n}{3}+1}, \dots, P_{\frac{2n}{3}}$ , and party  $P_3$  emulates  $P_{\frac{2n}{3}+1}, \dots, P_n$ . Clearly, all communication between virtual parties that occurs over authenticated channels can easily be emulated. Similarly, if a party  $P_i, i \in \{\frac{n}{3} + 1, \dots, n\}$  broadcasts in  $\Pi$ , then the party  $P_2$  or  $P_3$  emulating  $P_i$  can use  $\text{BC}_2$  or  $\text{BC}_3$ , respectively, to carry out  $P_i$ 's virtual broadcast over  $\text{BC}_i$ .

We can now map the set of real honest parties to sets of virtual honest parties. For instance, for  $H = \{P_1, P_2\}$ , the virtual parties in  $H'_1 = \{P_1, \dots, P_{\frac{2n}{3}}\}$  are honest. Similarly, for  $H = \{P_1, P_3\}$  and  $H = \{P_2, P_3\}$  we have virtual honest sets  $H'_2$  and  $H'_3$ , respectively. By the initial assumptions, in particular the one in Equation 2, it thus follows that  $\mathbb{P}(\Pi'(\text{AUTH}, \text{BC}_2, \text{BC}_3)(H, \vec{x}_H) \subseteq \text{BC}_1(H, \vec{x}_H)) > 1 - \epsilon$  for any  $H$  of size two. But this contradicts Corollary 16. ◀

In a similar fashion, one can prove the following statement for the  $n$ -party case.

► **Lemma 30.**  $\{\text{AUTH}\} \not\stackrel{\frac{1}{4}}{\rightarrow} \text{XC}_{1,2}$ .

Also, using almost the same arguments, we can prove the analogue of Lemma 20.

► **Lemma 31.** *For all  $i \neq j \in [n]$   $\{\text{AUTH}, \text{BC}_i\} \rightarrow \text{XC}_{i,j}$ .*

Finally, we can also restate Lemma 21 for the  $n$ -party case. Like the previous two lemmata, the proof proceeds in a similar fashion as the proof for the three-party case.

► **Lemma 32.**  $\{\text{XC}_{1,2}, \text{XC}_{1,3}, \text{XC}_{2,3}, \text{AUTH}\} \not\stackrel{\frac{1}{3}}{\rightarrow} \text{BC}_1$ .

## 6 Conclusion and Outlook

In this work, we showed strong separation results between broadcast and authenticated channels. In particular, we showed that weak broadcast admits a strong separation from broadcast. In order to derive these separations, we generalized known techniques for proving impossibility to cover also probabilistic constructions. We believe that the formal techniques and the framework that we introduced here will prove useful to future efforts in proving similar results. We also initiated the natural study of *asymmetric consistency primitives*, in which a (strict) subset of the parties has input and every party receives output. Although both broadcast and weak broadcast are examples of such primitives, our work is the first to consider primitives in which the subset of parties with input is not a singleton set. We show that for the example of the XOR-cast, this type of consistency primitive falls into a previously undiscovered intermediate layer between authenticated channels and broadcast. As such, we believe that our work opens up several interesting lines of future research. In regards to further extending the scope of impossibility results, it would be interesting to see whether our techniques for probabilistic constructions can also be used to derive stronger bounds in settings with more complicated setup such as [4, 1]. Another interesting direction for future research would be a closer study of asymmetric consistency primitives in the above sense. A first question in this area would be to see if the hierarchy of three-party specifications considered in this work has an even deeper structure than outlined here, or, more generally, to classify all such specifications. A second immediate question would be to investigate how the picture changes when we consider primitives with more than three parties or when switching to stronger models of corruption, such as the *general adversary model* [7, 8, 16] (as opposed to the threshold setting we considered here). Conceptually, it would also be worthwhile to derive connections between such results and the field of information theoretic MPC.

---

### References

- 1 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, jul 2005.
- 2 Michael J. Fischer, Nancy A. Lynch, and Michael Merritt. Easy impossibility proofs for distributed consensus problems. In Michael A. Malcolm and H. Raymond Strong, editors, *4th ACM Symposium Annual on Principles of Distributed Computing*, pages 59–70, Minaki, Ontario, Canada, aug 5–7, 1985. Association for Computing Machinery.
- 3 Matthias Fitzi, Daniel Gottesman, Martin Hirt, Thomas Holenstein, and Adam Smith. Detectable byzantine agreement secure against faulty majorities. In Aleta Ricciardi, editor, *21st ACM Symposium Annual on Principles of Distributed Computing*, pages 118–126, Monterey, California, USA, jul 21–24, 2002. Association for Computing Machinery.
- 4 Matthias Fitzi and Ueli M. Maurer. From partial consistency to global broadcast. In *32nd Annual ACM Symposium on Theory of Computing*, pages 494–503, Portland, Oregon, USA, may 21–23, 2000. ACM Press.
- 5 Ronald L. Graham and Andrew Chi-Chih Yao. On the improbability of reaching byzantine agreements (preliminary version). In *21st Annual ACM Symposium on Theory of Computing*, pages 467–478, Seattle, Washington, USA, may 15–17, 1989. ACM Press.
- 6 Martin Hirt, Ueli Maurer, and Pavel Raykov. Broadcast amplification. In Yehuda Lindell, editor, *TCC 2014: 11th Theory of Cryptography Conference*, volume 8349 of *Lecture Notes in Computer Science*, pages 419–439, San Diego, CA, USA, feb 24–26, 2014. Springer, Berlin, Germany. doi:10.1007/978-3-642-54242-8\_18.



- 7 Martin Hirt and Ueli M. Maurer. Player simulation and general adversary structures in perfect multiparty computation. *Journal of Cryptology*, 13(1):31–60, 2000. Extended abstract in *Proc. 16th of ACM PODC '97*.
- 8 Martin Hirt and Daniel Tschudi. Efficient general-adversary multi-party computation. In Kazue Sako and Palash Sarkar, editors, *Advances in Cryptology – ASIACRYPT 2013, Part II*, volume 8270 of *Lecture Notes in Computer Science*, pages 181–200, Bengalore, India, dec 1–5, 2013. Springer, Berlin, Germany. doi:10.1007/978-3-642-42045-0\_10.
- 9 Alexander Jaffe, Thomas Moscibroda, and Siddhartha Sen. On the price of equivocation in byzantine agreement. In Darek Kowalski and Alessandro Panconesi, editors, *31st ACM Symposium Annual on Principles of Distributed Computing*, pages 309–318, Funchal, Madeira, Portugal, jul 16–18, 2012. Association for Computing Machinery.
- 10 Anna Rochelle Karlin and Andrew Chi-Chih Yao. Probabilistic lower bounds for the byzantine generals problem. unpublished manuscript, 1984.
- 11 Leslie Lamport. The weak byzantine generals problem. *Journal of the ACM*, 30(3):668–676, jul 1983.
- 12 Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, jul 1982.
- 13 Julian Loss, Ueli Maurer, and Daniel Tschudi. Hierarchy of three-party consistency specifications. In *2016 IEEE International Symposium on Information Theory (ISIT)*, pages 3048–3052. IEEE, 2016.
- 14 Ueli Maurer. Towards a theory of consistency primitives. In Rachid Guerraoui, editor, *International Symposium on Distributed Computing — DISC 2004*, volume 3274 of *Lecture Notes in Computer Science*, pages 379–389. Springer, Berlin, Germany, 2004.
- 15 D. V. S. Ravikant, Venkitasubramaniam Muthuramakrishnan, V. Srikanth, K. Srinathan, and C. Pandu Rangan. On byzantine agreement over (2,3)-uniform hypergraphs. In Rachid Guerraoui, editor, *International Symposium on Distributed Computing — DISC 2004*, volume 3274 of *Lecture Notes in Computer Science*, pages 450–464. Springer, Berlin, Germany, Oct 2004.
- 16 Pavel Raykov. Broadcast from minicast secure against general adversaries. In Magnús M. Halldórsson, Kazuo Iwama, Naoki Kobayashi, and Bettina Speckmann, editors, *ICALP 2015: 42nd International Colloquium on Automata, Languages and Programming, Part II*, volume 9135 of *Lecture Notes in Computer Science*, pages 701–712, Kyoto, Japan, jul 6–10, 2015. Springer, Berlin, Germany. doi:10.1007/978-3-662-47666-6\_56.



# Broadcast and Minimum Spanning Tree with $o(m)$ Messages in the Asynchronous CONGEST Model

Ali Mashreghi<sup>1</sup>

Department of Computer Science, University of Victoria, BC, Canada  
ali.mashreghi87@gmail.com

Valerie King<sup>2</sup>

Department of Computer Science, University of Victoria, BC, Canada  
val@uvic.ca

---

## Abstract

We provide the first asynchronous distributed algorithms to compute broadcast and minimum spanning tree with  $o(m)$  bits of communication, in a sufficiently dense graph with  $n$  nodes and  $m$  edges. For decades, it was believed that  $\Omega(m)$  bits of communication are required for any algorithm that constructs a broadcast tree. In 2015, King, Kutten and Thorup showed that in the KT1 model where nodes have initial knowledge of their neighbors' identities it is possible to construct MST in  $\tilde{O}(n)$  messages in the synchronous CONGEST model. In the CONGEST model messages are of size  $O(\log n)$ . However, no algorithm with  $o(m)$  messages were known for the asynchronous case. Here, we provide an algorithm that uses  $O(n^{3/2} \log^{3/2} n)$  messages to find MST in the asynchronous CONGEST model. Our algorithm is randomized Monte Carlo and outputs MST with high probability. We will provide an algorithm for computing a spanning tree with  $O(n^{3/2} \log^{3/2} n)$  messages. Given a spanning tree, we can compute MST with  $\tilde{O}(n)$  messages.

**2012 ACM Subject Classification** Computing methodologies → Distributed algorithms, Mathematics of computing → Graph algorithms

**Keywords and phrases** Distributed Computing, Minimum Spanning Tree, Broadcast Tree

**Digital Object Identifier** 10.4230/LIPIcs.DISC.2018.37

**Related Version** A full version of the paper is available at [22], <https://arxiv.org/abs/1806.04328>.

---

<sup>1</sup> Funded with an NSERC grant.

<sup>2</sup> Funded with an NSERC grant.



## 1 Introduction

We consider a distributed network as an undirected graph with  $n$  nodes and  $m$  edges, and the problem of finding a spanning tree and a minimum spanning tree (MST) with efficient communication. That is, we require that every node in the graph learns exactly the subset of its incident edges which are in the spanning tree or MST, resp. A spanning tree enables a message to be broadcast from one node to all other nodes with only  $n - 1$  edge traversals. In a sensor or ad hoc network where the weight of a link between nodes reflects the amount of energy required to transmit a message along the link [19], the minimum spanning tree (MST) provides an energy efficient means of broadcasting. The problem of finding a spanning tree in a network has been studied for more than three decades, since it is the building block of many other fundamental problems such as *counting*, *leader election*, and *deadlock resolution* [3].

A spanning tree can be constructed by a simple breadth-first search from a single node using  $m$  bits of communication. The tightness of this communication bound was a “folk theorem”, according to Awerbuch, Goldreich, Peleg and Vainish [4]. Their 1990 paper defined the KT1 model where nodes have unique IDs and know only their neighbors. It showed, for a limited class of algorithms, a lower bound of  $\Omega(m)$  messages in a synchronous KT1 network. In 2015, Kutten et al. [19] proved a lower bound for general randomized algorithms with  $O(\log n)$  bit messages, in the KT0 model, where nodes do not know their neighbors. In 2015, King, Kutten, and Thorup gave the first distributed algorithm (“KKT”) with  $o(m)$  communication to build a broadcast tree and MST in the KT1 model. They devised Monte Carlo algorithms in the synchronous KT1 model with  $\tilde{O}(n)$  communication [18]. This paper and a followup paper [21] left open the problem of whether a  $o(m)$  bit communication algorithm in the asynchronous model was possible, for either the spanning tree or MST problem, when nodes know their neighbors’ IDs.

In an asynchronous network, there is no global clock. All processors may wake up at the start and send messages, but further actions by a node are event-driven, i.e., in response to messages received. The pioneer work of Gallager, Humblet, and Spira [14] (“GHS”) presented an asynchronous protocol for finding the MST in the CONGEST model, where messages are of size  $O(\log n)$ . GHS requires  $O(m + n \log n)$  messages and  $O(n \log n)$  time if all nodes are awakened simultaneously. Afterwards, researchers worked on improving the time complexity of MST algorithms in the CONGEST model but the message complexity remained  $\Omega(m)$ . In this paper, we provide the first algorithm in the KT1 model which uses  $o(m)$  bits of communication for finding a spanning tree in an asynchronous network, specifically we show the following:

► **Theorem 1.** *Given any network of  $n$  nodes where all nodes awake at the start, a spanning tree and a minimum spanning tree can be built with  $O(n^{3/2} \log^{3/2} n)$  messages in the asynchronous KT1 CONGEST model, with high probability.*

### 1.1 Techniques

Many distributed algorithms to find an MST use the Boruvka method: Starting from the set of isolated nodes, a forest of edge disjoint rooted trees which are subtrees of the MST are maintained. The algorithm runs in phases: In a phase, in parallel, each tree  $A$  finds a minimum weight *outgoing edge*, that is, one with exactly one endpoint in  $A$  and its other endpoint in some other tree  $B$ . Then the outgoing edge is inserted to create the “merged” tree containing the nodes of  $A$  and  $B$ . In what seems an inherently synchronous process, every tree (or a constant fraction of the trees) participates in some merge, the number of

trees is reduced by a constant factor per phase, and  $O(\log n)$  phases suffice to form a single tree. [14, 3, 18, 21].

The KKT paper introduced procedures *FindAny* and *FindMin* which can find any or the minimum outgoing edge leaving the tree, respectively. These require  $O(|T|)$  messages and  $\tilde{O}(|T|)$ , resp., where  $|T|$  is the number of nodes in the tree  $T$  or a total of  $\tilde{O}(n)$  per phase. As this is done synchronously in KKT, only  $O(\log n)$  phases are needed, for a total number of only  $O(n \log n)$  messages to build a spanning tree.

While *FindAny* and *FindMin* are asynchronous procedures, the Boruvka approach of [18] does not seem to work in an asynchronous model with  $o(m)$  messages, as it does not seem possible to prevent only one tree from growing, one node at a time, while the other nodes are delayed, for a cost of  $O(n^2)$  messages. The asynchronous GHS also uses  $O(\log n)$  phases to merge trees in parallel, but it is able to synchronize the growth of the trees by assigning a *rank* to each tree. A tree which finds a minimum outgoing edge waits to merge until the tree it is merging with is of equal or higher rank. The GHS algorithm subtly avoids traversing the whole tree until a minimum weight outgoing edge to an appropriately ranked tree is found. This method seems to require communication over all edges in the worst case.

Asynchrony precludes approaches that can be used in the synchronous model. For example, in the synchronous model, if nodes of low degree send messages to all their neighbors, in one round all nodes learn which of their neighbors do not have low degree, and therefore they can construct the subgraph of higher degree nodes. In the asynchronous model, a node, not hearing from its neighbor, does not know when to conclude that its neighbor is of higher degree.

The technique for building a spanning tree in our paper is very different from the technique in [18] or [14]. We grow one tree  $T$  rooted at one preselected *leader* in phases. (If there is no preselected leader, then this may be done from a small number of randomly self-selected nodes.) Initially, each node selects itself with probability  $1/\sqrt{n \log n}$  as a *star node*. (We use  $\log n$  to denote  $\log_2 n$ .) This technique is inspired from [10], and provides a useful property that every node whose degree is at least  $\sqrt{n} \log^{3/2} n$  is adjacent to a star node with high probability. Initially, star nodes (and low-degree nodes) send out messages to all of their neighbors. Each high-degree node which joins  $T$  waits until it hears from a star node and then invites it to join  $T$ . In addition, when low-degree and star nodes join  $T$ , they invite their neighbors to link to  $T$  via their incident edges. Therefore, with high probability, the following invariant for  $T$  is maintained as  $T$  grows:

**Invariant:**  $T$  includes all neighbors of any star or low-degree node in  $T$ , as well. Each high-degree node in  $T$  is adjacent to a star node in  $T$ .

The challenge is for high-degree nodes in  $T$  to find neighbors outside  $T$ . If in each phase, an outgoing edge from a high-degree node in  $T$  to a high-degree node  $x$  (not in  $T$ ) is found and  $x$  is invited to join  $T$ , then  $x$ 's adjacent star node (which must lie outside  $T$  by the Invariant) is also found and invited to join. As the number of star nodes is  $O(\sqrt{n}/\log^{1/2} n)$ , this number also bounds the number of such phases. The difficulty is that there is no obvious way to find an outgoing edge to a high degree node because, as mentioned above, in an asynchronous network, a high degree node has no apparent way to determine if its neighbor has high degree without receiving a message from its neighbor.

Instead, we relax our requirement for a phase. With each phase either **(A)** A high-degree node (and star node) is added to  $T$  or **(B)**  $T$  is expanded so that the number of outgoing edges to low-degree nodes is reduced by a constant factor. As there are no more than  $O(\sqrt{n}/\log^{1/2} n)$  phases of type **A** and no more than  $O(\log n)$  phases of type **B** between each type **A** phase, there are a total of  $O(\sqrt{n} \log^{1/2} n)$  phases before all nodes are in  $T$ . The

key idea for implementing a phase of type **B** is that the tree  $T$  waits until its nodes have heard enough messages passed by low-degree nodes over outgoing edges before initiating an expansion. The efficient implementation of a phase, which uses only  $O(n \log n)$  messages, requires a number of tools which are described in the preliminaries section.

Once a spanning tree is built, we use it as a communication network while finding the MST. This enables us to “synchronize” a modified GHS which uses *FindMin* for finding minimum outgoing edges, using a total of  $\tilde{O}(n)$  messages.

**Note:** If we do not assume the existence of a pre-selected leader, or the graph is not connected, then a variant of the algorithm described in the arxiv version [22] is needed.

## 1.2 Related work

The Awerbuch, Goldreich, Peleg and Vainish [4] lower bound on the number of messages holds only for (randomized) algorithms where messages may contain a constant number of IDs, and IDs are processed by comparison only and for general deterministic algorithms, where ID’s are drawn from a very large size universe.

Time to build an MST in the CONGEST model has been explored in several papers. Algorithms include, in the asynchronous KT0 model, [14, 3, 13, 26], and in the synchronous KT0 model, [20, 15, 7, 17]. Recently, in the synchronous KT0 model, Pandurangan gave a [23]  $\tilde{O}(D + \sqrt{n})$  time and  $\tilde{O}(m)$  message randomized algorithm, which Elkin improved by logarithmic factors with a deterministic algorithm [11]. The time complexity to compute spanning tree in the algorithm of [18] is  $O(n \log n)$  which was improved to  $O(n)$  in [21].

Lower bounds on time for approximating the minimum spanning tree has been proved in the synchronous KT0 model In [8, 25]. Kutten et al. [19] show an  $\Omega(m)$  lower bound on message complexity for randomized general algorithms in the KT0 model.

*FindAny* and *FindMin* which appear in the KKT algorithms build on ideas for sequential dynamic connectivity in [16]. A sequential dynamic *ApproxCut* also appeared in that paper [16]. Solutions to the sequential linear sketching problem for connectivity [1] share similar techniques but require a more complex step to verify when a candidate edge name is an actual edge in the graph, as the edges names are no longer accessible once the sketch is made (See Subsection 2.3).

The *threshold detection* problem was introduced by Emek and Korman [12]. It assumes that there is a rooted spanning tree  $T$  where events arrive online at  $T$ ’s nodes. Given some threshold  $k$ , a termination signal is broadcast by the root if and only if the number of events exceeds  $k$ . We use a naive solution of a simple version of the problem here.

A synchronizer, introduced by Awerbuch [2] and studied in [6, 5, 24, 9], is a general technique for simulating a synchronous algorithm on an asynchronous network using communications along a spanning tree. To do this, the spanning tree must be built first. Using a general synchronizer imposes an overhead of messages that affect *every single step* of the synchronous algorithm that one wants to simulate, and would require more communication than our special purpose method of using our spanning tree to synchronize the modified GHS.

## 1.3 Organization

Section 2 describes the model. Section 3 gives the spanning tree algorithm for the case of a connected network and a single known leader. Finally, Section 4 provides the MST algorithm. (Due to lack of space the algorithm for computing a minimum spanning *forest* in disconnected graphs or minimum spanning tree for dealing with the case of no pre-selected leader is available on the arxiv version [22]. This variant of the algorithm has the same message complexity.)

## 2 Preliminaries

### 2.1 Model

Let  $c \geq 1$  be any constant. The communications network is the undirected graph  $G = (V, E)$  over which a spanning tree or MST will be found. Edge weights are integers in  $[1, n^c]$ . IDs are assigned uniquely by the adversary from  $[1, n^c]$ . All nodes have knowledge of  $c$  and  $n$  which is an upper bound on  $|V|$  (number of nodes in the network) within a *constant* factor. All nodes know their own ID along with the ID of their neighbors (KT1 model) and the weights of their incident edges. Nodes have no other information about the network. e.g., they do not know  $|E|$  or the maximum degree of the nodes in the network. Nodes can only send direct messages to the nodes that are adjacent to them in the network. If the edge weights are not unique they can be made unique by appending the ID of the endpoints to its weight, so that the MST is unique. Nodes can only send direct messages to the nodes that are adjacent to them in the network. Our algorithm is described in the CONGEST model in which each message has size  $O(\log n)$ . Its time is trivially bounded by the total number of messages. The KT1 CONGEST model has been referred to as the “standard model” [4].

Message cost is the sum over all edges of the number of messages sent over each edge during the execution of the algorithm. If a message is sent it is eventually received, but the adversary controls the length of the delays and there is no guarantee that messages sent by the same node will be received in the order they are sent. There is no global clock. All nodes awake at the start of the protocol simultaneously. After awaking and possibly sending its initial messages, a processor acts only in response to receiving messages.

We say a network “finds” a subgraph if at the end of the distributed algorithm, every node knows exactly which of its incident edges in the network are part of the subgraph. The algorithm here is Monte Carlo, in that it succeeds with probability  $1 - n^{-c''}$  for any constant  $c''$  (“w.h.p.”).

We initially assume there is a special node (called *leader*) at the start and the graph is connected. These assumptions are dropped in the algorithm we provide for disconnected graphs in the full version of the paper.

### 2.2 Definitions and Subroutines

$T$  is initially a tree containing only the leader node. Thereafter,  $T$  is a tree rooted at the leader node. We use the term *outgoing edge* from  $T$  to mean an edge with exactly one endpoint in  $T$ . An outgoing edge is described as if it is directed; it is *from* a node in  $T$  and *to* a node not in  $T$  (the “external” endpoint).

The algorithm uses the following subroutines and definitions:

- *Broadcast*( $M$ ): Procedure whereby the node  $v$  in  $T$  sends message  $M$  to its children and its children broadcast to their subtrees.
- *Expand*: A procedure for adding nodes to  $T$  and preserving the Invariant after doing so.
- *FindAny*: Returns to the leader an outgoing edge chosen uniformly at random with probability  $1/16$ , or else it returns  $\emptyset$ . The leader then broadcasts the result. *FindAny* requires  $O(n)$  messages. We specify *FindAny*( $E'$ ) when we mean that the outgoing edge must be an outgoing edge in a particular subset  $E' \subseteq E$ .
- *FindMin*: is similarly defined except the edge is the (unique) minimum cost outgoing edge. This is used only in the minimum spanning tree algorithm. *FindMin* requires  $O(n \log^2 n / \log \log n)$  messages.



- *ApproxCut*: A function which w.h.p. returns an estimate in  $[k/32, k]$  where  $k$  is the number of outgoing edges from  $T$  and  $k > c \log n$  for  $c$  a constant. It requires  $O(n \log n)$  messages.  
*FindAny* and *FindMin* are described in [18] (The *FindAny* we use is called *FindAny-C* there.) *FindAny-C* was used to find *any* outgoing edge in the previous paper. It is not hard to see that the edge found is a random edge from the set of outgoing edges; we use that fact here. The relationships among *FindAny*, *FindMin* and *ApproxCut* below are described in the next subsection.
- *Found<sub>L</sub>(v)*, *Found<sub>O</sub>(v)*: Two lists of edges incident to node  $v$ , over which  $v$  will send invitations to join  $T$  the next time  $v$  participates in *Expand*. After this, the list is emptied. Edges are added to *Found<sub>L</sub>(v)* when  $v$  receives  $\langle \text{Low-degree} \rangle$  message or the edge is found by the leader by sampling and its external endpoint is low-degree. Otherwise, an edge is added to *Found<sub>O</sub>(v)* when  $v$  receives a  $\langle \text{Star} \rangle$  message over an edge or if the edge is found by the leader by sampling and its external endpoint is high-degree. Note that star nodes that are low-degree send both  $\langle \text{Low-degree} \rangle$  and  $\langle \text{Star} \rangle$ . This may cause an edge to be in both lists which is handled properly in the algorithm.
- *T-neighbor(v)*: A list of neighbors of  $v$  in  $T$ . This list, except perhaps during the execution of *Expand*, includes all low-degree neighbors of  $v$  in  $T$ . This list is used to exclude from *Found<sub>L</sub>(v)* any non-outgoing edges.
- *ThresholdDetection(k)*: A procedure which is initiated by the leader of  $T$ . The nodes in  $T$  experience no more than  $k < n^2$  events w.h.p. The leader is informed w.h.p. when the number of events experienced by the nodes in  $T$  reaches the threshold  $k/4$ . Here, an event is the receipt of  $\langle \text{Low-degree} \rangle$  over an outgoing edge. Following the completion of *Expand*, all edges  $(u, v)$  in *Found<sub>L</sub>(u)* are events if  $v \notin T\text{-neighbor}(u)$ .  $O(|T| \log n)$  messages suffice.

### 2.3 Implementation of *FindAny*, *FindMin* and *ApproxCut*

We briefly review *FindAny* in [18] and explain its connection with *ApproxCut*. The key insight is that an outgoing edge is incident to exactly one endpoint in  $T$  while other edges are incident to zero or two endpoints. If there were exactly one outgoing edge, the parity of the sum of all degrees in  $T$  would be 1, and the parity of bit-wise XOR of the binary representation of the names of all incident edges would be the name of the one outgoing edge.

To deal with possibility of more than one outgoing edge, the leader creates an efficient means of sampling edges at different rates: Let  $l = \lceil 2 \log n \rceil$ . The leader selects and broadcasts one pairwise independent hash function  $h : [\text{edge\_names}] \rightarrow [1, 2^l]$ , where *edge\_name* of an edge is a unique binary string computable by both its endpoints, e.g.,  $\{x, y\} = x \cdot y$  for  $x < y$ . Each node  $y$  forms the vector  $\vec{h}(y)$  whose  $i^{\text{th}}$  bit is the parity of its incident edges that hash to  $[0, 2^i]$ ,  $i = 0, \dots, l$ . Starting with the leaves, a node in  $T$  computes the bitwise XOR of the vectors from its children and itself and then passes this up the tree, until the leader has computed  $\vec{b} = \text{XOR}_{y \in T} \vec{h}(y)$ . The key insight implies that for each index  $i$ ,  $\vec{b}_i$  equals the parity of just the outgoing edges mapped to  $[0, 2^i]$ . Let *min* be the smallest index  $i$  s.t.  $\vec{b}_i = 1$ . With constant probability, exactly one edge of the outgoing edges has been mapped to  $[1, 2^{\text{min}}]$ . The leader broadcasts *min*. Nodes send back up the XOR of the *edge\_names* of incident edges which are mapped by  $h$  to this range. If exactly one outgoing edge has been indeed mapped to that range, the leader will find it by again determining the XOR of the *edge\_names* sent up. One more broadcast from the leader can be used to verify that this edge exists and is incident to exactly one node in  $T$ .

Since each edge has the same probability of failing in  $[0, 2^{min}]$ , this procedure gives a randomly selected edge. Note also that the leader can instruct the nodes to exclude certain edges from the XOR, say incident edges of weight greater than some  $w$ . In this way the leader can binary search for the minimal weight outgoing edge to carry out *FindMin*. Similarly, the leader can select random edges without replacement.

Observe that if the number of outgoing edges is close to  $2^j$ , we'd expect  $min$  to be  $l - j$  with constant probability. Here we introduce distributed asynchronous *ApproxCut* which uses the sampling technique from *FindMin* but repeats it  $O(\log n)$  times with  $O(\log n)$  randomly chosen hash functions. Let  $min\_sum$  be the minimum  $i$  for which the sum of  $b_i$ 's exceeds  $c \log n$  for some constant  $c$ . We show  $2^{min\_sum}$  approximates the number of outgoing edges within a constant factor from the actual number. *ApproxCut* pseudocode is given in Algorithm 5.

We show:

► **Lemma 2.** *With probability  $1 - 1/n^c$ , ApproxCut returns an estimate in  $[k/32, k]$  where  $k$  is the number of outgoing edges and  $k > c' \log n$ ,  $c'$  a constant depending on  $c$ . It uses  $O(n \log n)$  messages.*

The proof is given in Section 3.2.

### 3 Asynchronous ST construction with $o(m)$ messages

In this section we explain how to construct a spanning tree when there is a preselected leader and the graph is connected.

Initially, each node selects itself with probability  $1/\sqrt{n \log n}$  as a *star node*. Low-degree and star nodes initially send out  $\langle Low-degree \rangle$  and  $\langle Star \rangle$  messages to all of their neighbors, respectively. (We will be using the  $\langle M \rangle$  notation to show a message with content  $M$ .) A low-degree node which is a star node sends both types of messages. At any point during the algorithm, if a node  $v$  receives a  $\langle Low-degree \rangle$  or  $\langle Star \rangle$  message through some edge  $e$ , it adds  $e$  to  $Found_L(v)$  or  $Found_O(v)$  resp.

The algorithm FindST-Leader runs in phases. Each phase has three parts: **1) Expansion** of  $T$  over found edges since the previous phase and restoration of the Invariant; **2) Search** for an outgoing edge to a high-degree node; **3) Wait** until messages to nodes in  $T$  have been received over a *constant fraction of the outgoing edges* whose external endpoint is low-degree.

**1) Expansion.** Each phase is started with *Expand*. *Expand* adds to  $T$  any nodes which are external endpoints of outgoing edges placed on a *Found* list of any node in  $T$  since the last time that node executed *Expand*. In addition, it restores the Invariant for  $T$ .

**Implementation.** *Expand* is initiated by the leader and broadcast down the tree. When a node  $v$  receives  $\langle Expand \rangle$  message for the first time (it is not in  $T$ ), it joins  $T$  and makes the sender its parent. If it is a high-degree node and is not a star, it has to wait until it receives a  $\langle Star \rangle$  message over some edge  $e$ , and then adds  $e$  to  $Found_O(v)$ . It then forwards  $\langle Expand \rangle$  over the edges in  $Found_L(v)$  or  $Found_O(v)$  and empties these lists. Otherwise, if it is a low-degree node or a star node, it forwards  $\langle Expand \rangle$  to *all* of its neighbors.

On the other hand, if  $v$  is already in  $T$ , it forwards  $\langle Expand \rangle$  message to its children in  $T$  and along any edges in  $Found_L(v)$  or  $Found_O(v)$ , i.e. outgoing edges which were “found” since the previous phase, and empties these lists. All  $\langle Expand \rangle$  requests received by  $v$  are answered, and their sender is added to  $T\text{-neighbor}(v)$ . The procedure ends in a bottom-up

way and ensures that each node has heard from all the nodes it sent  $\langle \text{Expand} \rangle$  requests to before it contacts its parent.

Let  $T^i$  denote  $T$  after the execution of  $\text{Expand}$  in phase  $i$ . Initially  $T^0$  consists of the leader node and as its Found lists contain all its neighbors, after the first execution of  $\text{Expand}$ , if the leader is high-degree,  $T_1$  satisfies the invariant. An easy inductive argument on  $T^i$  shows:

► **Observation 1.** *For all  $i > 0$ , upon completion of  $\text{Expand}$ , all the nodes reachable by edges in the Found lists of any node in  $T^{i-1}$  are in  $T^i$ , and for all  $v \in T$ ,  $T\text{-neighbor}(v)$  contains all the low-degree neighbors of  $v$  in  $T$ .*

$\text{Expand}$  is called in line 6 of the main algorithm 1. The pseudocode is given in  $\text{Expand}$  Algorithm 1.

**2) Search for an outgoing edge to a high degree node.** A sampling of the outgoing edges without replacement is done using  $\text{FindAny}$  multiple times. The sampling either (1) finds an outgoing edge to a high degree node, or (2) finds all outgoing edges, or (3) determines w.h.p. that at least half the outgoing edges are to low-degree nodes and there are at least  $2c \log n$  such edges. If the first two cases occur, the phase ends.

**Implementation.** Endpoints of sampled edges in  $T$  communicate over the outgoing edge to determine if the external endpoint is high-degree. If at least one is, that edge is added to the  $\text{Found}_O$  list of its endpoint in  $T$  and the phase ends. If there are fewer than  $2 \log n$  outgoing edges, all these edges are added to  $\text{Found}_O$  and the phase ends. If there are no outgoing edges, the algorithm ends. If all  $2 \log n$  edges go to low-degree nodes, then the phase continues with Step 3) below. This is implemented in the **while** loop of  $\text{FindST-Leader}$ .

Throughout this section we will be using the following fact from Chernoff bounds: Assume  $X_1, X_2, \dots, X_T$  are independent Bernoulli trials where each trial's outcome is 1 with probability  $0 < p < 1$ . Chernoff bounds imply that given constants  $c, c_1 > 1$  and  $c_2 < 1$  there is a constant  $c'$  such that if there are  $T \geq c' \log n$  independent trials, then  $\Pr(X > c_1 \cdot E[X]) < 1/n^c$  and  $\Pr(X < c_2 \cdot E[X]) < 1/n^c$ , where  $X$  is sum of the  $X_1, \dots, X_T$ .

We show:

► **Lemma 3.** *After Search, at least one of the following must be true with probability  $1 - 1/n^{c'}$ , where  $c'$  is a constant depending on  $c$ : 1) there are fewer than  $2c \log n$  outgoing edges and the leader learns them all; 2) an outgoing edge to a high-degree node is found, or 3) there are at least  $2c \log n$  outgoing edges and at least half the outgoing edges are to low-degree nodes.*

**Proof.** Each  $\text{FindAny}$  has a probability of  $1/16$  of returning an outgoing edge and if it returns an edge, it is always outgoing. After  $48c \log n$  repetitions without replacement, the expected number of edges returned is  $3c \log n$ . As these trials are independent, Chernoff bounds imply that at least  $2/3$  of trials will be successful with probability at least  $1 - 1/n^c$ , i.e.,  $2c \log n$  edges are returned if there are that many, and if there are fewer, all will be returned.

The edges are picked uniformly at random by independent repetitions of  $\text{FindAny}$ . If more than half the outgoing edges are to high-degree nodes, the probability that all edges returned are to low-degree nodes is  $1/2^{2c \log n} < 1/n^{2c}$ . ◀

**3) Wait to hear from outgoing edges to low-degree external nodes.** This step forces the leader to wait until  $T$  has been contacted over a constant fraction of the outgoing edges to (external) low-degree nodes. Note that we do not know how to give a good estimate on the number of low-degree nodes which are neighbors of  $T$ . Instead we count outgoing edges.

**Implementation.** This step occurs only if the  $2c \log n$  randomly sampled outgoing edges all go to low-degree nodes and therefore the number of outgoing edges to low-degree nodes is at least this number. In this case, the leader waits until  $T$  has been contacted through a constant fraction of these edges.

If this step occurs, then w.h.p., at least half the outgoing edges go to low-degree nodes. Let  $k$  be the number of outgoing edges;  $k \geq 2c \log n$ . The leader calls *ApproxCut* to return an estimate  $q \in [k/32, k]$  w.h.p. It follows that w.h.p. the number of outgoing edges to low-degree nodes is  $k/2$ . Let  $r = q/2$ . Then  $r \in [k/64, k/2]$ .

The nodes  $v \in T$  will eventually receive at least  $k/2$  messages over outgoing edges of the form  $\langle \text{Low-degree} \rangle$ . Note that these messages must have been received by  $v$  after  $v$  executed *Expand* and added to  $Found_L(v)$ , for otherwise, these would not be outgoing edges.

The leader initiates a *ThresholdDetection* procedure whereby there is an event for a node  $v$  for each outgoing edge  $v$  has received a  $\langle \text{Low-degree} \rangle$  message over since the last time  $v$  executed *Expand*. As the *ThresholdDetection* procedure is initiated after the leader finishes *Expand*, the  $T\text{-neighbor}(v)$  includes any low-degree neighbor of  $v$  that is in  $T$ . Using  $T\text{-neighbor}(v)$ ,  $v$  can determine which edges in  $Found_L(v)$  are outgoing.

Each event experienced by a node causes it to flip a coin with probability  $\min\{c \log n/r, 1\}$ . If the coin is heads, then a trigger message labelled with the phase number is sent up to the leader. The leader is triggered if it receives at least  $(c/2) \log n$  trigger messages for that phase. When the leader is triggered, it begins a new phase. Since there are  $k/2$  triggering events, the expected number of trigger messages eventually generated is  $(c \log n/r)(k/2) \geq c \log n$ . Chernoff bounds imply that at least  $(c/2) \log n$  trigger messages will be generated w.h.p. Alternatively, w.h.p., the number of trigger messages received by the leader will not exceed  $(c/2) \log n$  until at least  $k/8$  events have occurred, as this would imply twice the expected number. We can conclude that w.h.p. the leader will trigger the next phase after  $1/4$  of the outgoing edges to low-degree nodes have been found.

► **Lemma 4.** *When the leader receives  $(c/2) \log n$  messages with the current phase number, w.h.p., at least  $1/4$  of the outgoing edges to low-degree nodes have been added to  $Found_L$  lists.*

### 3.1 Proof of the main theorem

Here we prove Theorem 1 as it applies to computing the spanning tree of a connected network with a pre-selected leader.

► **Lemma 5.** *W.h.p., after each phase except perhaps the first, either (A) A high-degree node (and star node) is added to  $T$  or (B)  $T$  is expanded so that the number of outgoing edges to low-degree nodes is reduced by a  $1/4$  factor (or the algorithm terminates with a spanning tree).*

**Proof.** By Lemma 3 there are three possible results from the **Search** phase. If a sampled outgoing edge to a high-degree node is found, this edge will be added to the  $Found_O$  list of its endpoint in  $T$ . If the **Search** phase ends in fewer than  $2c \log n$  edges found and none of them are to high degree nodes, then w.h.p. these are all the outgoing edges to low-degree nodes, these edges will all be added to some  $Found_L$ . If there are no outgoing edges, the

algorithm terminates and a spanning tree has been found. If the third possible result occurs, then there are at least  $2 \log n$  outgoing edges, half of which go to low-degree nodes. By Lemma 4, the leader will trigger the next phase and it will do so after at least  $1/4$  of the outgoing edges to low-degree nodes have been added to  $Found_L$  lists.

By Observation 1, all the endpoints of the edges on the  $Found$  lists will be added to  $T$  in the next phase, and there is at least one such edge or there are no outgoing edges and the spanning tree has been found. When  $Expand$  is called in the next phase,  $T$  acquires a new high degree node in two possible ways, either because an outgoing edge on a  $Found$  list is to a high-degree node or because the recursive  $Expand$  on outgoing edges to low-degree edges eventually leads to an external high-degree node. In either case, by the Invariant,  $T$  will acquire a new star node as well as a high-degree node. Also by the Invariant, all outgoing edges must come from high-degree nodes. Therefore, if no high-degree nodes are added to  $T$  by  $Expand$ , then no new outgoing edges are added to  $T$ . On the other hand,  $1/4$  of the outgoing edges to low-degree nodes have become non-outgoing edges as their endpoints have been added to  $T$ . So we can conclude that the number of outgoing edges to low-degree nodes have been decreased by  $1/4$  factor. ◀

It is not hard to see:

► **Lemma 6.** *The number of phases is bounded by  $O(\sqrt{n} \log^{1/2} n)$ .*

**Proof.** By Lemma 5, every phase except perhaps the first, is of type A or type B. Chernoff bounds imply that w.h.p., the number of star nodes does not exceed its expected number ( $\sqrt{n}/\log^{1/2} n$ ) by more than a constant factor, hence there are no more than  $O(\sqrt{n}/\log^{1/2} n)$  phases of type A. Before and after each such phase, the number of outgoing edges to low-degree nodes is reduced by at least a fraction of  $1/4$ ; hence, there are no more than  $\log_{4/3} n^2 = O(\log n)$  phases of type B between phases of type A. ◀

Finally, we count the number of messages needed to compute the spanning tree.

► **Lemma 7.** *The overall number of messages is  $O(n^{3/2} \log^{3/2} n)$ .*

**Proof.** The initialization requires  $O(\sqrt{n} \log^{3/2} n)$  messages from  $O(n)$  low-degree nodes and  $O(n)$  messages from each of  $O(\sqrt{n}/\log^{1/2} n)$  stars. In each phase,  $Expand$  requires a number of messages which is linear in the size of  $T$  or  $O(n)$ , except that newly added low-degree and star nodes send to their neighbors when they are added to  $T$ , but this adds just a constant factor to the initialization cost.  $FindAny$  is repeated  $O(\log n)$  times for a total cost of  $O(n \log n)$  messages.  $ApproxCut$  requires the same number. The Threshold Detector requires only  $O(\log n)$  messages to be passed up  $T$  or  $O(n \log n)$  messages overall. Therefore, by Lemma 6 the number of messages over all phases is  $O(n \log^{3/2} n)$ . ◀

Theorem 1 for spanning trees in connected networks with a pre-selected leader follows from Lemmas 7 and 6.

### 3.2 Proof of *ApproxCut* Lemma

**Proof.** Let  $W$  be the set of the outgoing edges. For a fixed  $z$  and  $i$ , we have:

$$Pr(h_{z,i}(T) = 1) = Pr(\text{an odd number of edges in } W \text{ hash to } [2^i]) \geq$$

$$Pr(\exists! e \in W \text{ hashed to } [2^i]).$$

This probability is at least  $1/16$  for  $i = l - \lceil \log |W| \rceil - 2$  (Lemma 5 of [18]). Therefore, since  $X_j = \sum_{z=1}^{c \log n} h_{z,j}$  (from pseudocode),  $E[X_j] = \sum E[h_{z,j}] \geq c \log n / 16$ , where  $j = l - \lceil \log |W| \rceil - 2$ . Note that  $j = l - \lceil \log |W| \rceil - 2$  means that  $\frac{2^j}{2^{j+3}} < |W| < \frac{2^j}{2^{j+1}}$ . Consider  $j - 4$ . Since the probability of an edge being hashed to  $[2^{j-4}]$  is  $\frac{2^{j-4}}{2^l}$ , we have

$$Pr(h_{z,j-4}(T) = 1) \leq Pr(\exists e \in \text{Hashed to } [2^{j-4}]) = |W| \frac{2^{j-4}}{2^l} \leq \frac{1}{2^5} \leq \frac{1}{32}.$$

Thus,  $E[X_{j-4}] \leq c \log n / 32$ . Since an edge that is hashed to  $[2^{j-k}]$  (for  $k > 4$ ) is already hashed to  $[2^{j-4}]$ , we have:

$$\begin{aligned} Pr(h_{z,j-4}(T) = 1 \vee \dots \vee h_{z,0}(T) = 1) &\leq Pr(\exists e \in \text{Hashed to } [2^{j-4}] \text{ or } \dots \text{ or } [2^0]) = \\ Pr(\exists e \in \text{Hashed to } [2^{j-4}]) &= \frac{1}{32}. \end{aligned}$$

Let  $y_z$  be 1 if  $h_{z,j-4}(T) = 1 \vee \dots \vee h_{z,0}(T) = 1$ , and 0 otherwise. Also, let  $Y = \sum_{z=1}^{c \log n} y_z$ . We have  $E[Y] \leq c \log n / 32$ . Also, for any positive integer  $a$ ,

$$Pr(X_{j-4} > a \vee \dots \vee X_0 > a) \leq Pr(Y > a).$$

From Chernoff bounds:

$$Pr(X_j < (3/4)c \log n / 16) = Pr(X_j < (3/4)E[X_j]) < 1/n^{c'}$$

and,

$$\begin{aligned} Pr(X_{j-4} > (3/2)c \log n / 16 \vee \dots \vee X_0 > (3/2)c \log n / 16) &\leq Pr(Y > (3/2)c \log n / 16) = \\ Pr(Y > (3/2)c \log n / 32) &< Pr(Y > (3/2)E[Y]) < 1/n^{c'}. \end{aligned}$$

Therefore, by finding the smallest  $i$  (called *min* in pseudocode) for which  $X_i > (3/2)c \log n / 16$ , w.h.p. *min* is in  $[j - 3, j]$ . As a result,  $2|W| \leq 2^{l-\text{min}} \leq 64|W|$ . Therefore,  $|W|/32 \leq 2^{l-\text{min}}/64 \leq |W|$ .

Furthermore, broadcasting each of the  $O(\log n)$  hash functions and computing the corresponding vector takes  $O(n)$  messages; so, the lemma follows.  $\blacktriangleleft$

### 3.3 Pseudocode

---

**Algorithm 1** Initialization of the spanning tree algorithm.

---

- 1: **procedure** INITIALIZATION
  - 2:   Every node selects itself to be a *star* node with probability of  $1/\sqrt{n \log n}$ .
  - 3:   Nodes that have degree  $< \sqrt{n} \log^{3/2} n$  are *low-degree* nodes. Otherwise, they are *high-degree* nodes. (Note that they may also be star nodes at the same time.)
  - 4:   Star nodes send  $\langle \text{Star} \rangle$  messages to all of their neighbors.
  - 5:   Low-degree nodes send  $\langle \text{Low-degree} \rangle$  messages to all of their neighbors (even if they are star nodes too).
  - 6: **end procedure**
-

---

**Algorithm 2** Asynchronous protocol for the leader to find a spanning tree.

---

```

1: procedure FINDST-LEADER
2:   Leader initially adds all of its incident edges to its  $Found_L$  list. // By exception
   leader does not need to differentiate between  $Found_L$  and  $Found_O$ 
3:    $i \leftarrow 0$ 
4:   repeat (Phase  $i$ )
5:      $i \leftarrow i + 1$ .
6:     Leader calls  $Expand()$ . // Expansion
       // Search and Sampling:
7:      $counter \leftarrow 0, A \leftarrow \emptyset$ .
8:     while  $counter < 48c \log n$  do
9:        $FindAny(E \setminus A)$ .
10:      if  $FindAny$  is successful and finds an edge  $(u, v)$  ( $u \in T$  and  $v \notin T$ ) then
11:         $u$  sends a message to  $v$  to query  $v$ 's degree, and sends it to the leader.
12:         $u$  adds  $(u, v)$  to either  $Found_L(u)$  or  $Found_O(u)$  based on  $v$ 's degree.
13:      end if
14:       $counter \leftarrow counter + 1$ .
15:    end while
16:    if  $|A| = 0$  then
17:      terminate the algorithm as there are no outgoing edges.
18:    else if  $|A| < 2 \log n$  (few edges) or  $\exists(u, v) \in A$  s.t.  $v$  is high-degree then
19:      Leader starts a new phase to restore the Invariant.
20:    else (at least half of the outgoing edges are to low-degree nodes) // Wait:
21:       $r \leftarrow ApproxCut()/2$ .
22:      Leader calls  $ThresholdDetection(r)$ .
23:      Leader waits to trigger and then starts a new phase.
24:    end if
25:  until
26: end procedure

```

---

**Algorithm 3** Given  $r$  at phase  $i$ , this procedure detects when nodes in  $T$  receive at least  $r/4$   $\langle Low - degree \rangle$  messages over outgoing edges.  $c$  is a constant.

---

```

1: procedure THRESHOLDDETECTION
2:   Leader calls Broadcast( $\langle Send-trigger, r, i \rangle$ ).
3:   When a node  $u \in T$  receives  $\langle Send-trigger, r, i \rangle$ , it first participates in the broadcast.
   Then, for every event, i.e. every edge  $(u, v) \in Found(u)_L$  such that  $v \notin T-neighbor(u)$ ,
    $u$  sends to its parent a  $\langle Trigger, i \rangle$  message with probability of  $c \log n / r$ .
4:   A node that receives  $\langle Trigger, i \rangle$  from a child keeps sending up the message until it
   reaches the leader. If a node receives an  $\langle Expand \rangle$  before it sends up a  $\langle Trigger, i \rangle$ , it
   discards the  $\langle Trigger, i \rangle$  messages as an Expand has already been triggered.
5:   Once the leader receives at least  $c \log n / 2$   $\langle Trigger, i \rangle$  messages, the procedure terminates
   and the control is returned to the calling procedure.
6: end procedure

```

---



---

**Algorithm 4** Leader initiates Expand by sending  $\langle Expand \rangle$  to all of its children. If this is the first Expand, leader sends to all of its neighbors. Here,  $x$  is any non-leader node.

---

```

1: procedure EXPAND
2:   When node  $x$  receives an  $\langle Expand \rangle$  message over an edge  $(x, y)$ :
3:    $x$  adds  $y$  to  $T\text{-neighbor}(x)$ .
4:   if  $x$  is not in  $T$  then
5:     The first node that  $x$  receives  $\langle Expand \rangle$  from becomes  $x$ 's parent. //  $x$  joins  $T$ 
6:     if  $x$  is a high-degree node and  $x$  is not a star node then
7:       It waits to receive a  $\langle Star \rangle$  over some edge  $e$ , then adds  $e$  to  $Found_O(x)$ .
8:       It forwards  $\langle Expand \rangle$  over edges in  $Found_L(x)$  and  $Found_O(x)$  (only once in
          case an edge is in both lists), then removes those edges from the Found lists.
9:     else ( $x$  is a low-degree or star node)
10:      It forwards the  $\langle Expand \rangle$  message to all of its neighbors.
11:    end if
12:  else ( $x$  is already in  $T$ )
13:    If the sender is not its parent, it sends back  $\langle Done\text{-by-reject} \rangle$ . Else, it forwards
       $\langle Expand \rangle$  to its children in  $T$ , over the edges in  $Found_L(x)$  and  $Found_O(x)$ ,
      then removes those edges from the Found lists.
14:  end if
  // Note that if  $x$  added more edges to its Found list after forward of
   $\langle Expand \rangle$ , the new edges will be dealt with in the next Expand.
15:  When a node receives  $\langle Done \rangle$  messages (either  $\langle Done\text{-by-accept} \rangle$  or  $\langle Done\text{-by-reject} \rangle$ )
  from all of the nodes it has sent to, it considers all nodes that have sent
   $\langle Done\text{-by-accept} \rangle$  as its children. Then, it sends up  $\langle Done\text{-by-accept} \rangle$  to its parent.
16:  The algorithm terminates when the leader receives  $\langle Done \rangle$  from all of its children.
17: end procedure

```

---

**Algorithm 5** Approximates the number of outgoing edges within a constant factor.  $c$  is a constant.

---

```

1: procedure APPROXCUT( $T$ )
2:   Leader broadcasts  $c \log n$  random 2-wise independent hash functions defined from
    $[1, n^{2c}] \rightarrow [2^l]$ .
3:   For node  $y$ , and hash function  $h_z$  vector  $\vec{h}_z(y)$  is computed where  $h_{z,i}(y)$  is the parity
   of incident edges that hash to  $[2^i]$ ,  $i = 0, \dots, l$ .
4:   For hash function  $h_z$ ,  $\vec{h}_z(T) = \bigoplus_{y \in T} \vec{h}_z(y)$  is computed in the leader.
5:   For each  $i = 0, \dots, l$ ,  $X_i = \sum_{z=1}^{c \log n} h_{z,i}(T)$ .
6:   Let  $min$  be the smallest  $i$  s.t.  $X_i \geq (3/4)c \log n / 16$ .
7:   Return  $2^{l-min} / 64$ .
8: end procedure

```

---

#### 4 Finding MST with $o(m)$ asynchronous communication

The MST algorithm implements a version of the GHS algorithm which grows a forest of disjoint subtrees (“fragments”) of the MST in parallel. We reduce the message complexity of GHS by using *FindMin* to find minimum weight outgoing edges *without* having to send messages across every edge. But, by doing this, we require the use of a spanning tree to help synchronize the growth of the fragments.

Note that GHS nodes send messages along their incident edges for two main purposes: (1) to see whether the edge is outgoing, and (2) to make sure that fragments with higher rank are slowed down and do not impose a lot of time and message complexity. Therefore, if we use *FindMin* instead of having nodes to send messages to their neighbors, we cannot make sure that higher ranked fragments are slowed down. Our protocol works in phases where in each phase only fragments with smallest ranks continue to grow while other fragments wait. A spanning tree is used to control the fragments based on their rank. (See [14] for the original GHS.)

**Implementation of FindMST.** Initially, each node forms a fragment containing only that node which is also the leader of the fragment and fragments all have rank zero. A *fragment identity* is the node ID of the fragment's leader; all nodes in a fragment know its identity and its current rank. Let the pre-computed spanning tree  $T$  be rooted at a node  $r$ . All fragment leaders wait for instructions that are broadcast by  $r$  over  $T$ .

The algorithm runs in phases. At the start of each phase,  $r$  broadcasts the message  $\langle Rank\text{-request} \rangle$  to learn the current minimum rank among all fragments after this broadcast. Leaves of  $T$  send up their fragment rank. Once an internal node in  $T$  receives the rank from all of its children (in  $T$ ) the node sends up the minimum fragment rank it has received including its own. This kind of computation is also referred to as a *convergecast*.

Then,  $r$  broadcasts the message  $\langle Proceed, minRank \rangle$  where *minRank* is the current minimum rank among all fragments. Any fragment leader that has rank equal to *minRank*, *proceeds* to finding minimum weight outgoing edges by calling *FindMin* on its own fragment tree. These fragments then send a  $\langle Connect \rangle$  message over their minimum weight outgoing edges. When a node  $v$  in fragment  $F$  (at rank  $R$ ) sends a  $\langle Connect \rangle$  message over an edge  $e$  to a node  $v'$  in fragment  $F'$  (at rank  $R'$ ), since  $R$  is the current minimum rank, two cases may happen: (Ranks and identities are updated here.)

1.  $R < R'$ : In this case,  $v'$  answers immediately to  $v$  by sending back an  $\langle Accept \rangle$  message, indicating that  $F$  can merge with  $F'$ . Then,  $v$  initiates the merge by changing its fragment identity to the identity of  $F'$ , making  $v'$  its parent, and broadcasting  $F'$ 's identity over fragment  $F$  so that all nodes in  $F$  update their fragment identity as well. Also, the new fragment (containing  $F$  and  $F'$ ) has rank  $R'$ .
2.  $R = R'$ :  $v'$  responds  $\langle Accept \rangle$  immediately to  $v$  if the minimum outgoing edge of  $F'$  is  $e$ , as well. In this case,  $F$  merges with  $F'$  as mentioned in rule 1, and the new fragment will have  $F'$ 's identity. Also, both fragments increase their rank to  $R' + 1$ .  
Otherwise,  $v'$  does not respond to the message until  $F'$ 's rank increases. Once  $F'$  increased its rank, it responds via an  $\langle Accept \rangle$  message, fragments merge, and the new fragment will update its rank to  $R'$ .

The key point here is that fragments at minimum rank are not kept waiting. Also, the intuition behind rule 2 is as follows. Imagine we have fragments  $F_1, F_2, \dots, F_k$  which all have the same rank and  $F_i$ 's minimum outgoing edge goes to  $F_{i+1}$  for  $i \leq k - 1$ . Now, it is either the case that  $F_k$ 's minimum outgoing edge goes to a fragment with higher rank or it goes to  $F_k$ . In either case, rule 2 allows the fragments  $F_{k-1}, F_{k-2}, \dots$  to update their identities in a cascading manner right after  $F_k$  increased its rank.

When all fragments finish their merge at this phase they have increased their rank by at least one. Now, it is time for  $r$  to star a new phase. However, since communication is asynchronous we need a way to tell whether all fragments have finished. In order to do this,  $\langle Done \rangle$  messages are convergecast in  $T$ . Nodes that were at minimum rank send up to their parent in  $T$  a  $\langle Done \rangle$  message only after they increased their rank and received  $\langle Done \rangle$  messages from all of their children in  $T$ .

---

**Algorithm 6** MST construction with  $\tilde{O}(n)$  messages.  $T$  is a spanning tree rooted at  $r$ .

---

```

1: procedure FINDMST
2:   All nodes are initialized as fragments at rank 0.
   // Start of a phase
3:    $r$  calls Broadcast( $\langle Rank\text{-request} \rangle$ ), and  $minRank$  is computed via a convergecast.
4:    $r$  calls Broadcast( $\langle Proceed, minRank \rangle$ ).
5:   Fragment leaders at rank  $minRank$  that have received the  $\langle Proceed, minRank \rangle$ 
   message, call FindMin. Then, these fragments merge by sending Connect messages
   over their minimum outgoing edges. If there is no outgoing edge the fragment leader
   terminates the algorithm.
6:   Upon receipt of  $\langle Proceed, minRank \rangle$ , a node  $v$  does the following:
   If it is a leaf in  $T$  at rank  $minRank$ , sends up  $\langle Done \rangle$  after increasing its rank.
   If it is a leaf in  $T$  with a rank higher than  $minRank$ , it immediately sends up  $\langle Done \rangle$ .
   If it is not a leaf in  $T$ , waits for  $\langle Done \rangle$  from its children in  $T$ . Then, sends up the
    $\langle Done \rangle$  message after increasing its rank.
7:    $r$  waits to receive  $\langle Done \rangle$  from all of its children, and starts a new phase at step 3.
8: end procedure

```

---

As proved in Lemma 8, this algorithm uses  $\tilde{O}(n)$  messages.

► **Lemma 8.** *FindMST uses  $O(n \log^3 n / \log \log n)$  messages and finds the MST w.h.p.*

**Proof.** All fragments start at rank zero. Before a phase begins, two broadcasts and convergecasts are performed to only allow fragments at minimum rank to proceed. This requires  $O(n)$  messages. In each phase, finding the minimum weight outgoing edges using FindMin takes  $O(n \log^2 n / \log \log n)$  over all fragments. Also, it takes  $O(n)$  for the fragments to update their identity since they just have to send the identity of the higher ranked fragment over their own fragment. As a result, each phase takes  $O(n \log^2 n / \log \log n)$  messages.

A fragment at rank  $R$  must contain at least two fragments with rank  $R - 1$ ; therefore, a fragment with rank  $R$  must have at least  $2^R$  nodes. So, the rank of a fragment never exceeds  $\log n$ . Also, each phase increases the minimum rank by at least one. Hence, there are at most  $\log n$  phases. As a result, message complexity is  $O(n \log^3 n / \log \log n)$ . ◀

From Lemma 8, Theorem 1 for minimum spanning trees follows.

## 5 Conclusion

We presented the first asynchronous algorithm for computing the MST in the CONGEST model with  $\tilde{O}(n^{3/2})$  communication when nodes have initial knowledge of their neighbors' identities. This shows that the KT1 model is significantly more communication efficient than KT0 even in the asynchronous model. Open problems that are raised by these results are: (1) Does the asynchronous KT1 model require substantially more communication than the synchronous KT1 model? (2) Can we improve the time complexity of the algorithm while maintaining the message complexity?

---

**References**

---

- 1 Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Graph sketches: sparsification, spanners, and subgraphs. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI symposium on Principles of Database Systems*, pages 5–14. ACM, 2012.
- 2 Baruch Awerbuch. Complexity of network synchronization. *Journal of the ACM (JACM)*, 32(4):804–823, 1985.
- 3 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. ACM, 1987.
- 4 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.
- 5 Baruch Awerbuch, Shay Kutten, Yishay Mansour, Boaz Patt-Shamir, and George Varghese. A time-optimal self-stabilizing synchronizer using a phase clock. *IEEE Transactions on Dependable and Secure Computing*, 4(3), 2007.
- 6 Baruch Awerbuch and David Peleg. Network synchronization with polylogarithmic overhead. In *Foundations of Computer Science, 1990. Proceedings., 31st Annual Symposium on*, pages 514–522. IEEE, 1990.
- 7 Michael Elkin. A faster distributed protocol for constructing a minimum spanning tree. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 359–368. Society for Industrial and Applied Mathematics, 2004.
- 8 Michael Elkin. An unconditional lower bound on the time-approximation trade-off for the distributed minimum spanning tree problem. *SIAM Journal on Computing*, 36(2):433–456, 2006.
- 9 Michael Elkin. Synchronizers, spanners. In *Encyclopedia of Algorithms*, pages 1–99. Springer, 2008.
- 10 Michael Elkin. Distributed exact shortest paths in sublinear time. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017*, pages 757–770, New York, NY, USA, 2017. ACM. doi:10.1145/3055399.3055452.
- 11 Michael Elkin. A simple deterministic distributed mst algorithm, with near-optimal time and message complexities. *arXiv preprint arXiv:1703.02411*, 2017.
- 12 Yuval Emek and Amos Korman. Efficient threshold detection in a distributed environment. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 183–191. ACM, 2010.
- 13 Michalis Faloutsos and Mart Molle. Optimal distributed algorithm for minimum spanning trees revisited. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 231–237. ACM, 1995.
- 14 Robert G. Gallager, Pierre A. Humblet, and Philip M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and systems (TOPLAS)*, 5(1):66–77, 1983.
- 15 Juan A Garay, Shay Kutten, and David Peleg. A sublinear time distributed algorithm for minimum-weight spanning trees. *SIAM Journal on Computing*, 27(1):302–316, 1998.
- 16 Bruce M Kapron, Valerie King, and Ben Mountjoy. Dynamic graph connectivity in polylogarithmic worst case time. In *Proceedings of the twenty-fourth annual ACM-SIAM symposium on Discrete algorithms*, pages 1131–1142. Society for Industrial and Applied Mathematics, 2013.
- 17 Maleq Khan and Gopal Pandurangan. A fast distributed approximation algorithm for minimum spanning trees. In *Proceedings of the 20th International Conference on Distributed Computing, DISC’06*, pages 355–369, Berlin, Heidelberg, 2006. Springer-Verlag. doi:10.1007/11864219\_25.

- 18 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. ACM, 2015.
- 19 Shay Kutten, Gopal Pandurangan, David Peleg, Peter Robinson, and Amitabh Trehan. On the complexity of leader election. *Journal of the ACM (JACM)*, 62(1):7, 2015.
- 20 Shay Kutten and David Peleg. Fast distributed construction of  $k$ -dominating sets and applications. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 238–251. ACM, 1995.
- 21 Ali Mashreghi and Valerie King. Time-communication trade-offs for minimum spanning tree construction. In *Proceedings of the 18th International Conference on Distributed Computing and Networking*, page 8. ACM, 2017.
- 22 Ali Mashreghi and Valerie King. Broadcast and minimum spanning tree with  $o(m)$  messages in the asynchronous congest model. *arXiv*, 2018. [arXiv:1806.04328](https://arxiv.org/abs/1806.04328).
- 23 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.
- 24 David Peleg and Jeffrey D Ullman. An optimal synchronizer for the hypercube. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 77–85. ACM, 1987.
- 25 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.
- 26 Gurdip Singh and Arthur J Bernstein. A highly asynchronous minimum spanning tree protocol. *Distributed Computing*, 8(3):151–161, 1995.



# Fault-Tolerant Consensus with an Abstract MAC Layer

Calvin Newport<sup>1</sup>

Georgetown University, Washington, D.C., USA  
cnewport@cs.georgetown.edu

Peter Robinson<sup>2</sup>

McMaster University, Hamilton, Ontario, Canada  
peter.robinson@mcmaster.ca

---

## Abstract

In this paper, we study fault-tolerant distributed consensus in wireless systems. In more detail, we produce two new randomized algorithms that solve this problem in the abstract MAC layer model, which captures the basic interface and communication guarantees provided by most wireless MAC layers. Our algorithms work for any number of failures, require no advance knowledge of the network participants or network size, and guarantee termination with high probability after a number of broadcasts that are polynomial in the network size. Our first algorithm satisfies the standard agreement property, while our second trades a faster termination guarantee in exchange for a looser agreement property in which most nodes agree on the same value. These are the first known fault-tolerant consensus algorithms for this model. In addition to our main upper bound results, we explore the gap between the abstract MAC layer and the standard asynchronous message passing model by proving fault-tolerant consensus is impossible in the latter in the absence of information regarding the network participants, even if we assume no faults, allow randomized solutions, and provide the algorithm a constant-factor approximation of the network size.

**2012 ACM Subject Classification** Theory of computation → Distributed algorithms

**Keywords and phrases** abstract MAC layer, wireless networks, consensus, fault tolerance

**Digital Object Identifier** 10.4230/LIPIcs.DISC.2018.38

**Related Version** A full version of the paper is available at [42], <https://www.cas.mcmaster.ca/robinson/random-aml.pdf>.

## 1 Introduction

Consensus provides a fundamental building block for developing reliable distributed systems [23–25]. Accordingly, it is well studied in many different system models [36]. Until recently, however, little was known about solving this problem in distributed systems made up of devices communicating using commodity wireless cards. Motivated by this knowledge gap, this paper studies consensus in the *abstract MAC layer* model, which abstracts the basic behavior and guarantees of standard wireless MAC layers. In recent work [41], we proved deterministic fault-tolerant consensus is impossible in this setting. In this paper, we describe and analyze the first known randomized fault-tolerant consensus algorithms for this well-motivated model.

---

<sup>1</sup> Calvin Newport acknowledges the support of the National Science Foundation, award number 1733842.

<sup>2</sup> Peter Robinson acknowledges the support of the Natural Sciences and Engineering Research Council of Canada (NSERC), RGPIN-2018-06322.





**The Abstract MAC Layer.** Most existing work on distributed algorithms for wireless networks assumes low-level synchronous models that force algorithms to directly grapple with issues caused by contention and signal fading. Some of these models describe the network topology with a graph (c.f., [8, 16, 20, 28, 32, 38]), while others use signal strength calculations to determine message behavior (c.f., [17, 21, 26, 27, 37, 39]).

As also emphasized in [41], these models are useful for asking foundational questions about distributed computation on shared channels, but are not so useful for developing algorithmic strategies suitable for deployment. In real systems, algorithms typically do not operate in synchronous rounds and they are not provided unmediated access to the radio. They must instead operate on top of a general-purpose MAC layer which is responsible for many network functions, including contention management, rate control, and co-existence with other network traffic.

Motivated by this reality, in this paper we adopt the *abstract MAC layer* model [34], an asynchronous broadcast-based communication model that captures the basic interfaces and guarantees provided by common existing wireless MAC layers. In more detail, if you provide the abstract MAC layer a message to broadcast, it will eventually be delivered to nearby nodes in the network. The specific means by which contention is managed – e.g., CSMA, TDMA, uniform probabilistic routines such as DECAY [8] – is abstracted away by the model. At some point after the contention management completes, the abstract MAC layer passes back an *acknowledgment* indicating that it is ready for the next message. This acknowledgment contains no information about the number or identities of the message recipient.

(In the case of the MAC layer using CSMA, for example, the acknowledgment would be generated after the MAC layer detects a clear channel. In the case of TDMA, the acknowledgment would be generated after the device’s turn in the TDMA schedule. In the case of a probabilistic routine such as DECAY, the acknowledgment would be generated after a sufficient number of attempts to guarantee successful delivery to all receivers with high probability.)

The abstract MAC abstraction, of course, does not attempt to provide a detailed representation of any specific existing MAC layer. Real MAC layers offer many more modes and features than is captured by this model. In addition, the variation studied in this paper assumes messages are always delivered, whereas more realistic variations would allow for occasional losses.

This abstraction, however, still serves to capture the fundamental dynamics of real wireless application design in which the lower layers dealing directly with the radio channel are separated from the higher layers executing the application in question. An important goal in studying this abstract MAC layer, therefore, is attempting to uncover principles and strategies that can close the gap between theory and practice in the design of distributed systems deployed on standard layered wireless architectures.

**Our Results.** In this paper, we studied randomized fault-tolerant consensus algorithms in the abstract MAC layer model. In more detail, we study binary consensus and assume a single-hop network topology. Notice, our use of randomization is necessary, as deterministic consensus is impossible in the abstract MAC layer model in the presence of even a single fault (see our generalization of FLP from [41]).

To contextualize our results, we note that the abstract MAC layer model differs from standard asynchronous message passing models in two main ways: (1) the abstract MAC layer model provides the algorithm no advance information about the network size or membership,

requiring nodes to communicate with a blind broadcast primitive instead of using point-to-point channels, (2) the abstract MAC layer model provides an acknowledgment to the broadcaster at some point after its message has been delivered to all of its neighbors. This acknowledgment, however, contains no information about the number or identity of these neighbors (see above for more discussion of this fundamental feature of standard wireless MAC layers).

Most randomized fault-tolerant consensus algorithms in the asynchronous message passing model strongly leverage knowledge of the network. A strategy common to many of these algorithms, for example, is to repeatedly collect messages from at least  $n - f$  nodes in a network of size  $n$  with at most  $f$  crash failures (e.g., [9]). This strategy does not work in the abstract MAC layer model as nodes do not know  $n$ .

To overcome this issue, we adapt an idea introduced in early work on fault-tolerant consensus in the asynchronous shared memory model: *counter racing* (e.g., [5, 12]). At a high-level, this strategy has nodes with initial value 0 advance a shared memory counter associated with 0, while nodes with initial value 1 advance a counter associated with 1. If a node sees one counter get ahead of the other, they adopt the initial value associated with the larger counter, and if a counter gets sufficiently far ahead, then nodes can decide.

Our first algorithm (presented in Section 3) implements a counter race of sorts using the acknowledged blind broadcast primitive provided by the model. Roughly speaking, nodes continually broadcast their current proposal and counter, and update both based on the pairs received from other nodes. Proving safety for this type of strategy in shared memory models is simplified by the atomic nature of register accesses. In the abstract MAC layer model, by contrast, a broadcast message is delivered non-atomically to its recipients, and in the case of a crash, may not arrive at some recipients at all.<sup>3</sup> Our safety analysis, therefore, requires novel analytical tools that tame a more diverse set of possible system configurations.

To achieve liveness, we use a technique loosely inspired by the randomized delay strategy introduced by Chandra in the shared memory model [12]. In more detail, nodes probabilistically decide to replace certain sequences of their counter updates with *nop* placeholders. We show that if these probabilities are adapted appropriately, the system eventually arrives at a state where it becomes likely for only a single node to be broadcasting updates, allowing progress toward termination.

Formally, we prove that with high probability in the network size  $n$ , the algorithm terminates after  $O(n^3 \log n)$  broadcasts are scheduled. This holds regardless of which broadcasts are scheduled (i.e., we do not impose a fairness condition), and regardless of the number of faults. The algorithm, as described, assumes nodes are provided unique IDs that we treat as comparable black boxes (to prevent them from leaking network size information). We subsequently show how to remove that assumption by describing an algorithm that generates unique IDs in this setting with high probability.

Our second algorithm (presented in Section 4) trades a looser agreement guarantee for more efficiency. In more detail, we describe and analyze a solution to *almost-everywhere* agreement [18], that guarantees most nodes agree on the same value. This algorithm terminates after  $O(n^2 \log^4 n \log \log n)$  broadcasts, which is a linear factor faster than our first algorithm (ignoring log factors). The almost-everywhere consensus algorithm consists of two phases. The first phase is used to ensure that almost all nodes obtain a good approximation

---

<sup>3</sup> We note that register simulations are also not an option in our model for two reasons: standard simulation algorithms require knowledge of  $n$  and a majority correct nodes, whereas we assume no knowledge of  $n$  and wait-freedom.

of the network size. In the second phase, nodes use this estimate to perform a sequence of broadcasts meant to help spread their proposal to the network. Nodes that did not obtain a good estimate in Phase 1 will leave Phase 2 early. The remaining nodes, however, can leverage their accurate network size estimates to probabilistically sample a subset to actively participate in each round of broadcasts. To break ties between simultaneously active nodes, each chooses a random rank using the estimate obtained in Phase 1. We show that with high probability, after not too long, there exists a round of broadcasts in which the first node receiving its acknowledgment is both active and has the minimum rank among other active nodes – allowing its proposal to spread to all remaining nodes.

Finally, we explore the gap between the abstract MAC layer model and the related asynchronous message passing model. We prove (in Section 5) that fault-tolerant consensus is impossible in the asynchronous message passing model in the absence of knowledge of network participants, even if we assume no faults, allow randomized algorithms, and provide a constant-factor approximation of  $n$ . This differs from the abstract MAC layer model where we solve this problem without network participant or network size information, and assuming crash failures. This result implies that the fact that broadcasts are acknowledged in the abstract MAC layer model is crucial to overcoming the difficulties induced by limited network information.

**Related Work.** Consensus provides a fundamental building block for reliable distributed computing [23–25]. It is particularly well-studied in asynchronous models [2, 35, 40, 44].

The abstract MAC layer approach<sup>4</sup> to modeling wireless networks was introduced in [33] (later expanded to a journal version [34]), and has been subsequently used to study several different problems [14, 15, 29, 30, 41]. The most relevant of this related work is [41], which was the first paper to study consensus in the abstract MAC layer model. This previous paper generalized the seminal FLP [19] result to prove deterministic consensus is impossible in this model even in the presence of a single failure. It then goes on to study deterministic consensus in the absence of failures, identifying the pursuit of fault-tolerant *randomized* solutions as important future work – the challenge taken up here.

We note that other researchers have also studied consensus using high-level wireless network abstractions. Vollset and Ezhilchelvan [45], and Alekeish and Ezhilchelvan [4], study consensus in a variant of the asynchronous message passing model where pairwise channels come and go dynamically – capturing some behavior of mobile wireless networks. Their correctness results depend on detailed liveness guarantees that bound the allowable channel changes. Wu et al. [46] use the standard asynchronous message passing model (with unreliable failure detectors [13]) as a stand-in for a wireless network, focusing on how to reduce message complexity (an important metric in a resource-bounded wireless setting) in solving consensus.

A key difficulty for solving consensus in the abstract MAC layer model is the absence of advance information about network participants or size. These constraints have also been studied in other models. Ruppert [43], and Bonnet and Raynal [10], for example, study the amount of extra power needed (in terms of shared objects and failure detection, respectively) to solve wait-free consensus in *anonymous* versions of the standard models. Attiya et al. [6] describe consensus solutions for shared memory systems without failures or unique ids. A

---

<sup>4</sup> There is no *one* abstract MAC layer model. Different studies use different variations. They all share, however, the same general commitment to capturing the types of interfaces and communication/timing guarantees that are provided by standard wireless MAC layers

series of papers [3, 11, 22], starting with the work of Cavin et al. [11], study the related problem of *consensus with unknown participants* (CUPs), where nodes are only allowed to communicate with other nodes whose identities have been provided by a *participant detector* formalism.

Closer to our own model is the work of Abboud et al. [1], which also studies a single hop network where nodes broadcast messages to an unknown group of network participants. They prove deterministic consensus is impossible in these networks under these assumptions without knowledge of network size. In this paper, we extend these existing results by proving this impossibility still holds even if we assume randomized algorithms and provided the algorithm a constant-factor approximation of the network size. This bound opens a sizable gap with our abstract MAC layer model in which consensus is solvable without this network information.

We also consider almost-everywhere (a.e.) agreement [18], a weaker variant of consensus, where a small number of nodes are allowed to decide on conflicting values, as long as a sufficiently large majority agrees. Recently, a.e. agreement has been studied in the context of peer-to-peer networks (c.f. [7, 31]), where the adversary can isolate small parts of the network thus rendering (everywhere) consensus impossible. We are not aware of any prior work on a.e. agreement in the wireless settings.

## 2 Model and Problem

In this paper, we study a variation of the *abstract MAC layer* model, which describes system consisting of a single hop network of  $n \geq 1$  computational devices (called *nodes* in the following) that communicate wirelessly using communication interfaces and guarantees inspired by commodity wireless MAC layers.

In this model, nodes communicate with a *bcast* primitive that guarantees to eventually deliver the broadcast message to all the other nodes (i.e., the network is single hop). At some point after a given *bcast* has succeeded in delivering a message to all other nodes, the broadcaster receives an *ack* informing it that the broadcast is complete (as detailed in the introduction, this captures the reality that most wireless contention management schemes have a definitive point at which they know a message broadcast is complete). This acknowledgment contains no information about the number or identity of the receivers.

We assume a node can only broadcast one message at a time. That is, once it invokes *bcast*, it cannot broadcast another message until receiving the corresponding *ack* (formally, overlapping messages are discarded by the MAC layer). We also assume any number of nodes can permanently stop executing due to crash failures. As in the classical message passing models, a crash can occur during a broadcast, meaning that some nodes might receive the message while others do not.

This model is event-driven with the relevant events scheduled asynchronously by an arbitrary *scheduler*. In more detail, for each node  $u$ , there are four event types relevant to  $u$  that can be scheduled:  $init_u$  (which occurs at the beginning of an execution and allows  $u$  to initialize),  $recv(m)_u$  (which indicates that  $u$  has received message  $m$  broadcast from another node),  $ack(m)_u$  (which indicates that the message  $m$  broadcast by  $u$  has been successfully delivered), and  $crash_u$  (which indicates that  $u$  is crashed for the remainder of the execution).

A distributed algorithm specifies for each node  $u$  a finite collection of steps to execute for each of the non-*crash* event types. When one of these events is scheduled by the scheduler, we assume the corresponding steps are executed atomically at the point that the event is

scheduled. Notice that one of the steps that a node  $u$  can take in response to these events is to invoke a  $bcast(m)_u$  primitive for some message  $m$ . When an event includes a  $bcast$  primitive we say it is *combined* with a broadcast.<sup>5</sup>

We place the following constraints on the scheduler. It must start each execution by scheduling an *init* event for each node; i.e., we study the setting where all participating nodes are activated at the beginning of the execution. If a node  $u$  invokes a valid  $bcast(m)_u$  primitive, then for each  $v \neq u$  that is not crashed when the broadcast primitive is invoked, the scheduler must subsequently either schedule a single  $recv(m)_v$  or  $crash_v$  event at  $v$ . At some point after these events are scheduled, it must then eventually schedule an  $ack(m)_u$  event at  $u$ . These are the only *recv* and *ack* events it schedules (i.e., it cannot create new messages from scratch or cause messages to be received/acknowledged multiple times). If the scheduler schedules a  $crash_v$  event, it cannot subsequently schedule any future events for  $u$ .

We assume that in making each event scheduling decision, the scheduler can use the schedule history as well as the algorithm definition, but it does not know the nodes' private states (which includes the nodes' random bits). When the scheduler schedules an event that triggers a broadcast (making it a combined event), it is provided this information so that it knows it must now schedule receive events for the message. We assume, however, that the scheduler does not learn the *contents* of the broadcast message.<sup>6</sup>

Given an execution  $\alpha$ , we say the *message schedule* for  $\alpha$ , also indicated  $msg[\alpha]$ , is the sequence of message events (i.e., *recv*, *ack*, and *crash*) scheduled in the execution. We assume that a message schedule includes indications of which events are combined with broadcasts.

**The Consensus Problem.** In this paper, we study binary consensus with probabilistic termination. In more detail, at the beginning of an execution each node is provided an *initial value* from  $\{0, 1\}$  as input. Each node has the ability to perform a single irrevocable *decide* action for either value 0 or 1. To solve consensus, an algorithm must guarantee the following three properties: (1) *agreement*: no two nodes decide different values; (2) *validity*: if a node decides value  $b$ , then at least one node started with initial value  $b$ ; and (3) *termination (probabilistic)*: every non-crashed node decides with probability 1 in the limit.

Studying finite termination bounds is complicated in asynchronous models because the scheduler can delay specific nodes taking steps for arbitrarily long times. In this paper, we circumvent this issue by proving bounds on the number of scheduled events before the system reaches a *termination state* in which every non-crashed node has: (a) decided; or (b) will decide whenever the scheduler gets around to scheduling its next *ack* event.

Finally, in addition to studying consensus with standard agreement, we also study *almost-everywhere* agreement, in which only a specified majority fraction (typically a  $1 - o(n)$  fraction of the  $n$  total nodes) must agree.

<sup>5</sup> Notice, we can assume without loss of generality, that the steps executed in response to an event never invoke more than a single *bcast* primitive, as any additional broadcasts invoked at the same time would lead to the messages being discarded due to the model constraint that a node must receive an *ack* for the current message before broadcasting a new message.

<sup>6</sup> This adversary model is sometimes called *message oblivious* and it is commonly considered a good fit for schedulers that control network behavior. This follows because it allows the scheduler to adapt the schedule based on the number of messages being sent and their sources – enabling it to model contention and load factors. On the other hand, there is not good justification for the idea that this schedule should somehow also depend on the specific bits contained in the messages sent. Notice, our liveness proof specifically leverages the message oblivious assumption as it prevents the scheduler from knowing which nodes are sending updates and which are sending *nop* messages.

---

**Algorithm 1** Counter Race Consensus (for node  $u$  with UID  $id_u$  and initial value  $v_u$ )
 

---

Initialization:

$c_u \leftarrow 0$   
 $n_u \leftarrow 2$   
 $C_u \leftarrow \{(id_u, c_u, v_u)\}$   
 $peers \leftarrow \{id_u\}$   
 $phase \leftarrow 0$   
 $active \leftarrow true$   
 $decide \leftarrow -1$   
 $k \leftarrow 3$   
 $c \leftarrow k + 3$   
**bcast**( $nop, id_u, n_u$ )

On Receiving  $ack(m)$ :

$phase \leftarrow phase + 1$   
**if**  $m = (decide, b)$  **then**  
     **decide**( $b$ ) and **halt**()  
**else**  
      $newm \leftarrow \perp$   
      $C'_u \leftarrow C_u$   
      $\hat{c}_u^{(0)} \leftarrow$  max counter in  $C'_u$  paired with value 0 (default to 0 if no such elements)  
      $\hat{c}_u^{(1)} \leftarrow$  max counter in  $C'_u$  paired with value 1 (default to 0 if no such elements)  
     **if**  $\hat{c}_u^{(0)} > \hat{c}_u^{(1)}$  **then**  $v_u \leftarrow 0$   
     **else if**  $\hat{c}_u^{(1)} > \hat{c}_u^{(0)}$  **then**  $v_u \leftarrow 1$   
     **if**  $\hat{c}_u^{(0)} \geq \hat{c}_u^{(1)} + k$  **or**  $decide = 0$  **then**  $newm \leftarrow (decide, 0)$   
     **else if**  $\hat{c}_u^{(1)} \geq \hat{c}_u^{(0)} + k$  **or**  $decide = 1$  **then**  $newm \leftarrow (decide, 1)$   
     **if**  $newm = \perp$  **then**  
         **if**  $\max\{\hat{c}_u^{(0)}, \hat{c}_u^{(1)}\} \leq c_u$  **and**  $m \neq nop$  **then**  $c_u \leftarrow c_u + 1$   
         **else if**  $\max\{\hat{c}_u^{(0)}, \hat{c}_u^{(1)}\} > c_u$  **then**  $c_u \leftarrow \max\{\hat{c}_u^{(0)}, \hat{c}_u^{(1)}\}$   
         **update**  $(id_u, *, *)$  element in  $C_u$  with new  $c_u$  and  $v_u$   
          $newm \leftarrow (counter, id_u, c_u, v_u, n_u)$   
     **if**  $phase \% c = 1$  **then** with probability  $1/n_u$   $active \leftarrow true$  otherwise  $active \leftarrow false$   
     **if**  $newm = (decide, *)$  **or**  $active = true$  **then**  
         **bcast**( $newm$ )  
     **else**  
         **bcast**( $nop, id_u, n_u$ )

On Receiving Message  $m$ :

**updateEstimate**( $m$ )  
**if**  $m = (decide, b)$  **then**  
      $decide \leftarrow b$   
**else if**  $m = (counter, id, c, v, n')$  **then**  
     **if**  $\exists c', v'$  such that  $(id, c', v') \in C_u$  **then**  
         **remove**  $(id, c', v')$  from  $C_u$   
     **add**  $(id, c, v)$  to  $C_u$

---

---

**Algorithm 2** The `updateEstimate( $m$ )` subroutine called by Counter Race Consensus during `recv( $m$ )` event.

---

```

if  $m$  contains a UID  $id$  and network size estimate  $n'$  then
     $peers \leftarrow peers \cup \{id\}$ 
     $n_u \leftarrow \max\{n_u, |peers|, n'\}$ 

```

---

### 3 Upper Bound

Here we describe and analyze our first randomized binary consensus algorithm: *counter race consensus* (see Algorithms 1 and 2 for pseudocode, and Section 3.1 for a high-level description of its behavior). This algorithm assumes no advance knowledge of the network participants or network size. Nodes are provided unique IDs, but these are treated as comparable black boxes, preventing them from leaking information about the network size. (We will later discuss how to remove the unique ID assumption.) It tolerates any number of crash faults. The detailed proofs can be found in the full paper [42].

#### 3.1 Algorithm Description

The counter race consensus algorithm is described in pseudocode in the figures labeled Algorithm 1 and 2. Here we summarize the behavior formalized by this pseudocode.

The core idea of this algorithm is that each node  $u$  maintains a counter  $c_u$  (initialized to 0) and a proposal  $v_u$  (initialized to its consensus initial value). Node  $u$  repeatedly broadcasts  $c_u$  and  $v_u$ , updating these values before each broadcast. That is, during the *ack* event for its last broadcast of  $c_u$  and  $v_u$ , node  $u$  will apply a set of *update rules* to these values. It then concludes the *ack* event by broadcasting these updated values. This pattern repeats until  $u$  arrives at a state where it can safely commit to deciding a value.

The update rules and decision criteria applied during the *ack* event are straightforward. Each node  $u$  first calculates  $\hat{c}_u^{(0)}$ , the largest counter value it has sent or received in a message containing proposal value 0, and  $\hat{c}_u^{(1)}$ , the largest counter value it has sent or received in a message containing proposal value 1.

If  $\hat{c}_u^{(0)} > \hat{c}_u^{(1)}$ , then  $u$  sets  $v_u \leftarrow 0$ , and if  $\hat{c}_u^{(1)} > \hat{c}_u^{(0)}$ , then  $u$  sets  $v_u \leftarrow 1$ . That is,  $u$  adopts the proposal that is currently “winning” the counter race (in case of a tie, it does not change its proposal).

Node  $u$  then checks to see if either value is winning by a large enough margin to support a decision. In more detail, if  $\hat{c}_u^{(0)} \geq \hat{c}_u^{(1)} + 3$ , then  $u$  commits to deciding 0, and if  $\hat{c}_u^{(1)} \geq \hat{c}_u^{(0)} + 3$ , then  $u$  commits to deciding 1.

What happens next depends on whether or not  $u$  committed to a decision. If  $u$  did *not* commit to a decision (captured in the **if** `newm =  $\perp$`  **then** conditional), then it must update its counter value. To do so, it compares its current counter  $c_u$  to  $\hat{c}_u^{(0)}$  and  $\hat{c}_u^{(1)}$ . If  $c_u$  is smaller than one of these counters, it sets  $c_u \leftarrow \max\{\hat{c}_u^{(0)}, \hat{c}_u^{(1)}\}$ . Otherwise, if  $c_u$  is the largest counter that  $u$  has sent or received so far, it will set  $c_u \leftarrow c_u + 1$ . Either way, its counter increases. At this point,  $u$  can complete the *ack* event by broadcasting a message containing its newly updated  $c_u$  and  $v_u$  values.

On the other hand, if  $u$  committed to deciding value  $b$ , then it will send a *(decide,  $b$ )* message to inform the other nodes of its decision. On subsequently receiving an *ack* for this message,  $u$  will decide  $b$  and halt. Similarly, if  $u$  ever receives a *(decide,  $b$ )* message from *another* node, it will commit to deciding  $b$ . During its next *ack* event, it will send its



own  $(decide, b)$  message and decide and halt on its corresponding  $ack$ . That is, node  $u$  will not decide a value until it has broadcast its commitment to do so, and received an  $ack$  on the broadcast.

The behavior described above guarantees agreement and validity. It is not sufficient, however, to achieve liveness, as an ill-tempered scheduler can conspire to keep the race between 0 and 1 too close for a decision commitment. To overcome this issue we introduce a random delay strategy that has nodes randomly step away from the race for a while by replacing their broadcast values with *nop* placeholders ignored by those who receive them. Because our adversary does not learn the *content* of broadcast messages, it does not know which nodes are actively participating and which nodes are taking a break (as in both cases, nodes continually broadcast messages) – thwarting its ability to effectively manipulate the race.

In more detail, each node  $u$  partitions its broadcasts into *groups* of size 6. At the beginning of each such group,  $u$  flips a weighted coin to determine whether or not to replace the counter and proposal values it broadcasts during this group with *nop* placeholders – eliminating its ability to affect other nodes’ counter/proposal values. As we will later elaborate in the liveness analysis, the goal is to identify a point in the execution in which a single node  $v$  is broadcasting its values while all other nodes are broadcasting *nop* values – allowing  $v$  to advance its proposal sufficiently far ahead to win the race.

To be more specific about the probabilities used in this logic, node  $u$  maintains an estimate  $n_u$  of the number of nodes in the network. It replaces values with *nop* placeholders in a given group with probability  $1/n_u$ . (In the pseudocode, the *active* flag indicates whether or not  $u$  is using *nop* placeholders in the current group.) Node  $u$  initializes  $n_u$  to 2. It then updates it by calling the *updateEstimate* routine (described in Algorithm 2) for each message it receives.

There are two ways for this routine to update  $n_u$ . The first is if the number of unique IDs that  $u$  has received so far (stored in *peers*) is larger than  $n_u$ . In this case, it sets  $n_u \leftarrow |peers|$ . The second way is if it learns another node has an estimate  $n' > n_u$ . In this case, it sets  $n_u \leftarrow n'$ . Node  $u$  learns about other nodes’ estimates, as the algorithm has each node append its current estimate to all of its messages (with the exception of *decide* messages). In essence, the nodes are running a network size estimation routine parallel to its main counter race logic – as nodes refine their estimates, their probability of taking useful breaks improves.

### 3.2 Safety

We begin our analysis by proving that our algorithm satisfies the agreement and validity properties of the consensus problem. Validity follows directly from the algorithm description. Our strategy to prove agreement is to show that if any node sees a value  $b$  with a counter at least 3 ahead of value  $1 - b$  (causing it to commit to deciding  $b$ ), then  $b$  is the only possible decision value. Race arguments of this type are easier to prove in a shared memory setting where nodes work with objects like atomic registers that guarantee linearization points. In our message passing setting, by contrast, in which broadcast messages arrive at different receivers at different times, we will require more involved definitions and operational arguments.<sup>7</sup>

We start with a useful definition. We say  $b$  *dominates*  $1 - b$  at a given point in the execution, if every (non-crashed) node at this point believes  $b$  is winning the race, and none of the messages in transit can change this perception.

<sup>7</sup> We had initially hoped there might be some way to simulate linearizable shared objects in our model. Unfortunately, our nodes’ lack of information about the network size thwarted standard simulation strategies which typically require nodes to collect messages from a majority of nodes in the network before proceeding to the next step of the simulation.

To formalize this notion we need some notation. In the following, we say *at point  $t$*  (or *at  $t$* ), with respect to an event  $t$  from the message schedule of an execution  $\alpha$ , to describe the state of the system immediately after event  $t$  (and any associated steps that execute atomically with  $t$ ) occurs. We also use the notation *in transit at  $t$*  to describe messages that have been broadcast but not yet received at every non-crashed receiver at  $t$ .

► **Definition 1.** Fix an execution  $\alpha$ , event  $t$  in the corresponding message schedule  $msg[\alpha]$ , consensus value  $b \in \{0, 1\}$ , and counter value  $c \geq 0$ . We say  $\alpha$  is  $(b, c)$ -dominated at  $t$  if the following conditions are true:

1. For every node  $u$  that is not crashed at  $t$ :  $\hat{c}_u^{(b)}[t] > c$  and  $\hat{c}_u^{(1-b)}[t] \leq c$ , where at point  $t$ ,  $\hat{c}_u^{(b)}[t]$  (resp.  $\hat{c}_u^{(1-b)}[t]$ ) is the largest value  $u$  has sent or received in a counter message containing consensus value  $b$  (resp.  $1 - b$ ). If  $u$  has not sent or received any counter messages containing  $b$  (resp.  $1 - b$ ), then by default it sets  $\hat{c}_u^{(b)}[t] \leftarrow 0$  (resp.  $\hat{c}_u^{(1-b)}[t] \leftarrow 0$ ) in making this comparison.
2. For every message of the form  $(counter, id, 1 - b, c', n')$  that is in transit at  $t$ :  $c' \leq c$ .

The following lemma formalizes the intuition that once an execution becomes dominated by a given value, it remains dominated by this value.

► **Lemma 2.** Assume some execution  $\alpha$  is  $(b, c)$ -dominated at point  $t$ . It follows that  $\alpha$  is  $(b, c)$ -dominated at every  $t'$  that comes after  $t$ .

**Proof.** In this proof, we focus on the suffix of the message schedule  $msg[\alpha]$  that begins with event  $t$ . For simplicity, we label these events  $E_1, E_2, E_3, \dots$ , with  $E_1 = t$ . We will prove the lemma by induction on this sequence.

The base case ( $E_1$ ) follows directly from the lemma statement. For the inductive step, we must show that if  $\alpha$  is  $(b, c)$ -dominated at point  $E_i$ , then it will be dominated at  $E_{i+1}$  as well. By the inductive hypothesis, we assume the execution is dominated immediately before  $E_{i+1}$  occurs. Therefore, the only way the step is violated is if  $E_{i+1}$  transitions the system from dominated to non-dominated status. We consider all possible cases for  $E_{i+1}$  and show none of them can cause such a transition.

The first case is if  $E_{i+1}$  is a  $crash_u$  event for some node  $u$ . It is clear that a crash cannot transition a system into non-dominated status.

The second case is if  $E_{i+1}$  is a  $recv(m)_u$  event for some node  $u$ . This event can only transition the system into a non-dominated status if  $m$  is a counter message that includes  $1 - b$  and a counter  $c' > c$ . For  $u$  to receive this message, however, means that the message was in transit immediately before  $E_{i+1}$  occurs. Because we assume the system is dominated at  $E_i$ , however, no such message can be in transit at this point (by condition 2 of the domination definition).

The third and final case is if  $E_{i+1}$  is a  $ack(m)_u$  event for some node  $u$ , that is combined with a  $bcast(m')_u$  event, where  $m'$  is a counter message that includes  $1 - b$  and a counter  $c' > c$ . Consider the values  $\hat{c}_u^{(b)}$  and  $\hat{c}_u^{(1-b)}$  set by node  $u$  early in the steps associated with this  $ack(m)_u$  event. By our inductive hypothesis, which tells us that the execution is dominated right before this  $ack(m)_u$  event occurs, it must follow that  $\hat{c}_u^{(b)} > \hat{c}_u^{(1-b)}$  (as  $\hat{c}_u^{(b)} = \hat{c}_u^{(b)}[E_i]$  and  $\hat{c}_u^{(1-b)} = \hat{c}_u^{(1-b)}[E_i]$ ). In the steps that immediately follow, therefore, node  $u$  will set  $v_u \leftarrow b$ . It is therefore impossible for  $u$  to then broadcast a counter message with value  $v_u = 1 - b$ . ◀

To prove agreement, we are left to show that if a node commits to deciding some value  $b$ , then it must be the case that  $b$  dominates the execution at this point – making it the only possible decision going forward. The following helper lemma, which captures a useful property about counters, will prove crucial for establishing this point.

► **Lemma 3.** *Assume event  $t$  in the message schedule of execution  $\alpha$  is combined with a  $\text{bcast}(m)_v$ , where  $m = (\text{counter}, id_v, c, b, n_v)$ , for some counter  $c > 0$ . It follows that prior to  $t$  in  $\alpha$ , every node that is non-crashed at  $t$  received a counter message with counter  $c - 1$  and value  $b$ .*

**Proof.** Fix some  $t$ ,  $\alpha$ ,  $v$  and  $m = (\text{counter}, id_v, c, b, n_v)$ , as specified by the lemma statement. Let  $t'$  be the first event in  $\alpha$  such that at  $t'$  some node  $w$  has local counter  $c_w \geq c$  and value  $v_w = b$ . We know at least one such event exists as  $t$  and  $v$  satisfy the above conditions, so the earliest such event,  $t'$ , is well-defined. Furthermore, because  $t'$  must modify local counter and/or consensus values, it must also be an *ack* event.

For the purposes of this argument, let  $c_w$  and  $v_w$  be  $w$ 's counter and consensus value, respectively, immediately before  $t'$  is scheduled. Similarly, let  $c'_w$  and  $v'_w$  be these values immediately after  $t'$  and its steps complete (i.e., these values at point  $t'$ ). By assumption:  $c'_w \geq c$  and  $v'_w = b$ . We proceed by studying the possibilities for  $c_w$  and  $v_w$  and their relationships with  $c'_w$  and  $v'_w$ .

We begin by considering  $v_w$ . We want to argue that  $v_w = b$ . To see why this is true, assume for contradiction that  $v_w = 1 - b$ . It follows that early in the steps for  $t'$ , node  $w$  switches its consensus value from  $1 - b$  to  $b$ . By the definition of the algorithm, it only does this if at this point in the *ack* steps:  $\hat{c}_w^{(b)} > \hat{c}_w^{(1-b)} \geq c_w$  (the last term follows because  $c_w$  is included in the values considered when defining  $c_w^{(1-b)}$ ). Note, however, that  $c_w^{(b)}$  must be less than  $c$ . If it was greater than or equal to  $c$ , this would imply that a node ended an earlier event with counter  $\geq c$  and value  $b$  – contradicting our assumption that  $t'$  was the earliest such event. If  $c_w^{(b)} < c$  and  $c_w^{(b)} > c_w$ , then  $w$  must increase its  $c_w$  value during this event. But because  $\hat{c}_w^{(b)} > \hat{c}_w^{(1-b)} \geq c_w$ , the only allowable change to  $c_w$  would be to set it to  $\hat{c}_w^{(b)} < c$ . This contradicts the assumption that  $c'_w \geq c$ .

At this checkpoint in our argument we have argued that  $v_w = b$ . We now consider  $c_w$ . If  $c_w \geq c$ , then  $w$  starts  $t'$  with a sufficiently big counter – contradicting the assumption that  $t'$  is the earliest such event. It follows that  $c_w < c$  and  $w$  must increase this value during this event.

There are two ways to increase a counter; i.e., the two conditions in the *if/else-if* statement that follows the  $\text{newm} = \perp$  check. We start with the second condition. If  $\max\{\hat{c}_w^{(b)}, \hat{c}_w^{(1-b)}\} > c_w$ , then  $w$  can set  $c_w$  to this maximum. If this maximum is equal to  $\hat{c}_w^{(b)}$ , then this would imply  $\hat{c}_w^{(b)} \geq c$ . As argued above, however, it would then follow that a node had a counter  $\geq c$  and value  $b$  before  $t'$ . If this is not true, then  $\hat{c}_w^{(1-b)} > c_w^{(b)}$ . If this was the case, however,  $w$  would have adopted value  $1 - b$  earlier in the event, contradicting the assumption that  $v'_w = b$ .

At this next checkpoint in our argument we have argued that  $v_w = b$ ,  $c_w < c$ , and  $w$  increases  $c_w$  to  $c$  through the first condition of the *if/else if*; i.e., it must find that  $\max\{\hat{c}_w^{(b)}, \hat{c}_w^{(1-b)}\} \leq c_w$  and  $m \neq \text{nop}$ . Because this condition only increases the counter by 1, we can further refine our assumption to  $c_w = c - 1$ .

To conclude our argument, consider the implications of the  $m \neq \text{nop}$  component of this condition. It follows that  $t'$  is an  $\text{ack}(m)_w$  for an actual message  $m$ . It cannot be the case that  $m$  is a *decide* message, as  $w$  will not increase its counter on acknowledging a *decide*. Therefore,  $m$  is a counter message. Furthermore, because counter and consensus values are not modified after broadcasting a counter message but before receiving its subsequent acknowledgment, we know  $m = (\text{counter}, id_w, c_w, v_w, *) = (\text{counter}, id_w, c - 1, b, *)$  (we replace the network size estimate with a wildcard here as these estimates could change during this period).

Because  $w$  has an acknowledgment for this  $m$ , by the definition of the model, prior to  $t'$ : every non-crashed node received a counter message with counter  $c - 1$  and consensus value  $b$ . This is exactly the claim we are trying to prove. ◀

Our main safety theorem leverages the above two lemmas to establish that committing to decide  $b$  means that  $b$  dominates the execution. The key idea is that counter values cannot become too stale. By Lemma 3, if some node has a counter  $c$  associated with proposal value  $1 - b$ , then all nodes have seen a counter of size at least  $c - 1$  associated with  $1 - b$ . It follows that if some node thinks  $b$  is far ahead, then all nodes must think  $b$  is far ahead in the race (i.e.,  $b$  dominates). Lemma 2 then establishes that this dominance is permanent – making  $b$  the only possible decision value going forward.

► **Theorem 4.** *The Counter Race Consensus algorithm satisfies validity and agreement.*

**Proof.** Validity follows directly from the definition of the algorithm. To establish agreement, fix some execution  $\alpha$  that includes at least one decision. Let  $t$  be the first *ack* event in  $\alpha$  that is combined with a broadcast of a *decide* message. We call such a step a *pre-decision* step as it prepares nodes to decide in a later step. Let  $u$  be the node at which this *ack* occurs and  $b$  be the value it includes in the *decide* message. Because we assume at least one process decides in  $\alpha$ , we know  $t$  exists. We also know it occurs before any decision.

During the steps associated with  $t$ ,  $u$  sets  $newm \leftarrow (decide, b)$ . This indicates the following is true:  $\hat{c}_u^{(b)} \geq \hat{c}_u^{(1-b)} + 3$ . Based on this condition, we establish two claims about the system at  $t$ , expressed with respect to the value  $\hat{c}_u^{(1-b)}$  during these steps:

- *Claim 1.* The largest counter included with value  $1 - b$  in a counter message broadcast<sup>8</sup> before  $t$  is no more than  $\hat{c}_u^{(1-b)} + 1$ .

Assume for contradiction that before  $t$  some  $v$  broadcast a counter message with value  $1 - b$  and counter  $c > \hat{c}_u^{(1-b)} + 1$ . By Lemma 3, it follows that before  $t$  every non-crashed node receives a counter message with value  $1 - b$  and counter  $c - 1 \geq \hat{c}_u^{(1-b)} + 1$ . This set of nodes includes  $u$ . This contradicts our assumption that at  $t$  the largest counter  $u$  has seen associated with  $1 - b$  is  $\hat{c}_u^{(1-b)}$ .

- *Claim 2.* Before  $t$ , every non-crashed node has sent or received a counter message with value  $b$  and counter at least  $\hat{c}_u^{(1-b)} + 2$ .

By assumption on the values  $u$  has seen at  $t$ , we know that before  $t$  some node  $v$  broadcast a counter message with value  $b$  and counter  $c \geq \hat{c}_u^{(1-b)} + 3$ . By Lemma 3, it follows that before  $t$ , every node has sent or received a counter with value  $b$  and counter  $c - 1 \geq \hat{c}_u^{(1-b)} + 2$ .

Notice that claim 1 combined with claim 2 implies that the execution is  $(b, \hat{c}_u^{(1-b)} + 1)$ -dominated before  $t$ . By Lemma 2, the execution will remain dominated from this point forward. We assume  $t$  was the first pre-decision, and it will lead  $u$  to tell other nodes to decide  $u$  before doing so itself. Other pre-decision steps might occur, however, before all nodes have received  $u$ 's preference for  $b$ . With this in mind, let  $t'$  be any other pre-decision step. Because  $t'$  comes after  $t$  it will occur in a  $(b, \hat{c}_u^{(1-b)} + 1)$ -dominated system. This means that during the first steps of  $t'$ , the node will adopt  $b$  as its value (if it has not already done so), meaning it will also promote  $b$ .

To conclude, we have shown that once any node reaches a pre-decision step for a value  $b$ , then the system is already dominated in favor of  $b$ , and therefore  $b$  is the only possible decision value going forward. Agreement follows directly. ◀

<sup>8</sup> Notice, in these claims, when we say a message is “broadcast” we only mean that the corresponding *bcst* event occurred. We make no assumption on which nodes have so far received this message.

### 3.3 Liveness

We now turn our attention to liveness. Our goal is to prove the following theorem:

► **Theorem 5.** *With high probability, within  $O(n^3 \ln n)$  scheduled *ack* events, every node executing counter race consensus has either crashed, decided, or received a *decide* message. In the limit, this termination condition occurs with probability 1.*

Notice that this theorem does not require a fair schedule. It guarantees its termination criteria (with high probability) after *any*  $O(n^3 \ln n)$  scheduled *ack* events, regardless of *which* nodes these events occur at. Once the system arrives at a state in which every node has either crashed, decided, or received a *decide* message, the execution is now univalent (only one decision value is possible going forward), and each non-crashed node  $u$  will decide after at most two additional *ack* events at  $u$ .<sup>9</sup>

Our liveness proof is longer and more involved than our safety proof. This follows, in part, from the need to introduce multiple technical definitions to help identify the execution fragments sufficiently well-behaved for us to apply our probabilistic arguments. With this in mind, we divide the presentation of our liveness proof into two parts. The first part introduces the main ideas of the analysis and provides a road map of sorts to its component pieces. The second part, which contains the details, can be found in the full paper [42].

#### 3.3.1 Main Ideas

Here we discuss the main ideas of our liveness proof. A core definition used in our analysis is the notion of an  $x$ -run. Roughly speaking, for a given constant integer  $x \geq 2$  and node  $u$ , we say an execution fragment  $\beta$  is an  $x$ -run for some node  $u$ , if it starts and ends with an *ack* event for  $u$ , it contains  $x$  total *ack* events for  $u$ , and no other node has more than  $x$  *ack* events interleaved. We deploy a recursive counting argument to establish that an execution fragment  $\beta$  that contains at least  $n \cdot x$  total *ack* events, must contain a sub-fragment  $\beta'$  that is an  $x$ -run for some node  $u$ .

To put this result to use, we focus our attention on  $(2c + 1)$ -runs, where  $c = 6$  is the constant used in the algorithm definition to define the length of a *group* (see Section 3.1 for a reminder of what a group is and how it is used by the algorithm). A straightforward argument establishes that a  $(2c + 1)$ -run for some node  $u$  must contain at least one *complete group* for  $u$  – that is, it must contain all  $c$  broadcasts of one of  $u$ 's groups.

Combining these observations, it follows that if we partition an execution into *segments* of length  $n \cdot (2c + 1)$ , each such segment  $i$  contains a  $(2c + 1)$ -run for some node  $u_i$ , and each such run contains a complete group for  $u_i$ . We call this complete group the *target group*  $t_i$  for segment  $i$  (if there are multiple complete groups in the run, choose one arbitrarily to be the target).

These target groups are the core unit to which our subsequent analysis applies. Our goal is to arrive at a target group  $t_i$  that is *clean* in the sense that  $u_i$  is *active* during the group (i.e., sends its actual values instead of *nop* placeholders), and all broadcasts that arrive at  $u$  during this group come from *non-active* nodes (i.e., these received messages contain *nop* placeholders instead of values). If we achieve a *clean* group, then it is not hard to show that  $u_i$  will advance its counter at least  $k$  ahead of all other counters, pushing all other nodes into the termination criteria guaranteed by Theorem 5.

<sup>9</sup> In the case where  $u$  receives a *decide* message, the first *ack* might correspond to the message it was broadcasting when the *decide* arrived, and the second *ack* corresponds to the *decide* message that  $u$  itself will then broadcast. During this second *ack*,  $u$  will decide and halt.

To prove clean groups are sufficiently likely, our analysis must overcome two issues. The first issue concerns network size estimations. Fix some target group  $t_i$ . Let  $P_i$  be the nodes from which  $u_i$  receives at least one message during  $t_i$ . If all of these nodes have a network size estimate of at least  $n_i = |P_i|$  at the start of  $t_i$ , we say the group is *calibrated*. We prove that if  $t_i$  is calibrated, then it is clean with a probability in  $\Omega(1/n)$ .

The key, therefore, is proving most target groups are calibrated. To do so, we note that if some  $t_i$  is not calibrated, it means at least one node used an estimate strictly less than  $n_i$  when it probabilistically defined *active* at the beginning of this group. During this group, however, all nodes will receive broadcasts from at least  $n_i$  unique nodes, increasing all network estimates to size at least  $n_i$ .<sup>10</sup> Therefore, each target group that fails to be calibrated increases the minimum network size estimate in the system by at least 1. It follows that at most  $n$  target groups can be non-calibrated.

The second issue concerns probabilistic dependencies. Let  $E_i$  be the event that target group  $t_i$  is clean and  $E_j$  be the event that some other target group  $t_j$  is clean. Notice that  $E_i$  and  $E_j$  are not necessarily independent. If a node  $u$  has a group that overlaps both  $t_i$  and  $t_j$ , then its probabilistic decision about whether or not to be active in this group impacts the potential cleanliness of both  $t_i$  and  $t_j$ .

Our analysis tackles these dependencies by identifying a subset of target groups that are pairwise independent. To do so, roughly speaking, we process our target groups in order. Starting with the first target group, we mark as unavailable any future target group that overlaps this first group (in the sense described above). We then proceed until we arrive at the next target group *not* marked unavailable and repeat the process. Each available target group marks at most  $O(n)$  future groups as unavailable. Therefore, given a sufficiently large set  $T$  of target groups, we can identify a subset  $T'$ , with a size in  $\Omega(|T|/n)$ , such that all groups in  $T'$  are pairwise independent.

We can now pull together these pieces to arrive at our main liveness complexity claim. Consider the first  $O(n^3 \ln n)$  *ack* events in an execution. We can divide these into  $O(n^2 \ln n)$  segments of length  $(2c + 1)n \in \Theta(n)$ . We now consider the target groups defined by these segments. By our above argument, there is a subset  $T'$  of these groups, where  $|T'| \in \Omega(n \ln n)$ , and all target groups in  $T'$  are mutually independent. At most  $n$  of these remaining target groups are not calibrated. If we discard these, we are left with a slightly smaller set, of size still  $\Omega(n \ln n)$ , that contains only calibrated and pairwise independent target groups.

We argued that each calibrated group has a probability in  $\Omega(1/n)$  of being clean. Leveraging the independence between our identified groups, a standard concentration analysis establishes with high probability in  $n$  that at least one of these  $\Omega(n/\ln n)$  groups is clean – satisfying the Theorem statement.

### 3.4 Removing the Assumption of Unique IDs

The consensus algorithm described in this section assumes unique IDs. We now show how to eliminate this assumption by describing a strategy that generates unique IDs w.h.p., and discuss how to use this as a subroutine in our consensus algorithm.

We make use of a simple tiebreaking mechanism as follows: Each node  $u$  proceeds by iteratively extending a (local) random bit string that eventually becomes unique among the nodes. Initially,  $u$  broadcasts bit  $b_1$ , which is initialized to 1 (at all nodes), and each time  $u$

---

<sup>10</sup>This summary is eliding some subtle details tackled in the full analysis concerning which broadcasts are guaranteed to be received during a target group. But these details are not important for understanding the main logic of this argument.

samples a new bit  $b$ , it appends  $b$  to its current string and broadcasts the result. For instance, suppose that  $u$ 's most recently broadcast bit string is  $b_1 \dots b_i$ . Upon receiving  $ack(b_1 \dots b_i)$ , node  $u$  checks if it has received a message identical to  $b_1 \dots b_i$ . If it did not receive such a message, then  $u$  adopts  $b_1 \dots b_i$  as its ID and stops. Otherwise, some distinct node must have sampled the same sequence of bits as  $u$  and, in this case, the ID  $b_1 \dots b_i$  is considered to be already taken. (Note that nodes do not take receive events for their own broadcasts.) Node  $u$  continues by sampling its  $(i + 1)$ -th bit  $b_{i+1}$  uniformly at random, and then broadcasts the string  $b_1 \dots b_i b_{i+1}$ , and so forth. In the full paper [42], we prove the following result and describe how to combine it with our consensus algorithm:

► **Theorem 6.** *Consider an execution  $\alpha$  of the tiebreaking algorithm. Let  $t_u$  be an event in the message schedule  $msg[\alpha]$  such that node  $u$  is scheduled for  $\Omega(\log n)$  ack events before  $t_u$ . Then, for each correct node  $u$ , it holds that  $u$  has a unique ID of  $O(\log n)$  bits with high probability at  $t_u$ .*

## 4 Almost-Everywhere Agreement

In the previous section, we showed how to solve consensus in  $O(n^3 \log n)$  events. Here we show how to improve this bound by a near linear factor by loosening the agreement guarantees. In more detail, we consider a weaker variant of consensus, introduced in [18], called *almost-everywhere agreement*. This variation relaxes the agreement property of consensus such that  $o(n)$  nodes are allowed to decide on conflicting values so long as the remaining nodes all decide the same value. For many problems that use consensus as a subroutine, this relaxed agreement property is sufficient.

In more detail, we present an algorithm for solving almost-everywhere agreement in the abstract MAC layer model when nodes start with arbitrary (not necessarily binary) input values. The algorithm consists of two phases. We present the pseudo code in the full paper [42].

**Phase 1.** In this phase, nodes try to obtain an estimate of the network size by performing local coin flipping experiments. Each node  $u$  records the number of times that its coin comes up tails before observing the first heads in a variable  $X$ . Then,  $u$  broadcasts its value of  $X$  once, and each node updates  $X$  to the highest outcome that it has seen until it receives the *ack* for its broadcast. In our analysis, we show that, by the end of Phase 1, variable  $X$  is an approximation of  $\log_2(n)$  with an additive  $O(\log \log n)$  term, for all nodes in a large set called *EST*, and hence  $N := 2^X$  is a good approximation of the network size  $n$  for any node in *EST*.

**Phase 2.** Next, we use  $X$  and  $N$  as parameters of a randomly rotating leader election procedure. Each node decides after  $T = \Theta(N \log^3(N) \log \log(N))$  rounds. (Note that due to the asynchronous nature of the abstract MAC layer model, different nodes might be executing in different rounds at the same point in time.) We now describe the sequence of steps comprising a round in more detail: A node  $u$  becomes active with probability  $1/N_u$  at the start of each round.<sup>11</sup> If it is active, then  $u$  samples a random rank  $\rho$  from a range polynomial in  $X_u$ , and broadcasts a message  $\langle r, \rho, val \rangle$  where *val* refers to its current consensus input value. To ensure that the scheduler cannot derive any information about

<sup>11</sup> We use the convention  $N_u$  when referring to the local variable  $N$  of a specific node  $u$ .



whether a node is active in a round, inactive nodes simply broadcast a dummy message with infinite rank. While an (active or inactive) node  $v$  waits for its *ack* for round  $r$ , it keeps track of all received messages and defers processing of a message sent by a node in some round  $r' > r$  until the event in which  $v$  itself starts round  $r'$ . On the other hand, if a received message was sent in  $r' < r$ , then  $v$  simply discards that late message as it has already completed  $r'$ . Node  $v$  uses the information of messages originating from the same round  $r$  to update its consensus input value, if it receives such a message from an active node that has chosen a smaller rank than its own. (Recall that inactive nodes have infinite rank.) After  $v$  has finished processing the received messages, it moves on the next round.

We first provide some intuition why it is insufficient to focus on a round  $r$  where the “earliest” node is also active: Ideally, we want the node  $w_1$  that is the first to receive its *ack* for round  $r$  to be active *and* to have the smallest rank among all active nodes in round  $r$ , as this will force all other (not-yet decided) nodes to adopt  $w_1$ ’s value when receiving their own round  $r$  *ack*, ensuring a.e. agreement. However, it is possible that  $w_1$  and also the node  $w_2$  that receives its round  $r$  *ack* right after  $w_1$ , are among the few nodes that ended up with a small (possibly constant) value of  $X$  after Phase 1. We cannot use the size of  $EST$  to reason about this probability, as some nodes are much likelier to be in  $EST$  than others, depending on the schedule of events in Phase 1. In that case, it could happen that both  $w_1$  and  $w_2$  become active and choose a rank of 1. Note that it is possible that the receive steps of their broadcasts are scheduled such that roughly half of the nodes receive  $w_1$ ’s message before  $w_2$ ’s message, while the other half receive  $w_2$ ’s message first. If  $w_1$  and  $w_2$  have distinct consensus input values, then it can happen that both consensus values gain large support in the network as a result.

To avoid this pitfall, we focus on a set of rounds where all nodes *not* in  $EST$  have already terminated Phase 2 (and possibly decided on a wrong value): from that point onwards, only nodes with sufficiently large values of  $X$  and  $N$  keep trying to become active. We can show that every node in  $EST$  has a probability of at least  $\Omega(1/(n \log n))$  to become active and a probability of  $\Omega(1/\log n)$  to have chosen the smallest rank among all nodes that are active in the same round. Thus, when considering a sufficiently large set of rounds, we can show that the event, where the first node in  $EST$  that receives its *ack* in round  $r$  becomes active and also chooses a rank smaller than the rank of any other node active in the same round, happens with probability  $1 - o(1)$ .

In the full paper [42], we formalize the above discussion by proving the following main theorem regarding this algorithm:

► **Theorem 7.** *With high probability, the following two properties are true of our almost-everywhere consensus algorithm: (1) within  $O(n^2 \log^4 n \cdot \log \log n)$  scheduled *ack* events, every node has either crashed, decided, or will decide after it is next scheduled; (2) all but at most  $o(n)$  nodes that decide, decide the same value.*

## 5 Lower Bound

We conclude our investigation by showing a separation between the abstract MAC layer model and the related asynchronous message passing model. In more detail, we prove below that fault-tolerant consensus with constant success probability is impossible in a variation of the asynchronous message passing model where nodes are provided only a constant-fraction approximation of the network size and communicate using (blind) broadcast. This bound holds even if we assume no crashes and provide nodes unique ids from a small set. Notice, in the abstract MAC layer model, we solve consensus with broadcast under the harsher

---

**Algorithm 3** Almost-everywhere agreement in the abstract MAC layer model. Code for node  $u$ .

---

```

1:  $val \leftarrow$  consensus input value
2: ▷ Phase 1
3: initialize  $X \leftarrow 0$ ;  $R \leftarrow \emptyset$ 
4: while  $flip\_coin() = heads$  do
5:    $X \leftarrow X + 1$ 
6: bcast( $X$ )
7: while waiting for  $ack$  do
8:   add received messages to  $R$ 
9:  $X \leftarrow \max(R \cup \{X\})$ 
10:  $N \leftarrow 2^X$ 
11: ▷ Phase 2
12:  $T \leftarrow \lceil cN \log^3(N) \log \log(N) \rceil$ , where  $c$  is a sufficiently large constant.
13: initialize array of sets  $R[1], \dots, R[T] \leftarrow \emptyset$ 
14: for  $i \leftarrow 1, \dots, T$  do ▷ Start of round  $i$  at  $u$ 
15:    $u$  becomes active with probability  $\frac{1}{N}$ 
16:   if  $u$  is active then
17:      $\rho \leftarrow$  unif. at random sampled integer from  $[1, X^4]$ 
18:   else
19:      $\rho \leftarrow \infty$ 
20:   bcast( $\langle i, \rho, val \rangle$ )
21:   while waiting for  $ack$  do
22:     add received messages to  $R[i]$ 
23:     for each message  $m = \langle i', \rho', val' \rangle \in R[i]$  do
24:       if  $i' = i$  and  $\rho' < \rho$  then ▷ Received message from node with smaller rank
25:          $val \leftarrow val'$ 
26:       else if  $i' > i$  then ▷ Received message from node active in future round
27:         add  $m$  to  $R[i']$ 
28:       else
29:         discard message  $m$ 
30: decide on  $val$ 

```

---

constraints of no network size information, no ids, and crash failures. The difference is the fact that the broadcast primitive in the abstract MAC layer model includes an acknowledgment. This acknowledgment is therefore revealed to be the crucial element of the our model that allows algorithms to overcome lack of network information. We note that this bound is a generalization of the result from [1], which proved deterministic consensus was impossible under these constraints. In the full paper [42], we show that, for any given randomized algorithm we can construct scenarios that are indistinguishable for the nodes, thus causing conflicting decisions.

► **Theorem 8.** *Consider an asynchronous network of  $n$  nodes that communicate by broadcast and suppose that nodes are unaware of the network size  $n$ , but have knowledge of an integer that is guaranteed to be a 2-approximation of  $n$ . No randomized algorithm can solve binary consensus with a probability of success of at least  $1 - \epsilon$ , for any constant  $\epsilon < 2 - \sqrt{3}$ . This holds even if nodes have unique identifiers chosen from a range of size at least  $2n$  and all nodes are correct.*

---

References

---

- 1 Mohssen Abboud, Carole Delporte-Gallet, and Hugues Fauconnier. Agreement without knowing everybody: a first step to dynamicity. In *Proceedings of the International Conference on New Technologies in Distributed Systems*, 2008.
- 2 Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Failure detection and consensus in the crash-recovery model. *Distributed computing*, 13(2):99–125, 2000.
- 3 Eduardo AP Alchieri, Alysson Neves Bessani, Joni da Silva Fraga, and Fabíola Greve. Byzantine consensus with unknown participants. In *Proceedings of the International Conference on the Principles of Distributed Systems*. Springer, 2008.
- 4 Khaled Alekeish and Paul Ezhilchelvan. Consensus in sparse, mobile ad hoc networks. *IEEE Transactions on Parallel and Distributed Systems*, 23(3):467–474, 2012.
- 5 James Aspnes. Fast deterministic consensus in a noisy environment. *Journal of Algorithms*, 45(1):16–39, 2002.
- 6 Hagit Attiya, Alla Gorbach, and Shlomo Moran. Computing in totally anonymous asynchronous shared memory systems. *Information and Computation*, 173(2):162–183, 2002.
- 7 John Augustine, Gopal Pandurangan, Peter Robinson, and Eli Upfal. Distributed agreement in dynamic peer-to-peer networks. *J. Comput. Syst. Sci.*, 81(7):1088–1109, 2015. doi:10.1016/j.jcss.2014.10.005.
- 8 R. Bar-Yehuda, O. Goldreich, and A. Itai. On the Time Complexity of Broadcast in Radio Networks: an Exponential Gap Between Determinism and Randomization. In *Proceedings of the ACM Conference on Distributed Computing*, 1987.
- 9 Michael Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *Proceedings of the ACM Conference on Distributed Computing*, pages 27–30. ACM, 1983.
- 10 François Bonnet and Michel Raynal. Anonymous Asynchronous Systems: the Case of Failure Detectors. In *Proceedings of the International Conference on Distributed Computing*, 2010.
- 11 David Cavin, Yoav Sasson, and André Schiper. Consensus with unknown participants or fundamental self-organization. In *ADHOC-NOW*, 2004.
- 12 Tushar Deepak Chandra. Polylog randomized wait-free consensus. In *Proceedings of the ACM Conference on Distributed Computing*, 1996.
- 13 Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.
- 14 Alejandro Cornejo, Nancy Lynch, Saira Viqar, and Jennifer L Welch. Neighbor Discovery in Mobile Ad Hoc Networks Using an Abstract MAC Layer. In *Proceedings of the Annual Allerton Conference on Communication, Control, and Computing*, 2009.
- 15 Alejandro Cornejo, Saira Viqar, and Jennifer L Welch. Reliable Neighbor Discovery for Mobile Ad Hoc Networks. *Ad Hoc Networks*, 12:259–277, 2014.
- 16 A. Czumaj and W. Rytter. Broadcasting Algorithms in Radio Networks with Unknown Topology. *Journal of Algorithms*, 60:115–143, 2006.
- 17 Sebastian Daum, Seth Gilbert, Fabian Kuhn, and Calvin Newport. Broadcast in the Ad Hoc SINR Model. In *Proceedings of the International Conference on Distributed Computing*, 2013.
- 18 Cynthia Dwork, David Peleg, Nicholas Pippenger, and Eli Upfal. Fault tolerance in networks of bounded degree. *SIAM Journal on Computing*, 17(5):975–988, 1988.
- 19 Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2), 1985.
- 20 L. Gasiencic, D. Peleg, and Q. Xin. Faster Communication in Known Topology Radio Networks. *Distributed Computing*, 19(4):289–300, 2007.

- 21 O. Goussevskaia, R. Wattenhofer, M.M. Halldorsson, and E. Welzl. Capacity of Arbitrary Wireless Networks. In *Proceedings of the IEEE International Conference on Computer Communications*, 2009.
- 22 Fabiola Greve and Sebastien Tixeuil. Knowledge connectivity vs. synchrony requirements for fault-tolerant agreement in unknown networks. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks*, 2007.
- 23 Rachid Guerraoui, Michel Hurfin, Achour Mostéfaoui, Riucarlos Oliveira, Michel Raynal, and André Schiper. Consensus in asynchronous distributed systems: A concise guided tour. *Advances in Distributed Systems, Lecture Notes in Computer Science*, 1752:33–47, 2000.
- 24 Rachid Guerraoui and Andre Schiper. Consensus: the big misunderstanding [distributed fault tolerant systems]. In *Proceedings of the IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*, 1997.
- 25 Rachid Guerraoui and André Schiper. The generic consensus service. *IEEE Transactions on Software Engineering*, 27(1):29–41, 2001.
- 26 Magnus M. Halldorsson and Pradipta Mitra. Wireless Connectivity and Capacity. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, 2012.
- 27 Tomasz Jurdzinski, Dariusz R. Kowalski, Michal Rozanski, and Grzegorz Stachowiak. Distributed Randomized Broadcasting in Wireless Networks under the SINR Model. In *Proceedings of the International Conference on Distributed Computing*, 2013.
- 28 Tomasz Jurdziński and Grzegorz Stachowiak. Probabilistic Algorithms for the Wakeup Problem in Single-Hop Radio Networks. In *Algorithms and Computation*, pages 535–549. Springer, 2002.
- 29 Majid Khabbazian, Fabian Kuhn, Dariusz Kowalski, and Nancy Lynch. Decomposing Broadcast Algorithms Using Abstract MAC Layers. In *Proceedings of the International Workshop on the Foundations of Mobile Computing*, 2010.
- 30 Majid Khabbazian, Fabian Kuhn, Nancy Lynch, Muriel Medard, and Ali ParandehGheibi. MAC Design for Analog Network Coding. In *Proceedings of the International Workshop on the Foundations of Mobile Computing*, 2011.
- 31 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.
- 32 D.R. Kowalski and A. Pelc. Broadcasting in Undirected Ad Hoc Radio Networks. *Distributed Computing*, 18(1):43–57, 2005.
- 33 Fabian Kuhn, Nancy Lynch, and Calvin Newport. The Abstract MAC Layer. In *Proceedings of the International Conference on Distributed Computing*, 2009.
- 34 Fabian Kuhn, Nancy Lynch, and Calvin Newport. The Abstract MAC Layer. *Distributed Computing*, 24(3-4):187–206, 2011.
- 35 Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.
- 36 Nancy A Lynch. *Distributed algorithms*. Morgan Kaufmann, 1996.
- 37 Thomas Moscibroda. The Worst-Case Capacity of Wireless Sensor Networks. In *Proceedings of the ACM/IEEE International Conference on Information Processing in Sensor Networks*, 2007.
- 38 Thomas Moscibroda and Roger Wattenhofer. Maximal Independent Sets in Radio Networks. In *Proceedings of the ACM Conference on Distributed Computing*, 2005.
- 39 Thomas Moscibroda and Roger Wattenhofer. The Complexity of Connectivity in Wireless Networks. In *Proceedings of the IEEE International Conference on Computer Communications*, 2006.
- 40 Achour Mostefaoui and Michel Raynal. Solving consensus using Chandra-Touegs unreliable failure detectors. *Lecture Notes in Computer Science*, 1693:49–63, 1999.

- 41 Calvin Newport. Consensus with an Abstract MAC Layer. In *Proceedings of the ACM Conference on Distributed Computing*, 2014.
- 42 Calvin Newport and Peter Robinson. Fault-Tolerant Consensus with an Abstract MAC Layer. Technical report, <https://www.cas.mcmaster.ca/robinson/random-aml.pdf>, 2018.
- 43 Eric Ruppert. The Anonymous Consensus Hierarchy and Naming Problems. In *Proceedings of the International Conference on Principles of Distributed Systems*, 2007.
- 44 Andre Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10(3):149–157, 1997.
- 45 Einar W Vollset and Paul D Ezhilchelvan. Design and performance-study of crash-tolerant protocols for broadcasting and reaching consensus in MANETs. In *IEEE Symposium on Reliable Distributed Systems*, 2005.
- 46 Weigang Wu, Jiannong Cao, and Michel Raynal. Eventual clusterer: A modular approach to designing hierarchical consensus protocols in manets. *IEEE Transactions on Parallel and Distributed Systems*, 20(6):753–765, 2009.

# Randomized $(\Delta + 1)$ -Coloring in $O(\log^* \Delta)$ Congested Clique Rounds

**Merav Parter**

Weizmann IS, Rehovot, Israel

merav.parter@weizmann.ac.il

**Hsin-Hao Su**

UNC-Charlotte, North Carolina, USA

hsinhao@csail.mit.edu

---

## Abstract

$(\Delta + 1)$ -vertex coloring is one of the most fundamental symmetry breaking graph problems, receiving tremendous amount of attention over the last decades. We consider the congested clique model where in each round, every pair of vertices can exchange  $O(\log n)$  bits of information.

In a recent breakthrough, Yi-Jun Chang, Wenzheng Li, and Seth Pettie [CLP-STOC'18] presented a randomized  $(\Delta + 1)$ -list coloring algorithm in the LOCAL model that works in  $O(\log^* n + \text{Det}_{\text{deg}}(\log \log n))$  rounds, where  $\text{Det}_{\text{deg}}(n')$  is the deterministic LOCAL complexity of  $(\text{deg} + 1)$ -list coloring algorithm on  $n'$ -vertex graphs. Unfortunately, the CLP algorithm uses large messages and hence cannot be efficiently implemented in the congested clique model when the maximum degree  $\Delta$  is large (in particular, when  $\Delta = \omega(\sqrt{n})$ ).

Merav Parter [P-ICALP'18] recently provided a randomized  $(\Delta + 1)$ -coloring algorithm in  $O(\log \log \Delta \cdot \log^* \Delta)$  congested clique rounds based on a careful partitioning of the input graph into almost-independent subgraphs with maximum degree  $\sqrt{n}$ . In this work, we significantly improve upon this result and present a randomized  $(\Delta + 1)$ -coloring algorithm with  $O(\log^* \Delta)$  rounds, with high probability. At the heart of our algorithm is an adaptation of the CLP algorithm for coloring a subgraph with  $o(n)$  vertices and maximum degree  $\Omega(n^{5/8})$  in  $O(\log^* \Delta)$  rounds. The approach is built upon a combination of techniques, this includes: the graph sparsification of [Parter-ICALP'18], and a palette sampling technique adopted to the CLP framework.

**2012 ACM Subject Classification** Theory of computation  $\rightarrow$  Distributed algorithms

**Keywords and phrases** Distributed Graph Algorithms, Coloring, congested clique

**Digital Object Identifier** 10.4230/LIPIcs.DISC.2018.39

## 1 Introduction & Related Work

Graph coloring is one of the most central symmetry breaking graph problems, and as such it has been receiving much attention. In the distributed setting, vertex coloring as many other symmetry breaking tasks are mostly studied in the LOCAL model where the messages sent in a given round are allowed to be arbitrarily large. Indeed, the recent breakthrough results for vertex coloring [5, 8] and MIS [6] use large messages, potentially of size  $\Omega(n)$ . This poses a strong motivation for studying these problems in bandwidth restricted models.

The congested clique model of distributed computing was introduced by Lotker, Pavlov, Patt-Shamir, and Peleg [14]. In this model, the communication is all-to-all, and per round, each node can send  $O(\log n)$  bits to each other node. One can view the congested clique as being orthogonal to the LOCAL model: the former abstracts away locality (each node is one-hop from each other node), and the latter abstracts away congestion. The fact that the



© Merav Parter and Hsin-Hao Su;

licensed under Creative Commons License CC-BY

32nd International Symposium on Distributed Computing (DISC 2018).

Editors: Ulrich Schmid and Josef Widder; Article No. 39; pp. 39:1–39:18

Leibniz International Proceedings in Informatics



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

congested clique model escapes any locality based lower bounds (e.g., [12]) makes it very attractive for studying the net (or clean) effect of bandwidth limitation on local computation. Ghaffari [6] posed the following question:

*Can we solve the classic local problems: MIS, maximal matching,  $(\Delta + 1)$ -vertex-coloring, or  $(2\Delta - 1)$ -edge-coloring – much faster in the congested clique model?*

This question was answered in the affirmative in [6] for the MIS problem by presenting a randomized MIS algorithm that works in  $\tilde{O}(\sqrt{\log \Delta})$  congested clique rounds. Very recently, this bound was further improved to  $O(\log \log \Delta)$  rounds by Ghaffari et al. [7]. The latter work also improved the round complexity of other local problems (e.g., maximal matching) in the related model of Massively Parallel Computation (MPC), which is arguably the most popular model framework for large-scale computation (e.g., MapReduce, Hadoop and Spark [7])<sup>1</sup>. The problem of  $(\Delta + 1)$ -vertex coloring in the congested clique model was recently studied by [15], presenting a randomized algorithm with  $O(\log \log \Delta \cdot \log^* \Delta)$  rounds. We also note that earlier works in the congested clique model considered weaker versions of MIS and coloring, see [3, 10, 9].

An orthogonal line of research considers the power of all-to-all communication for *deterministic* local algorithms. Censor et al. [4] presented a quite general scheme for derandomization in the congested clique model by combining the methods of bounded independence with efficient computation of the conditional expectation. They provided a deterministic MIS algorithm that works in  $O(\log \Delta \cdot \log n)$  congested clique rounds. [15] recently showed a deterministic  $(\Delta + 1)$ -coloring in  $O(\log \Delta)$  rounds. Barenboim and Khazanov [1] presented improved deterministic local algorithms as a function of the graph's *arboricity*.

## 1.1 Our Result and Technical Overview

Our main result is an adaptation of the Chang-Li-Pettie (CLP) algorithm to the congested clique model:

► **Theorem 1.** *There is a randomized algorithm that computes a  $(\Delta + 1)$ -coloring in  $O(\log^* \Delta)$  rounds of the congested clique model, with high probability<sup>2</sup>.*

Our algorithm is based on the recent graph sparsification technique of [15] combined with modified versions of the key coloring algorithms of [5]. The starting observation made in [15] is that the CLP algorithm can be simulated in  $O(\log^* \Delta)$  congested clique rounds when  $\Delta = O(\sqrt{n})$ . To handle a graph with arbitrarily large degrees, [15] applies  $O(\log \log \Delta)$  phases of a graph specification procedure, until all unsolved pieces to be colored are subgraphs with maximum degree  $O(\sqrt{n})$  and hence can be colored in  $O(\log^* \Delta)$  rounds by employing the CLP algorithm. In this work, we significantly improve upon this approach, and break this  $\sqrt{n}$ -barrier by modifying the key coloring procedures of the CLP algorithm. This modified CLP allows us to color  $o(n)$ -vertex subgraphs with maximum degree  $\Omega(n^{5/8})$  in  $O(\log^* \Delta)$  rounds. The high-level description of our algorithm is as follows: Given a graph  $G$  with maximum degree  $\Delta$ ,  $G$  is carefully partitioned into: (i) a collection of  $\Delta^{1/4}$  *independent* subgraphs  $G_i$  with maximum degree  $\Delta(G_i) = O(\Delta^{3/4})$  and (ii) a left-over subgraph  $G^*$  with  $N = \tilde{O}(n/\Delta^{3/8})$  vertices and maximum degree  $\Delta^* = \tilde{O}(\Delta^{5/8})$ . The improvement over [15] comes from the fact that our algorithm applies only a constant number of phases of the graph sparsification procedure (for coloring the subgraphs of (i)), rather than  $O(\log \log \Delta)$

<sup>1</sup> A general simulation result between these models has been recently provided by [2].

<sup>2</sup> As usual, by high probability we mean  $1 - 1/n^c$  for some constant  $c \geq 1$ .



as in [15]. The first collection of  $G_i$  subgraphs are treated as independent in the sense that each subgraph  $G_i$  is given a distinct set of  $\Delta(G_i) + 1$  colors in  $[1, \Delta + 1]$  and thus these subgraphs can be colored simultaneously within  $O(\log^* \Delta)$  rounds using the [15] algorithm. The partitioning into these graphs is done in a careful manner so that allocating  $\Delta(G_i) + 1$  colors to each of them still respects the total number  $\Delta + 1$  of allowed colors. The main challenge is in coloring the left-over subgraph  $G^*$  overcoming the fact that its degree is  $\Omega(n^{1/2+\epsilon})$ . Unlike the previous  $\Delta^{1/4}$  subgraphs, here we ran out of budget of free colors and hence this subgraph should be colored using a list-coloring algorithm only after all other subgraphs  $G_i$  are colored. As will be described later on, the CLP algorithm is based on the knowledge of the second neighborhood of the vertices (which can be obtained in 2 rounds in the LOCAL model). In our setting, the degree of  $G^*$  is too large for allowing the vertices collecting their entire second neighborhoods. The key challenge is in bypassing all critical points of the CLP that are based on this kind of knowledge. To do that we employ several congested clique routing techniques combined with a palette sampling technique adopted to the CLP framework.

**Technical History of Coloring and a Short Exposition of the CLP Algorithm.** The first step for breaking the KMW lower bound [12] was made by Schneider and Wattenhofer [17] who showed that when vertices have sufficiently many excess colors in their palette<sup>3</sup> the graph can be colored considerably faster. Elkin, Pettie and Su made the first connection between the above observation to a concrete structural graph property. Specifically, they showed that an  $(1 - \epsilon)$ -sparse graph<sup>4</sup> can be transformed within a single round into a graph in which each vertex has  $\Omega(\epsilon\Delta)$  many access colors in its palette. This graph characterization was the basis of the decomposition technique by Harris, Schneider and Su [8] which we describe next.

For a given parameter  $\epsilon \in (0, 1)$ , [8] decomposed the input graph  $G$  into an  $\epsilon$ -sparse and an  $\epsilon$ -dense subgraphs. To color the sparse subgraph, [8] employed the approach of [17], and their key contribution is a novel dense coloring procedure. [8] showed that the dense subgraph consists of a collection of almost-clique components with weak diameter 2. Informally, the dense coloring procedure was based on having a leader in each such almost-clique; the leader collected the palettes and neighbor-lists of all the vertices in its clique, and colored them locally such that most of these colors are legal.

The approach of [5] is based on a hierarchical version of [8] with  $O(\log \log \Delta)$  sparsity levels. This partitions the vertices in the graph into  $O(\log \log \Delta)$  layers. The algorithm further groups these layers into  $O(\log^* \Delta)$  *Strata* where all layers in a given strata are colored simultaneously. When coloring vertices in each stratum, the algorithm applies modified versions of the dense coloring procedure in [8], which are based upon collecting the information of each almost-clique to a leader. Since the diameter of the almost-clique is shown to be 2, this information can be easily collected in the LOCAL model, but might take many rounds, when the message size is restricted to  $O(\log n)$  bits.

**How to Break the  $\Delta = O(\sqrt{n})$  Barrier?** The main obstacle for simulating the CLP algorithm when  $\Delta = \omega(\sqrt{n})$  in the congested clique model concerns two critical places in the CLP algorithm, where vertices collect  $O(\Delta^2)$  messages (e.g., their second neighborhood). The first place is for (1) defining the  $\epsilon$ -dense subgraph and the second place is for (2) coloring the

<sup>3</sup> The excess of colors of vertex  $v$  is the number of colors in  $v$ 's palette minus the number of uncolored neighbors of  $v$ .

<sup>4</sup> I.e., a graph in which in the 2-hop neighborhood of each vertex, there are only  $(1 - \epsilon^2)\Delta^2$  triangles.

dense regions by collecting palettes to the leader of each almost-clique. These steps could be implemented in  $O(1)$  rounds only when  $\Delta = O(\sqrt{n})$ , but require polynomially many rounds for  $\Delta = \Omega(n^{1/2+\epsilon})$ . Thus breaking this  $\sqrt{n}$  barrier calls for alternative procedures that avoid learning the 2-neighborhoods of the vertices. We now elaborate about these technicalities and our approach to handle them.

To compute the dense subgraph, in the CLP algorithm every vertex  $v$  computes the number of mutual neighbors with each of its neighbors, i.e., it computes  $|N(v) \cap N(u)|$  for every  $u \in N(v)$ . Indeed this can be easily done if a vertex knows the neighbors of its neighbors. In our setting, we use the fact that the left-over subgraph  $G^*$  has  $N = \tilde{O}(n/\Delta^{3/8})$  vertices and allocate each vertex  $v$  in  $G^*$  a subset of  $r = n/N$  relay vertices that share the computational load of vertex  $v$ . Specifically, in our scheme, each relay vertex of  $v$  is responsible for computing the intersection size  $|N(v) \cap N(u)|$  for a subset of  $\Delta/r$  neighbors  $u \in N(v)$ , it would then communicate the outcome of this computation to  $v$ .

The second spot in which the CLP algorithm collects the second neighborhood of (some of the vertices) is in the dense coloring procedures. Our key technical contribution is in showing that it is sufficient for each almost-clique member to send to its leader a random sample of  $O(\sqrt{\Delta})$  colors in its palette rather than its *entire*  $\Theta(\Delta)$ -size palette. Since each almost-clique contains  $O(\Delta)$  vertices and since the maximum degree of the subgraph  $G^*$  is  $O(n^{2/3})$ , each leader is a target of  $O(n)$  message. Such a routing pattern can then be implemented  $O(1)$  rounds using the routing algorithm of Lenzen. The technical challenge is in showing the even-though the leader of an almost-clique  $C$  knows neither internal edges in  $C$  nor the complete individual palettes of the vertices in  $C$ , it can still mimic the CLP procedures, and color its clique vertices with almost the same success rate.

**Lenzen's Routing Algorithm.** Almost all congested clique algorithms are based on the Lenzen's routing algorithm [13]. This routing algorithm schedules in  $O(1)$  rounds the common communication setting where each vertex needs to send and receive  $O(n)$  messages.

## 2 Coloring Most Vertices Through Graph Sparsification

We make use of the following version of Chernoff bound:

► **Theorem 2** (Simple Corollary of Chernoff Bound). *Suppose  $X_1, X_2, \dots, X_\ell \in [0, 1]$  are independent random variables, and let  $X = \sum_{i=1}^{\ell} X_i$  and  $\mu = \mathbb{E}[X]$ . If  $\mu \geq 5 \log n$ , then with probability at least  $1 - 1/n^2$ ,  $X \in \mu \pm \sqrt{5\mu \log n}$ , and if  $\mu < 5 \log n$ , then  $X \leq \mu + 5 \log n$ .*

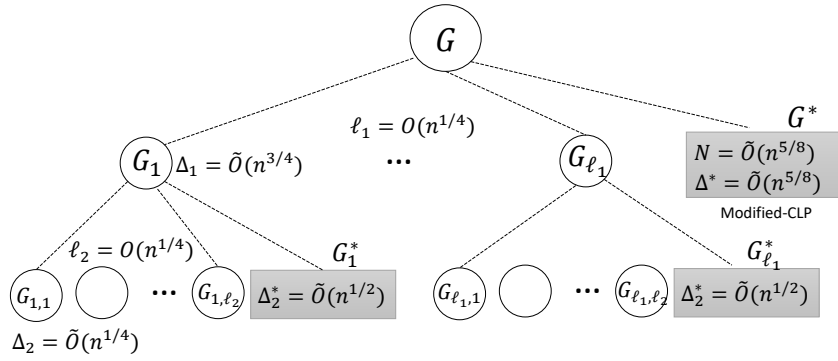
**The Algorithm.** The graph  $G$  is partitioned into  $\ell = \lceil \Delta^{1/4} \rceil$  subgraphs  $G_1, \dots, G_\ell$ , and a left-over subgraph  $G^*$ . This is done by dividing the vertices into  $\ell + 1$  subsets  $V_1, \dots, V_\ell, V^*$  by letting each vertex join  $V_i$  with probability

$$p_i = 1/\ell - 2\sqrt{5 \log n}/\sqrt{\Delta \cdot \ell},$$

for every  $i \in \{1, \dots, \ell\}$ , and joining  $V^*$  with the remaining probability of

$$p^* = 2\sqrt{5 \log n} \cdot \sqrt{\ell/\Delta} = \Theta(\log n/\Delta^{3/8}).$$

For every  $i \in \{1, \dots, \ell, *\}$ , let  $G_i = G[V_i]$  be the induced subgraph and let  $\Delta_i$  be the maximum degree of  $G_i$ . Using Chernoff bound of Theorem 2, for every  $i \in \{1, \dots, \ell\}$ , w.h.p., it holds:  $\Delta_i \leq \Delta/\ell - 2\sqrt{5 \log n} \cdot \sqrt{\Delta/\ell} + \sqrt{5 \log n} \cdot \sqrt{\Delta/\ell} \leq \Delta/\ell - 1$ .



**Figure 1** For ease of presentation, we omit log factors from considerations. The graph  $G$  is partitioned into  $\tilde{O}(n^{1/4})$  subgraphs  $G_i$  and a left-over subgraph  $G^*$ . Each subgraph  $G_i$  has maximum degree  $\tilde{O}(n^{1/4})$  and it is further divided into  $\tilde{O}(n^{1/4})$  subgraphs and a left-over subgraph in [15]. The non left-over subgraphs are given independent set of colors and are colored simultaneously by applying CLP. The left-over subgraphs are colored once all other subgraphs are colored. After all  $G_i$  graphs are colored, we apply our modified CLP algorithm to complete the coloring of  $G^*$ .

In the first phase of the coloring algorithm, all subgraphs  $G_1, \dots, G_\ell$  are colored independently and simultaneously. This is done by allocating a distinct set of  $\Delta_i + 1$  colors for each of the  $G_i$  subgraphs. Overall, we allocate  $\ell \cdot (\Delta/\ell) \leq \Delta$  colors. Since each  $\Delta_i = O(\Delta^{3/4}) = O(n^{3/4})$ , this can be done in  $O(\log^* \Delta)$  rounds for all  $G_1, \dots, G_\ell$  simultaneously using the following:

► **Lemma 3.** [15] *There is a randomized  $(\Delta + 1)$ -coloring algorithm in the congested clique model that works in  $O(\log(1/\epsilon) \log^* \Delta)$  rounds when  $\Delta = O((n/\log n)^{1-\epsilon})$  for any  $\epsilon \in (0, 1)$ .*

The algorithm of [15] also implies that the same round complexity is obtained where one is given  $k$  vertex-disjoint subgraphs each with maximum degree  $O((n/\log n)^{1-\epsilon})$ .

**Coloring the remaining left-over subgraph  $G^*$ .** The second phase of the algorithm completes the coloring of  $G^*$ . This coloring should agree with the colors computed for  $G \setminus G^*$ . Hence,  $G^*$  is colored by employing a *list* coloring algorithm that we describe in the next section. We next bound the number of vertices and the maximum degree  $\Delta(G^*)$  of  $G^*$  which provides the basis for our ability to list-color it efficiently in the congested clique model using our modified CLP algorithm. By Chernoff bound the following holds:

► **Observation 4.**  $|V(G^*)| = O(n \log n / \Delta^{3/8})$  and  $\Delta(G^*) = O(\Delta^{5/8} \cdot \log n)$ .

For an illustration of our algorithm, see Figure 1.

### 3 List-Coloring of the Remaining Subgraph

Recall that the input graph  $G$  has  $n$  vertices and maximum degree  $\Delta$ . At the heart of our coloring algorithm is a list-coloring procedure that colors a subgraph  $G^* \subseteq G$  with bounded number of vertices  $N \leq n$  vertices and bounded maximum degree  $\Delta^*$ . For ease of presentation, we first assume that each vertex  $v \in G^*$  is given a palette of size  $\Delta^* + 1$ . At the end of the section, we explain the needed adaptation for the case where every vertex  $v \in G^*$  has a palette with at least  $\max\{\deg(v, G^*) + 1, \Delta^* - (\Delta^*)^{3/5}\}$  colors.

► **Theorem 5.** *Given a subgraph  $G^* \subseteq G$  with  $N$  vertices and maximum degree  $\Delta^*$ , such that each vertex  $v \in G^*$  has a palette of size  $\Delta^* + 1$ . If  $\Delta^*$  satisfies*

$$(I) \Delta^* \in [\sqrt{n}, O(n^{2/3})], \quad (II) \Delta^* = O(n/\sqrt{N}), \quad (III) \Delta^* = O(n^2/N^2).$$

*then  $G^*$  can be colored in  $O(\log^* \Delta)$  rounds, with high probability.*

At the of the section, Lemma 17, we show that the remaining subgraph from the previous section indeed satisfies properties (I-III).

**Key Definitions from the CLP Algorithm.** For an  $\epsilon \in (0, 1)$ , an edge  $e = (u, v)$  is an  $\epsilon$ -friend if  $|N(u) \cap N(v)| \geq (1 - \epsilon) \cdot \Delta^*$ . The endpoints of an  $\epsilon$ -friend edge are  $\epsilon$ -friends. A vertex  $v$  is  $\epsilon$ -dense if  $v$  has at least  $(1 - \epsilon) \cdot \Delta^*$   $\epsilon$ -friends; otherwise it is  $\epsilon$ -sparse.

Given a subset of vertices  $\tilde{V} \subseteq V(G^*)$  (which will be the set of uncolored vertices after the preliminary OneShotColoring algorithm), we define a partition of  $\tilde{V}$  into layers  $(V_1, \dots, V_\ell, V_{sp})$  for  $\ell = O(\log \log \Delta)$  based on the local sparsity. Let  $(\epsilon_1, \dots, \epsilon_\ell)$  be the sequence of sparsity parameters where  $\epsilon_1 = (\Delta^*)^{-1/10}$ ,  $\epsilon_i = \sqrt{\epsilon_{i-1}}$  for  $i \in [2, \ell - 1]$ , and  $\epsilon_\ell = 1/K$  for a large enough constant  $K$ . For a sparsity parameter  $\epsilon_i$ , let  $V_{\epsilon_i}^d, V_{\epsilon_i}^s$  be the set of vertices which are  $\epsilon_i$ -dense (resp.,  $\epsilon_i$ -sparse). This defines a hierarchy of  $\ell$  layers:  $V_1, \dots, V_\ell$  where  $V_1 = \tilde{V} \cap V_{\epsilon_1}^d$ ,  $V_i = \tilde{V} \cap (V_{\epsilon_i}^d \setminus V_{\epsilon_{i-1}}^d)$  and  $V_{sp} = \tilde{V} \cap V_{\epsilon_\ell}^s$ .

The  $\epsilon_i$ -dense vertices  $V_{\epsilon_i}^d$  are then partitioned into  $\epsilon_i$ -almost cliques for every  $\epsilon_i$ . The  $\epsilon_i$ -almost cliques are the connected components of the graph induced on  $V_{\epsilon_i}^d$  and the  $\epsilon_i$ -friend edges incident to these vertices. The following lemma developed in [8] contains some important properties of  $\epsilon$ -almost cliques.

► **Lemma 6.** *Fix any  $\epsilon < 1/5$ . The following conditions are met for each  $\epsilon$ -almost clique  $C$ , and each vertex  $v \in C$ . (i) The external degree  $|N(v) \cup (V_\epsilon^d \setminus C)| \leq \epsilon \Delta^*$  (ii) The anti-degree  $|C \setminus (N(v) \cup \{v\})| \leq 3\epsilon \Delta^*$ , (iii)  $|C| \leq (1 + 3\epsilon \Delta^*)$ , and (iv)  $\text{dist}_G(u, v) \leq 2$  for each  $u, v \in C$ , i.e.,  $C$  has weak diameter at most 2.*

For  $i \in [1, \ell]$ , each layer  $V_i$  is further partitioned into blocks as follows. Let  $\{C_1, C_2, \dots\}$  be  $\epsilon_i$ -almost cliques, then each clique  $C_j$  defines a block  $B_j = C_j \cap V_i$ , that is the block  $B_j$  contains the subset of vertices in  $C_j$  that are  $\epsilon_i$ -dense but are *not*  $\epsilon_{i-1}$ -dense. Note that all the blocks form a partition of  $\tilde{V}$ .

**An Outline of CLP Algorithm.** The sketch of a slightly *modified* version of CLP algorithm is outlined in the following. The pseudocodes for the subroutines of [5] are provided in Appendix A.

1. Execute OneShotColoring which takes  $O(1)$  rounds. Let  $\tilde{V}$  be the set of uncolored vertices. If  $v \in \tilde{V}$  is in layer  $i$  for  $i \in [2, \ell]$ ,  $v$  has  $\Omega(\epsilon_{i-1}^2 \Delta^*)$  excess colors. If  $v \in \tilde{V}$  is in  $V_{sp}$ , then it has  $\Omega(\epsilon_\ell^2 \Delta^*) = \Omega(\Delta^*)$  excess colors.
2. Execute the Dense Coloring Procedure, which takes  $O(\log^* \Delta)$  rounds. For every  $v \in \tilde{V}$ , the number of uncolored its uncolored neighbors in layer  $i$  will be bounded by  $O(\epsilon_i^5 \Delta^*)$  for  $i \in [2, \ell]$ . All vertices in layer 1 become colored.
3. After Step 1 and Step 2, for every vertex  $v \in \tilde{V}$ , if  $v$  is in layer 1, then it will be colored. If  $v$  is in layer  $i$  for  $i \in [2, \ell]$ , then it has  $\Omega(\epsilon_{i-1}^2 \Delta^*)$  excess colors in the palette. Moreover, the number of neighbors of  $v \in V_i$  with lower or equal layer is at most  $O(\sum_{j=1}^i \epsilon_j^5 \Delta^*) = O(\epsilon_i^5 \Delta^*) = O(\epsilon_{i-1}^{2.5} \Delta^*)$ . Since the number of competing neighbors is significantly less than the number of excess colors (i.e.  $\epsilon_{i-1}^{2.5} \Delta^* \ll \epsilon_{i-1}^2 \Delta^*$ ), we can color the remaining vertices very efficiently by using the ColorBidding algorithm (c.f. Appendix A) in  $O(\log^* \Delta)$  rounds.

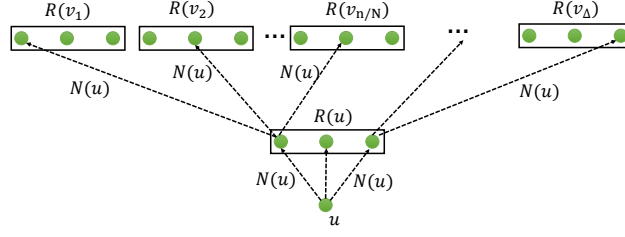
**Adaptation of CLP to the Congested Clique.** Our key contribution is in adopting the above CLP algorithm for coloring  $G^*$ , which might have maximum degree  $\Omega(n^{5/8})$ . For **Step 1**, Alg. `OneShotColoring` can be trivially implemented in  $O(1)$  rounds in the congested clique model, since each vertex is only sending one selected color to its neighbors.

The main challenge lies in **Step 2**. The basic idea of the dense coloring procedure is the following. Since the weak diameter of each block is 2, in the LOCAL model, it is possible for a leader in the block to collect the edges within the block and the palette of each vertex in the block. Then the leader assigns a random proper coloring to each vertex in the block. Since there are no internal conflicts in the block and the external number of neighbors is small for each vertex (i.e.,  $O(\epsilon_i \Delta^*)$  for a layer- $i$  block), the probability that a vertex is assigned the same color as any of its external neighbors is small ( $\text{poly}(\epsilon_i)$  if the vertex is in layer  $i$ ). Intuitively, after  $O(1)$  iterations, the probability that the vertex remains uncolored is  $O(\epsilon_i^5)$ . Therefore, it is plausible that for a given set of layer- $i$  vertices, the number of uncolored vertices is bounded  $O(\epsilon_i^5 \Delta^*)$ . However, this could have problems if a block is too small. In this case, the palette size of each vertex may also be small. The probability a layer- $i$  vertex remains uncolored may no longer be  $\text{poly}(\epsilon_i)$ . To deal with this issue, [5] groups the blocks into  $O(\log^* \Delta)$  strata. They showed that by coloring the strata in the right order, the palette of the vertices will be large enough at the time the procedure is executed. We describe it later in detail in the section **Coloring Vertices by Stratum**.

When it comes to the congested clique model, there are two obstacles. First, in the dense coloring procedure, each vertex has to know which layer and block it is in. For example, we need to compute the  $\epsilon$ -friends of each vertex *without* collecting its 2-hop neighborhoods. We show this can be achieved by using the idle vertices in  $G \setminus G^*$  as relaying vertices. The second obstacle is that in the congested clique, given a block  $B$ , we do not have the capacity to let each vertex in  $B$  send its incident edges and palette to the leader of  $B$ . Instead, we show that it will be sufficient if every vertex in  $B$  sends  $\sqrt{\Delta^*}$  independently sampled colors from its palette to the leader rather than the whole palette. Moreover, each vertex does not have to send the incident edges to the leader. This will be within our budget since  $|B| = O(\Delta^*)$  each leader receives  $O(\sqrt{\Delta^*} \cdot \Delta^*) = O(n)$  messages. The colors can be routed by using Lenzen's algorithm. We show such a modification have negligible effects when we color vertices by stratum.

For **Step 3**, in each iteration of `ColorBidding` algorithm, each vertex  $v$  sends a set of colors  $S_v$  to all its neighbors. The size of  $S_v$  is  $\tilde{O}((\Delta^*)^{1/5})$ . Since every vertex sends and receives  $\tilde{O}((\Delta^*)^{5/4}) = O(n)$  messages (by Property (I)), this can be implemented in  $O(1)$  rounds using Lenzen's routing algorithm.

**Computation of  $\epsilon$ -Friends.** We describe an  $O(1)$ -round procedure that allows each vertex  $v$  to compute its  $\epsilon_i$ -friends for each of the  $\ell = O(\log \log \Delta)$  sparsity values  $\epsilon_1, \dots, \epsilon_\ell$ . Using this information,  $v$  would be able to compute its minimum sparsity parameter  $\epsilon_i$  such that  $v$  is  $\epsilon_i$ -dense (but  $\epsilon_{i-1}$ -sparse). A trivial way to compute the  $\epsilon$ -friends of each vertex  $v$  is by collecting the neighbor-list of the  $v$ 's neighbors. Since this information contains  $\Omega((\Delta^*)^2)$  messages, it cannot be done in  $O(1)$  rounds when  $\Delta^* = \omega(\sqrt{n})$ . Instead, we use the fact that  $G^*$  has only  $N$  vertices and allocate to each vertex  $v \in G^*$ , a collection of  $n/N$  relay vertices  $R(v)$ . These relay vertices would help  $v$  in the computation of its  $\epsilon$ -friends. Towards that end, each vertex  $v$  first sends the IDs of its  $\Delta^*$  neighbors to each of its relay vertices  $R(v)$ . Hence, overall  $v$  sends  $\Delta^* \cdot |R(v)| = O(n)$  messages. At this point, the relay vertices  $R(v)$  of each vertex  $v$  know the neighbor list of their designated vertex  $v$ . Next,  $v$  partitions the "responsibilities" for its  $\Delta^*$  neighbors among its  $R(v)$  vertices. Formally, for  $r \in R(v)$ , let



■ **Figure 2** Illustration of  $\epsilon$ -Friends Computation via Relay Vertices. Verex  $u$  sends its neighbor-list  $N(u)$  to each  $r \in R(u)$ . Each  $r \in R(u)$  sends  $N(u)$  to the corresponding relay vertices of  $\Delta^*/|R(u)|$  neighbors in  $N(u)$ .

$N(v, r) \subseteq N(v)$  be the set of  $\Delta^*/|R(v)|$   $v$ 's neighbors assigned to  $r$ , where  $\cup_r N(v, r) = N(v)$ . Each  $r \in R(v)$  will receive the neighbor list of each vertex  $u \in N(v, r)$ . In total, each relay vertex  $r \in R(v)$  would be a target of  $\Delta^* \cdot |N(v, r)| = O((\Delta^*)^2 \cdot N/n) = O(n)$  messages, where the last bound holds due to property (II). To send the neighbor-list to the corresponding relay vertex, each vertex  $u$  uses its relay vertices  $R(u)$  again. Specifically, for each of its neighbors  $v_i \in N(u)$ ,  $u$  knows the ID of the relay vertex to which its neighbor list  $N(u)$  should be sent. It then partitions the responsibilities among its relay vertices by assigning  $\Delta^*/|R(u)|$  neighbors to each  $r' \in N(u)$ . The relay  $r' \in N(u)$  sends  $N(u)$  to the corresponding  $\Delta^*/|R(u)|$  relay vertices. Overall, each relay vertex sends  $O((\Delta^*)^2/|R(u)|) = O(n)$  messages (by Property (II)). For an illustration see Figure 2. Since each vertex is a source and a target of  $O(n)$  messages, this computation can be done in  $O(1)$  rounds by using Lenzen's routing algorithm. Each relay node  $r \in R(v)$  now holds the neighbor-list of  $\Delta^*/|R(v)|$  neighbors of  $v$  as well as the neighbor-list of  $v$ . This allows  $r$  to compute the intersection size between  $N(v)$  and  $N(u)$  for every  $u \in N(v, r)$ . More specifically, for each neighbor  $u$  of  $v$ , the relay node in  $R(v)$  responsible for  $u$  will send to  $v$  the minimum  $\epsilon'$  value such that  $u$  and  $v$  are  $\epsilon'$ -friends. Since  $v$  should receive  $O(\Delta^*)$  messages and each relay vertex  $r \in R(v)$  sends  $O(\Delta^*/|R(v)|)$  messages, this can be done in  $O(1)$  rounds using Lenzen's algorithm.

**Computation of Almost-Cliques and Block Partition.** Now each node  $v$  knows its  $\epsilon_i$ -friends for every sparsity level  $\epsilon_i$  for  $i \in [1, \ell]$ . It also knows which layer it is in. The next step is for  $v$  to know which block it is in.

To achieve this, we first compute  $\epsilon_i$ -almost cliques for each  $i \in [1, \ell]$ , where  $\ell = O(\log \log \Delta)$ . Recall that an  $\epsilon_i$ -almost clique is a component in the subgraph induced by  $\epsilon_i$ -friend edges and vertices in  $V_{\epsilon_i}^d$ . For  $i \in [1, \ell]$ , let  $C_{v, \epsilon_i}$  be the  $\epsilon_i$ -almost clique that contains  $v$ . Note that  $C_{v, \epsilon_i} = \emptyset$  if  $i$  is lower than the layer of  $v$ . For a specific level  $i$ , by using the  $O(1)$ -round connectivity-identification algorithm of [11], each vertex  $v$  is able to learn the ID of every vertex in  $C_{v, \epsilon_i}$ .

However, we cannot afford to apply the connectivity-identification algorithm in a serial manner for each sparsity class. Instead, we will use again  $n/N$  relay vertices assigned to each vertex in  $G^*$ . This is done as follows. For each  $v \in G^*$ , we allocate a relay vertex  $u_i$  that "plays" the role of  $v$  in the  $i^{\text{th}}$  application of the connectivity-identification algorithm, i.e., for computing  $\epsilon_i$ -almost cliques for every  $i \in \{1, \dots, O(\log \log \Delta)\}$ . Since  $n/N = \Omega(\log \log \Delta)$ , this is within our budget. By letting  $v$  sending its  $\Delta^*$  vertices to each relay vertex  $u_i$ , the latter have all the information needed to run the connectivity algorithm on behalf of  $v$ . This allows us to apply the  $O(\log \log \Delta)$  connectivity-identification algorithms *simultaneously*. At the end of this computation, each relay node of  $v$  sends to  $v$  the ID of every vertex in its connected component. This is possible since the size of each almost-clique is  $O(\Delta^*)$  (by

Lemma 6) and  $O(\Delta^* \cdot \log \log \Delta) = O(n)$  (by Property (I)). This allows each vertex  $v \in G^*$  to learn all the vertices in each almost clique  $C_{v,\epsilon_1}, \dots, C_{v,\epsilon_\ell}$ . Note that we are computing more information than we need here, but it will be used later.

We are now ready to compute the partitioning of the cliques into blocks. Suppose that a vertex  $v$  is in layer  $i(v)$ . The block containing  $v$ ,  $B_v$ , was defined as  $(C_{v,\epsilon_{i(v)}} \cap V_{i(v)})$ . Vertex  $v$  can identify all the members in  $B_v$  if it knows the layer of every vertex in  $G^*$ . This information can be obtained by every vertex in one round in the congested clique, since every vertex already knows its layer.

**Coloring Vertices by Stratum.** To ensure that the palettes of the vertices in each block contain enough color in the execution of the dense coloring procedure, the CLP algorithm groups the  $\ell$  layers into  $s = O(\log^* \Delta)$  Strata  $W_1, \dots, W_s$  where  $W_1 = V_1$  and

$$W_k = \bigcup_{i:\epsilon_i \in (\xi_{k-1}, \xi_k]} V_i \text{ where } \xi_1 = \epsilon_1 \text{ and } \xi_k = 1/\log(1/\xi_{k-1}) \text{ for } k \in [2, s].$$

Each vertex can easily determine the stratum of a vertex by its layer. The blocks are also divided into two categories, *large* blocks and *small* blocks. We say a block  $B$  is *good* if  $|B| \geq \Delta^*/\log^2(1/\xi_k)$ , where  $k$  is the stratum that  $B$  lies at. A vertex  $v$  determines whether its block  $B_v$  is large or small by the following criteria. Let  $i(v)$  denote the layer of  $v$ . If  $B_v$  is good and none of the following blocks  $(C_{v,\epsilon_{i(v)+1}} \cap V_{i(v)+1}), (C_{v,\epsilon_{i(v)+2}} \cap V_{i(v)+2}), \dots$ , or  $(C_{v,\epsilon_\ell} \cap V_\ell)$  are good, then  $B_v$  is a large block. Otherwise,  $B_v$  is a small block. Since each vertex  $v$  knows the vertices in each almost clique  $C_{v,\epsilon_1}, \dots, C_{v,\epsilon_\ell}$  and  $v$  knows the layers and the stratum of all other vertices, whether  $B_v$  is small or large can be determined locally.

Define  $W_k^S$  and  $W_k^L$  be the set of all vertices in stratum- $k$  small blocks and stratum- $k$  large blocks. We have that  $\tilde{V} = (W_1^S, \dots, W_s^S, W_1^L, \dots, W_s^L, V_{sp})$ . The vertices are colored in  $s+2$  stages. First, all the small blocks are colored in  $s$  phases: stratum by stratum. In other words, all the vertices in the small block are colored according to the order:  $W_s^S, \dots, W_1^S$ . Next, the algorithm colors the vertices of  $W' = \bigcup_{j=2}^s W_j^L$ , i.e., all the vertices in large blocks, except for those belonging to blocks of the first layer  $W_1^L$ . Lastly, the vertices of the large blocks in  $W_1^L$  are colored. At the end, the (small) subset of vertices that failed to be colored and the sparse vertices  $V_{sp}$  are colored in Step 3.

Suppose that we process the blocks according to the stratum in the order described above. In [5], they showed a crucial property that when we are coloring a small block in  $W_k^S$ , each vertex has  $\Delta^*/2 \log^2(1/\xi_k)$  *excess* colors (i.e. each vertex  $v$  has at least  $|N(v) \cap W_k^S| + \Delta^*/2 \log^2(1/\xi_k)$  colors) in its palette. When we process a large block, since the block is large, each vertex has at least  $\Delta^*/2 \log^2(1/\xi_k)$  colors in the palette.

The CLP algorithm consists of two different versions of dense coloring procedures, according to whether the blocks being processed are small or large. In a high level way, the differences are the following: In large blocks we do not have the excess colors. However, all blocks belongs to different almost-cliques by the definition of large. In small blocks, when we are processing blocks in  $W_k^S$ , it is possible that some blocks belong to the same almost-clique. However, since there are abundant number of excess colors, this allow us to process all these blocks together using a single leader. The *superblocks* are defined for this purpose. Consider  $W_k^S$  and suppose that stratum- $k$  spans layer  $i_0, \dots, i_1$ . Let  $\{C_1, C_2, \dots\}$  be  $\epsilon_{i_1}$  almost cliques. Each  $C_j$  defines a superblock  $R_j = C_j \cap W_k^S$ . Therefore,  $(R_1, R_2, \dots)$  is a partition of  $W_k^S$ . Each vertex  $v$  can easily identify the members in its superblock by using the same approach we described for computing the blocks.



**Version 1 – Dense Coloring Procedure for Strata of Small Blocks.** Consider the set of stratum- $k$  small blocks  $W_k^S$ . Recall that if stratum- $k$  spans layers  $i_0, i_0 + 1, \dots, i_1$ , then a super-block is a maximal almost- $\epsilon_{i_1}$  clique induced in  $W_k^S$ . Let  $S = W_k^S$  and  $S_1, \dots, S_g$  be the super-blocks.

Given vertices  $u, v \in S$ , we say that  $u$  has a *higher priority* than  $v$  if (1) the layer of  $u$  is lower than that of  $v$ , or (2)  $u$  and  $v$  are in the same layer but  $ID(u) < ID(v)$ . The CLP algorithm for coloring each super-block  $S_j$  works as follows. Let  $\pi : \{1, \dots, |S_j|\} \rightarrow S_j$  be a permutation ordered by the priority of the vertices, from the highest priority to the lowest. Now a leader processes each vertex  $\pi(1), \pi(2), \dots, \pi(|S_j|)$ . For vertex  $\pi(q)$ , it selects a color randomly from its palette excluding the colors used by its neighbors in  $\pi(1), \dots, \pi(q-1)$ . Also, CLP showed that in small blocks, each vertex has  $Z_{ex} = \Delta^*/2 \log^2(1/\xi_k)$  excess colors. Let  $N'(v)$  denote the higher priority neighbors of  $v$ . Suppose that  $v$  is in layer  $i$ , we must have  $|N'(v) \cap (S \setminus S_j)| \leq \epsilon_i \Delta^*$ . Therefore, when the vertex is being processed, the probability that it has an external conflict is at most  $(\epsilon_i \Delta^*) / (Z_{ex}) \leq 2\epsilon_i \log^2(1/\xi_k) \leq 2\epsilon_i \log^2(1/\epsilon_i)$ .

Since we cannot afford each vertex to send the whole palette and its incident edges to the leader in the congested clique model, we let each vertex randomly sample  $\sqrt{\Delta^*}$  colors and send them to the leader. Since  $|S_j| = O(\Delta^*)$ , each leader is receiving at most  $\sqrt{\Delta^*} \cdot O(\Delta^*) = O(n)$  messages (by Property (I)). Thus, the set of colors can be routed to the leader by using Lenzen's routing algorithm. The following is the description of our algorithm.

**ModifiedSmallDenseColoring** (Modified Alg. of DenseColoringStep, version 1 from [5]).

1. Consider a superblock  $S_j$ . Let  $\pi$  be the permutation of  $S_j$  ordered by the priority of the vertices. Each vertex  $\pi(q)$  sends a set  $C(\pi(q))$  of  $\sqrt{\Delta^*}$  colors selected u.a.r. from its palette to the leader. For each selected vertex  $\pi(q)$ , if  $C(\pi(q)) \setminus \{c(\pi(1)), \dots, c(\pi(q-1))\}$  is non-empty, then the leader assigns  $\pi(q)$  a color  $c(\pi(q))$  randomly selected from the set. Otherwise, we say that  $\pi(q)$  is *skipped*.
2. Each  $v \in S_j$  that has selected a color  $c(v)$  permanently color itself  $c(v)$ , if  $c(v)$  is not selected by any vertices  $u \in N'(v)$ . Otherwise, we say that  $v$  is *decolored*.

► **Lemma 7.** Let  $D_v$  denotes an upper bound on the external higher priority neighbors (i.e.  $|N'(v) \setminus (S \setminus S_j)|$ ). Let  $\delta_v = 2D_v/Z_{ex}$ . The probability that  $v = \pi(q)$  becomes decolored is at most  $\delta_v$ , conditioned on any choices of  $\pi(1), \pi(2), \dots, \pi(q-1)$  and all the other higher priority vertices in  $S \setminus S_j$ .

**Proof.** Consider a vertex  $\pi(q)$ . Since the anti-degree of  $\pi(q)$  is at most  $3\epsilon_{i_1} \Delta$ , at most  $3\epsilon_{i_1} \Delta$  vertices in  $S_j$  can be non-neighbors of  $\pi(q)$ . Moreover,  $\pi(q)$  has at least  $Z_{ex}$  uncolored neighbors outside of  $S$ . Therefore,  $|\text{Pal}(\pi(q)) \setminus \{c(\pi(1)), \dots, c(\pi(q-1))\}| \geq Z_{ex} - 3\epsilon_{i_1} \Delta^* \geq Z_{ex}/2$ . Consider any assignment of colors  $c(\pi(1)), \dots, c(\pi(q-1))$ . Since  $v$  selects the colors randomly, any color that is not  $c(\pi(1)), \dots, c(\pi(q-1))$  has the same probability to be assigned as  $c(\pi(q))$ . Therefore, the probability that  $c(\pi(q))$  is a specific color is at most  $1/|\text{Pal}(\pi(q)) \setminus \{c(\pi(1)), \dots, c(\pi(q-1))\}| \leq 2/Z_{ex}$ . Now consider any choices made by  $\pi(1), \pi(2), \dots, \pi(q-1)$  and any choices made by external neighbors in  $N'(\pi(q)) \cap (S \setminus S_j)$ . Vertices  $N'(\pi(q)) \cap (S \setminus S_j)$  are assigned with at most  $D_v$  different colors. Vertex  $v$  can only become decolored only if  $c(\pi(q))$  is one of the colors selected by the vertices in  $N'(\pi(q)) \cap (S \setminus S_j)$ . Therefore, the probability that  $\pi(q)$  is decolored is at most  $2D_v/Z_{ex}$ . ◀

► **Lemma 8.** In ModifiedSmallDenseColoring, with probability at least  $\exp(-\Omega(\text{poly}(\Delta^*)))$ , no vertices are skipped.

**Proof.** Recall that  $S_j$  is a superblock in stratum  $k$ , which spans layer  $i_0, \dots, i_1$ . Consider a vertex  $\pi(q)$ . Since the anti-degree of  $\pi(q)$  is at most  $3\epsilon_{i_1} \Delta$ , at most  $3\epsilon_{i_1} \Delta$  vertices in  $S_j$  can

be non-neighbors of  $\pi(q)$ . Therefore, the palette size of  $\pi(q)$  is at least  $|S_j| - 3\epsilon_{i_1}\Delta^* + Z_{ex}$ . The probability that  $\pi(q)$  is skipped is at most  $\left(\frac{i}{|S_j| + Z_{ex} - 3\epsilon_{i_1}\Delta^*}\right)^{\sqrt{\Delta^*}}$ . Note that

$$\begin{aligned} Z_{ex} - 3\epsilon_{i_1}\Delta^* &= \left(\frac{1}{(2\log^2(1/\xi_k))} - 3\epsilon_{i_1}\right) \cdot \Delta^* \geq \left(\frac{1}{(2\log^2(1/\epsilon_{i_1}))} - 3\epsilon_{i_1}\right) \cdot \Delta^* \\ &\geq \frac{\Delta^*}{4\log^2(1/\epsilon_{i_1})} && \frac{1}{\epsilon_i} \geq K \text{ for large enough constant } K \\ &\geq C \cdot \frac{\Delta^*}{\log^2(\Delta^*)} && \text{for some constant } C > 0 \end{aligned}$$

Let  $X$  be the random variable denoting the total number of vertices skipped. We have

$$\begin{aligned} \mathbf{E}[X] &= \sum_{i=1}^{|S_j|} \left(\frac{i}{|S_j| + Z_{ex} - 3\epsilon\Delta^*}\right)^{\sqrt{\Delta^*}} \leq |S_j| \cdot \left(\frac{|S_j|}{|S_j| + C \cdot \frac{\Delta^*}{\log^2(\Delta^*)}}\right)^{\sqrt{\Delta^*}} \\ &\leq 2\Delta^* \cdot \left(\frac{2\Delta^*}{2\Delta^* + C \cdot \frac{\Delta^*}{\log^2(\Delta^*)}}\right)^{\sqrt{\Delta^*}} \leq 2\Delta^* \cdot \left(\frac{1}{1 + \frac{C}{2} \cdot \frac{1}{\log^2(\Delta^*)}}\right)^{\sqrt{\Delta^*}} \quad |S_j| \leq 2\Delta^* \\ &\leq 2\Delta^* \cdot \left(\frac{1}{\exp\left(\frac{C}{4\log^2(\Delta^*)}\right)}\right)^{\sqrt{\Delta^*}} && 1 + x \geq \exp(x/2) \text{ for } 0 < x \leq 2 \\ &\leq 2\Delta^* \cdot \exp\left(-\frac{C}{4} \cdot \frac{\sqrt{\Delta^*}}{\log^2(\Delta^*)}\right) = \exp(-\Omega(\text{poly}(\Delta^*))) \quad \blacktriangleleft \end{aligned}$$

Therefore, by Lemma 7 and Lemma 8, a similar version of Lemma 17 in [5] holds.

► **Lemma 9.** *Consider an execution of ModifiedSmallDenseColoring. Let  $T$  be any subset of  $S$  and let  $\delta = \max_{v \in T} \delta_v$ . For any  $t$ , the number of uncolored vertices in  $T$  is at least  $t$  with probability at most  $\Pr(\text{Binomial}(|T|, \delta) \geq t) + \exp(-\Omega(\text{poly}(\Delta^*)))$ .*

**Proof.** The proof is essentially the same with that of Lemma 17 in [5]. Let  $T = \{v_1, \dots, v_{|T|}\}$  be the vertices listed according to their priorities. Conditioned on any choices of  $v_1 \dots v_{q-1}$ , the probability that  $v_q$  is decolored is at most  $\delta_v \leq \delta$  by Lemma 7. Therefore, the probability that at least  $t$  vertices are decolored is at most  $\Pr(\text{Binomial}(|T|, \delta) \geq t)$ . The probability that there is any skipped vertex is at most  $\exp(-\Omega(\text{poly}(\Delta^*)))$  by Lemma 8. If there are at least  $t$  uncolored vertices, then either there are at least  $t$  vertices that are decolored or some vertices are skipped. By taking an union over the two events, we conclude the probability there are at least  $t$  uncolored vertices is at most  $\Pr(\text{Binomial}(|T|, \delta) \geq t) + \exp(-\Omega(\text{poly}(\Delta^*)))$ . ◀

**Completing the proof for small blocks, other than stratum 1.** We will show that Lemma 6 in [5] holds by using our simulation. That is, we show that after  $O(1)$  iterations of ModifiedSmallDenseColoring, w.h.p. for every  $v \in \tilde{V}$  and for each layer  $i \in [2, l]$ , the number of uncolored layer- $i$  neighbors of  $v$  that are in  $W_k^S$  is at most  $\epsilon_i^5 \Delta^*$ .

Consider a vertex  $v$  in  $\tilde{V}$ . Let  $T$  be the set of layer- $i$  neighbors of  $v$  in  $S$ . Let  $\delta = \max_{u \in T} \delta_u \leq 2\epsilon_i \Delta^* / Z_{ex} \leq 4\epsilon_i \log^*(1/\epsilon_i)$ . We execute 6 iterations of procedure ModifiedSmallDenseColoring. Let  $t_0 = |T|$  and  $t_l = \max(2\delta t_{l-1}, \epsilon^5 \Delta^*)$ . Note that  $t_6 \leq \epsilon^5 \Delta^*$ . Suppose that the number of uncolored vertices in  $T$  is at most  $t_{l-1}$  at the beginning of iteration  $l$ . By Lemma 9, with probability at least  $1 - \Pr(\text{Binomial}(t_l, \delta) \geq t_{l-1}) - \exp(-\Omega(\text{poly}(\Delta^*))) = 1 - \exp(-\Omega(\text{poly}(\Delta^*))) = 1 - 1/\text{poly}(n)$ , at the end of iteration  $l$ , the number of uncolored vertices in  $T$  is at most  $t_l$ . By an union bound over such events over the 6 iterations, with probability at least  $1 - 1/\text{poly}(n)$ , the number of uncolored vertices in  $T$  is at most  $\epsilon_i^5 \Delta^*$ .

**Completing the proof small blocks of stratum 1.** Note that stratum 1 only consists of vertices that are in layer 1. Instead of proving an analogous lemma to Lemma 7 in [5], we prove the following lemma that bounds the maximum degree on each layer-1 vertex.

► **Lemma 10.** *Suppose that each vertex in  $W_1^S$  has at least  $\Delta^*/2 \log^2(1/\epsilon_1)$  excess colors w.r.t.  $W_1^S$ . Let  $v \in \tilde{V}$ . By executing `ModifiedSmallDenseColoring` for  $O(1)$  rounds, w.h.p. the number of uncolored vertices in  $|N(v) \cap W_1^S|$  is at most  $(\Delta^*)^{1/20}$  for all  $v \in W_1^S$ .*

**Proof.** Let  $T = N(v) \cap W_1^S$  be the neighbors of  $v$  in  $W_1^S$ . Let  $\delta = \max_{u \in T} \delta_u \leq 2\epsilon_1 \Delta^*/Z_{ex} = 4\epsilon_1 \log^*(1/\epsilon_1) < (\Delta^*)^{-1/20}/2$ , where the last inequality holds since  $\epsilon_1 = (\Delta^*)^{-1/10}$ . We will execute 19 iterations of `ModifiedSmallDenseColoring`.

Let  $t_0 = |T|$  and  $t_l = \max((\Delta^*)^{-1/20} t_{l-1}, (\Delta^*)^{1/20})$  for  $1 \leq l \leq 19$ . Suppose that the number of uncolored vertices in  $T$  is at most  $t_{l-1}$  at the beginning of iteration  $l$ . By Lemma 9, with probability at least  $1 - \Pr(\text{Binomial}(t_l, \delta) \geq t_{l-1}) - \exp(-\Omega(\text{poly}(\Delta^*))) = 1 - \exp(-\Omega(\text{poly}(\Delta^*))) = 1 - 1/\text{poly}(n)$ , at the end of iteration  $l$ , the number of uncolored vertices in  $T$  is at most  $t_l$ . Therefore, by an union bound on the events over the 19 iterations, w.h.p. the number of uncolored vertices in  $T$  is at most  $t_l \leq |T| \cdot (\Delta^*)^{19/20} \leq (\Delta^*)^{1/20}$ . ◀

By Lemma 10, since the maximum degree of the induced subgraph of layer 1 vertices is at most  $(\Delta^*)^{1/20}$  and  $N \cdot (\Delta^*)^{1/20} = O(n)$  (by Property (III)), a leader can collect the entire topology and the palette of each vertex and compute a coloring of the  $W_1^S$  locally.

**Version 2 – Dense Coloring Procedure for Strata of Large Blocks.** Let  $S = W_2^L \cup W_3^L \cup \dots \cup W_s^L$  or  $S = W_1^L$  be a set vertices in these large blocks. A crucial difference between large blocks and small blocks is that for any block  $B \subseteq S$ , if  $v \in B$ , the number of external neighbors in other blocks with lower or equal layers in  $S$  is at most  $O(\epsilon \Delta^*)$ . This property allows us to deal with all the large blocks simultaneous. Suppose that  $S$  is partitioned into  $S_1, \dots, S_g$  (vertex-disjoint) blocks, where each block  $S_j$  is associated with an ID,  $ID(S_j) = \min_{v \in S_j} ID(v)$ . We associate each  $S_j$  with parameters  $D_j$  and  $\delta_j$ . Roughly speaking,  $D_j$  represents an upper bound on both the external degree and the anti-degree of each vertex in  $S_j$  and  $\delta_j$  is an upper bound on the probability that a vertex in  $S_j$  fails to be colored in a single iteration of `ModifiedLargeDenseColoring`. We say a  $S_j$  has a higher priority than  $S_{j'}$  if either (1)  $\delta_j < \delta_{j'}$  or (ii)  $\delta_j = \delta_{j'}$  and  $ID(S_j) < ID(S_{j'})$ . For a vertex  $v \in S_j$ , let  $N''(v)$  be the neighbors of  $v$  higher priority blocks in  $S$ . We simplify the analysis (compared to that of [5]) by choosing a slightly larger  $\delta_j$ . We let  $\delta_j = 2 \cdot \sqrt{D_j/Z_j}$ . We modify the algorithm as follows:

**ModifiedLargeDenseColoring** (Modified Alg. of `DenseColoringStep`, version 2 from [5]).

1. Each cluster  $S_j$  selects  $(1-\delta)|S_j|$  vertices  $S'_j$  u.a.r. The vertices in  $S'_j$  are the *selected* vertices. Let  $\pi$  be a random permutation of selected vertices, chosen u.a.r. Each selected vertex  $\pi(q)$  send a set  $C(\pi(q))$  of  $\sqrt{\Delta^*}$  colors selected u.a.r. from its palette to the leader. For each selected vertex  $\pi(q)$ , if  $C(\pi(q)) \setminus \{c(\pi(1)), \dots, c(\pi(q-1))\}$  is non-empty, then the leader assigns  $\pi(q)$  a color  $c(\pi(q))$  randomly selected from the set. Otherwise, we say  $\pi(q)$  is *skipped*.
2. Each  $v \in S_j$  that has selected a color  $c(v)$  permanently color itself  $c(v)$ , if  $c(v)$  is not selected by any vertices  $u \in N''(v)$ . Otherwise, we say that  $v$  is *decolored*.

The algorithm above will be executed for  $O(1)$  iterations.

► **Lemma 11.** *[Analogue of Lemma 19 in [5]] Let  $T = \{v_1, \dots, v_k\}$  be any subset of uncolored vertices of  $S_j$ . The probability that  $v$  is decolored for all  $v \in T$  is  $O(\delta_j)^{|T|}$ , conditioned on*

any choices made by vertices in higher priority blocks than  $S_j$  and on whether vertices are selected in  $S_j \setminus T$ .

**Proof.** We assume that  $v_1, \dots, v_k$  are among the  $(1 - \delta_j)|S_j|$  vertices that are selected. Otherwise, the probability that all vertices in  $T$  are decolored would be 0. Let  $c_1, \dots, c_k$  be a sequence of colors. Let  $\mathcal{E}_q$  denote the event that  $v_m$  select  $c_m$  for  $m \in [1, q]$ . We have:

$$\Pr(\mathcal{E}_q \mid \mathcal{E}_{q-1}) \leq \frac{1}{\delta_j |S_j| - D_j} \leq \frac{1}{\delta_j \cdot Z_j - D_j} \leq \frac{1}{2 \cdot \sqrt{D_j Z_j} - D_j} \leq \frac{1}{\sqrt{D_j Z_j}}$$

Therefore,  $\Pr(\mathcal{E}_k) \leq \left( \frac{1}{\sqrt{D_j Z_j}} \right)^k$ . For every vertex  $v_q$ , there are at most  $D_j$  different colors chosen by  $N''(v_q)$  that can cause  $v_q$  to become decolored. By considering the  $(D_j)^k$  combination of forbidden colors for  $v_1, \dots, v_k$ , we conclude the probability that all vertices in  $T$  are decolored is at most  $(D_j / \sqrt{D_j Z_j})^k = O(\delta_j)^k$ . ◀

► **Lemma 12.** *Let  $T = \{v_1, \dots, v_k\}$  be any subset vertices of  $S_j$ . The probability that  $v_q$  is skipped for all  $q \in [1, k]$  is  $O(1/\sqrt{\Delta^*})^{|T|}$ . The statement is true even if we are conditioning on any choices made by vertices in higher priority blocks than  $S_j$  and on whether vertices are selected and decolored in  $S_j \setminus T$ .*

**Proof.** First we assume all nodes in  $T$  are selected in  $S'_j$ . Otherwise, the probability that  $v_q$  is skipped for all  $q \in [1, k]$  is 0. Let  $S''_j$  denote the set of vertices in  $S'_j$  that are skipped.

Let  $r_q$  be the rank of  $v_q$  among  $(S'_j \setminus T) \cup \{v_1, \dots, v_k\}$  in the permutation. Let  $\mathcal{E}_q$  denote the event that for  $m \in [1, q]$ , all the colors picked by  $v_m$  are assigned to some vertices in  $S'_j \setminus T$  with smaller ranks than  $v_m$ . Note that  $T \subseteq S''_j$  if and only if  $\mathcal{E}_k$  holds. If  $r_q$  is fixed, we have  $\Pr(\mathcal{E}_q \mid \mathcal{E}_{q-1}) \leq \left( \frac{r_q - 1}{|S'_j| - D_j} \right)^{\sqrt{\Delta^*}}$ . Therefore,

$$\begin{aligned} \Pr(T \subseteq S''_j) &= \mathbf{E}[\Pr(T \subseteq S''_j)]_{r_1 \dots r_k} \leq \mathbf{E} \left[ \prod_{q=1}^k \left( \frac{r_q - 1}{|S'_j| - D_j} \right)^{\sqrt{\Delta^*}} \right]_{r_1 \dots r_k} \\ &= \prod_{q=1}^k \mathbf{E} \left[ \left( \frac{r_q - 1}{|S'_j| - D_j} \right)^{\sqrt{\Delta^*}} \mid r_1 \dots r_{q-1} \right]_{r_q} \\ &\leq \prod_{q=1}^k \left( \frac{1}{|S'_j| - k + q} \sum_{x=1}^{|S'_j| - k + q} \left( \frac{x - 1}{|S'_j| - D_j} \right)^{\sqrt{\Delta^*}} \right) \leq \left( \frac{1}{|S'_j|} \cdot \sum_{x=1}^{|S'_j|} \left( \frac{x - 1}{|S'_j| - D_j} \right)^{\sqrt{\Delta^*}} \right)^k \\ &\leq \left( \frac{1}{|S'_j|} \cdot O \left( \frac{|S'_j|^{\sqrt{\Delta^*} + 1}}{\sqrt{\Delta^*} \cdot (|S'_j| - D_j)^{\sqrt{\Delta^*}}} \right) \right)^k \leq O \left( \frac{1}{\sqrt{\Delta^*}} \right)^k \quad |S'_j| \leq |S_j| - D_j \quad \blacktriangleleft \end{aligned}$$

► **Lemma 13.** *Let  $T$  be any subset vertices of  $S$  that have not been assigned a color and let  $\delta = \max_{j: S_j \cap T \neq \emptyset} \delta_j$ . After an iteration of the algorithm, the probability that the number of uncolored vertices in  $T$  is at least  $t$  is at most  $\binom{|T|}{t} \cdot (O(\delta + (\Delta^*)^{-1/2}))^t$ .*

**Proof.** Suppose that the clusters  $S_1, \dots, S_g$  are ordered by their priorities, from the highest to the lowest. Let  $U$  be a size- $t$  subset of  $T$ . we consider the  $3^t$  ways of partitioning  $U$  into  $U_1, U_2$ , and  $U_3$ . Note that a vertex remains uncolored only if it is either unselected, decolored, or skipped. We will calculate the probability that the vertices in  $U_1$  are not selected, vertices in  $U_2$  are decolored, and vertices in  $U_3$  are skipped.

Define  $U_i^{(j)} = U_i \cap S_j$  for  $i = 1, 2, 3$  and  $j = 1, 2, \dots, g$ . The probability that every vertex in  $U_1^{(j)}$  is unselected is at most  $O(\delta_j)^{|U_1^{(j)}|} \leq O(\delta)^{|U_1^{(j)}|}$ . By Lemma 11, the probability that every vertex in  $U_2^{(j)}$  is decolored is at most  $O(\delta_j)^{|U_2^{(j)}|} \leq O(\delta)^{|U_2^{(j)}|}$ . By Lemma 12, the probability that every vertex in  $U_3^{(j)}$  is skipped is at most  $O(\frac{1}{\sqrt{\Delta^*}})^{|U_3^{(j)}|}$ . Therefore, the probability that all vertices in  $U_1$  are unselected, all vertices in  $U_2$  are decolored, all vertices in  $U_3$  are skipped are at most  $\prod_{j=1}^g O(\delta)^{|U_1^{(j)}|} \cdot O(\delta)^{|U_2^{(j)}|} \cdot O((\Delta^*)^{-1/2})^{|U_3^{(j)}|} = O(\delta + (\Delta^*)^{-1/2})^t$ . By an union over all possible size- $t$  sets and partitions, the probability there are at least  $t$  uncolored vertices is at most  $3^t \cdot \binom{|T|}{t} \cdot O(\delta + (\Delta^*)^{-1/2})^t \leq \binom{|T|}{t} \cdot O(\delta + (\Delta^*)^{-1/2})^t$ .  $\blacktriangleleft$

**Maintenance of Invariants.** Suppose that  $S_j$  is a layer- $i$  block. We show that w.h.p. the following invariants are maintained after each iteration  $l$ .

- Invariant  $\mathcal{H}_l(v)$ : Both the anti-degree the external degree of  $v$  are at most  $D_j^{(l+1)}$ .
- Invariant  $\mathcal{H}_l(S_j)$ : the number of uncolored vertices of  $S_j$  is at least  $Z_j^{(l+1)}$ .
- Sequence  $(D_j^{(l)})$ :  $D_j^{(1)} = 3\epsilon_i \Delta^*$ , and  $D_j^{(l)} = \beta \cdot \delta_j^{(l-1)} \cdot D_j^{(l-1)}$ , for  $l > 1$ , where  $\beta > 1$  is an absolute constant.
- Sequence  $(Z_j^{(l)})$ :  $Z_j^{(1)} = \frac{\Delta^*}{\log^2(1/\epsilon_i)}$ , and  $Z_j^{(l)} = \delta_j^{(l-1)} \cdot Z_j^{(l-1)}$ , for  $l > 1$ .

► **Observation 14.**  $\delta_j^{(l)} = \Omega(1/\sqrt{\Delta^*})$  for every  $l \geq 1$  and so the probability in Lemma 13 is  $\binom{|T|}{t} \cdot (O(\delta))^t$ .

**Proof.** For a layer- $i$  block,  $D_j^{(1)} = 3\epsilon_i \cdot \Delta^*$ ,  $Z_j^{(1)} = \Delta^* / \log^2(1/\epsilon_i)$ , thus  $\delta_j^{(1)} = 2\sqrt{D_j^{(1)}/Z_j^{(1)}} = \Omega(\sqrt{\epsilon_i \cdot \log^2(1/\epsilon_i)}) = \Omega((\Delta^*)^{-1/20}) = \Omega((\Delta^*)^{-1/2})$ , since  $\epsilon_i \geq 1/(\Delta^*)^{1/10}$ . Next, note that  $\delta_j^{(l)}$  is increasing with  $l$ , since  $\delta_j^{(l)} = 2\sqrt{\beta} \cdot \delta_j^{(l-1)} > \delta_j^{(l-1)}$ .  $\blacktriangleleft$

For every block  $S_j$ , in the beginning it is clearly true that  $\mathcal{H}_0(v)$  holds for  $v \in S_j$  and  $\mathcal{H}_0(S_j)$  holds. Suppose that for every block  $S_j$ ,  $\mathcal{H}_{l-1}(v)$  holds for  $v \in S_j$  and  $\mathcal{H}_{l-1}(S_j)$  holds. Consider a block  $S_j$ . Since the number of unselected vertices is always at least  $\delta_j^{(l)} \cdot |S_j| \geq \delta_j^{(l)} \cdot Z_j^{(l)} = Z_j^{(l+1)}$ ,  $\mathcal{H}_l(S_j)$  hold with probability 1.

Consider a vertex  $v \in S_j$ . Let  $T$  be the set of uncolored external neighbors of  $v$  or the set of uncolored non-neighbor of  $v$  in  $S_j$ . Since  $\mathcal{H}_{l-1}(v)$  holds,  $|T| \leq D_j^{(l)}$ . Let  $t = \beta \delta_j^{(l)} \cdot D_j^{(l)}$ . By Lemma 13 and Obs. 14, we have  $\Pr(\mathcal{H}_l(v)) \geq 1 - \binom{|T|}{t} \cdot (K\delta_j^{(l)})^t \geq 1 - (\frac{Kt}{\beta T} \cdot \frac{e|T|}{t})^t \geq 1 - 1/\text{poly}(n)$ . By taking an union bound on the event  $\mathcal{H}^l(v)$  holds for all  $v \in S$ , w.h.p. the invariants hold after iteration  $l$ .

**Completing the proof for large blocks, other than stratum 1.** We show that after  $O(1)$  iterations of ModifiedLargeDenseColoring, w.h.p. for every  $v \in \tilde{V}$  and for each layer  $i \in [2, l]$ , the number of uncolored layer- $i$  neighbors of  $v$  that are in  $W_2^i \cup W_3^i \cup \dots \cup W_s^i$  is at most  $\epsilon_i^5 \Delta^*$ . (This is the analogue of Lemma 8 [5].) We execute Algorithm ModifiedLargeDenseColoring for 12 iterations. W.h.p. for  $l \in [0, 12]$ , the invariants  $\mathcal{H}_l(S_j)$  hold for every  $S_j$  and  $\mathcal{H}_l(v)$  holds for every  $u \in S$ . Let  $T$  be the set of layer- $i$  neighbors of  $v$  in  $S$  and

$$\delta^{(l)} = \max_{j: S_j \cap T \neq \emptyset} \delta_j^{(l)} = 2 \cdot \sqrt{D_j^{(l)}/Z_j^{(l)}} = 2\sqrt{\beta}^{l-1} \cdot \sqrt{3\epsilon_i \cdot \log^2(1/\epsilon_i)}.$$

Let  $t_0 = |T|$  and  $t_l = \max(\beta \delta^{(l)} t_{l-1}, \epsilon^5 \Delta^*)$ . Suppose that at the beginning of iteration  $l$ , the number of uncolored vertices in  $T$  is at most  $t_{l-1}$ . By Lemma 13 and Obs. 14, after iteration  $l$ , the probability that the number of uncolored vertices is more than  $t_l$

is at most  $\binom{t_{l-1}}{t_l} (K \cdot \delta^{(l)})^{t_l} \leq \exp(-\Omega(t_l)) = 1/\text{poly}(n)$ . Note that since  $\prod_{l=1}^{12} \delta^{(l)} \leq \prod_{l=1}^{12} \left( 2\sqrt{\beta}^{l-1} \cdot \sqrt{3\epsilon_i \cdot \log^2(1/\epsilon_i)} \right) \leq (2\sqrt{3})^{12} \beta^{66} \cdot \epsilon_i^6 \cdot \log^{12}(1/\epsilon) \leq \epsilon_i^5$ ,  $t_l = \epsilon^5 \Delta^*$ . By taking an union on the events over 12 iterations, we conclude w.h.p. the number of uncolored neighbors in  $T$  is at most  $t_l = \epsilon^5 \Delta^*$ .

**Completing the proof large blocks of stratum 1.** Note that stratum 1 only consists of vertices that are in layer 1. Instead of proving an analogous lemma to Lemma 9 in [5], we prove the following lemma that bounds the maximum degree on each layer-1 vertex.

► **Lemma 15.** *Let  $v \in \tilde{V}$ . By executing `ModifiedLargeDenseColoring` for  $O(1)$  rounds, w.h.p. the number of uncolored vertices in  $|N(v) \cap W_1^L|$  is at most  $(\Delta^*)^{1/20}$  for all  $v \in W_1^L$ .*

**Proof.** We execute 19 iterations of `ModifiedLargeDenseColoring`. W.h.p. for  $l \in [0, 38]$ , the invariants  $\mathcal{H}_l(S_j)$  hold for every  $S_j$  and  $\mathcal{H}_l(v)$  holds for every  $u \in S$ . Let  $T = N(v) \cap W_1^L$  be the neighbors of  $v$  in  $W_1^S$ . Let  $\delta^{(l)} = \max_{j: S_j \cap T \neq \emptyset} \delta_j^{(l)} = 2 \cdot \sqrt{D_j^{(l)}/Z_j^{(l)}} = 2\sqrt{\beta}^{l-1} \cdot \sqrt{3\epsilon_i \cdot \log^2(1/\epsilon_i)} \leq (\Delta^*)^{-1/20}/100$ .

Let  $t_0 = |T|$  and  $t_l = \max((\Delta^*)^{-1/20} t_{l-1}, (\Delta^*)^{1/20})$  for  $1 \leq l \leq 19$ . Suppose that the number of uncolored vertices in  $T$  is at most  $t_{l-1}$  at the beginning of iteration  $l$ . By Lemma 13 and Obs. 14, after iteration  $l$ , the probability that the number of uncolored vertices is more than  $t_l$  is at most  $\binom{t_{l-1}}{t_l} (K \cdot \delta^{(l)})^{t_l} \leq \exp(-\Omega(t_l)) = 1/\text{poly}(n)$ . Therefore, by an union bound on the events over the 19 iterations, w.h.p. the number of uncolored vertices in  $T$  is at most  $t_l \leq |T| \cdot (\Delta^*)^{19/20} \leq (\Delta^*)^{1/20}$ . ◀

By Lemma 15, since the maximum degree of the induced subgraph of layer 1 vertices is at most  $(\Delta^*)^{1/20}$  and  $N \cdot (\Delta^*)^{1/20} = O(n)$  (by Property (III)), a leader can collect the entire topology and the palette of each vertex and compute a coloring of the  $W_1^L$  locally.

**Coloring the Remaining Vertices – Simulation of `ColorBidding` Algorithm.** It is therefore remains to color two subsets of vertices: a subset  $U$  of vertices that were not colored by the Dense Coloring Procedures and  $V_{sp}$ . Similarly to [5], we will apply the `ColorBidding` Algorithm to first color all the vertices in  $U$ . By Lemma 10 in [5], after  $O(\log^* \Delta)$  iterations, the probability that a vertex remains uncolored is  $\exp(-\text{poly}(\Delta^*)) = 1/\text{poly}(n)$ . Then we repeat the same procedure to color vertices in  $V_{sp}$  in  $O(\log^* \Delta)$  rounds w.h.p.

The analysis is mostly straightforward from [5] and the main missing argument is in showing that a single iteration of `Alg. ColorBidding` can be simulated in  $O(1)$  rounds in the congested clique model. A simple calculation yields that each vertex selects  $\tilde{O}(\Delta^*)^{1/5}$  colors in this palette. Hence, each vertex is a target and a sender of  $\tilde{O}(\Delta^*)^{4/5} = O(n)$  messages, which fits the scheme of Lenzen's routing. A more detailed description is in the full version. The next observation follows from Lemma 2.3 in [16].

► **Observation 16.** *The list-coloring algorithm of Theorem 5 holds up to minor modifications even when every  $v \in G^*$  has at least  $\Delta^* - (\Delta^*)^{3/5}$  available colors in its palette.*

**Putting it all together.** It remains to show that the subgraph  $G^*$  from Section 2 indeed satisfies the conditions of Theorem 5 and Observation 16.

► **Lemma 17.** *(1) The subgraph  $G^*$  satisfies all the properties of Sec. 3. (2) Every  $v \in G^*$  has at least  $\max\{\deg(v, G^*) + 1, \Delta^* - (\Delta^*)^{3/5}\}$  available colors in its palette after coloring all its neighbors in  $G \setminus G^*$ .*

**Proof.** Part (1) follows by plugging the bounds of Observation 4. Next consider Claim (2). First, consider the case where  $\deg(v, G) \leq \Delta - (\Delta^* - \sqrt{5\Delta^* \cdot \log n})$ . In such case, even after coloring all neighbors of  $v$ , it still has an access of  $\Delta^* - \sqrt{5\Delta^* \cdot \log n} \geq \Delta^* - (\Delta^*)^{3/5}$  colors in its palette after coloring  $G \setminus G^*$  in the first phase. Now, consider a vertex  $v$  with  $\deg(v, G) \geq \Delta - (\Delta^* - \sqrt{5\Delta^* \cdot \log n})$ . Using Chernoff bound, w.h.p.,  $\deg(v, G^*) > (\Delta - (\Delta^* - \sqrt{\Delta^* \cdot 5 \log n})) \cdot p^* - \sqrt{5 \log n \Delta p^*} \geq \Delta^* - (\Delta^*)^{3/5}$ . A vertex  $v \in G^*$  has at least  $\deg(v, G^*) + 1$  available colors, since all its neighbors in  $G^*$  are uncolored at the beginning of the second phase and initially it was given  $(\Delta + 1)$  colors. ◀

---

## References

- 1 Leonid Barenboim and Victor Khazanov. Distributed symmetry-breaking algorithms for congested cliques. *arXiv preprint arXiv:1802.07209*, 2018.
- 2 Soheil Behnezhad, Mahsa Derakhshan, and MohammadTaghi Hajiaghayi. Brief announcement: Semi-mapreduce meets congested clique. *arXiv preprint arXiv:1802.10297*, 2018.
- 3 Andrew Berns, James Hegeman, and Sriram V Pemmaraju. Super-fast distributed algorithms for metric facility location. In *International Colloquium on Automata, Languages, and Programming*, pages 428–439. Springer, 2012.
- 4 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*, pages 11:1–11:16, 2017.
- 5 Yi-Jun Chang, Wenzheng Li, and Seth Pettie. An optimal distributed  $(\Delta + 1)$ -coloring algorithm? In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2018, Los Angeles, CA, USA*, pages 445–456. ACM, 2018.
- 6 Mohsen Ghaffari. Distributed MIS via all-to-all communication. In *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017, Washington, DC, USA, July 25-27, 2017*, pages 141–149, 2017.
- 7 Mohsen Ghaffari, Themis Gouleakis, Christian Konrad, Slobodan Mitrovic, and Ronitt Rubinfeld. Improved massively parallel computation algorithms for mis, matching, and vertex cover. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing, PODC 2018, Egham, United Kingdom*, pages 129–138, 2018.
- 8 David G Harris, Johannes Schneider, and Hsin-Hao Su. Distributed  $(\Delta + 1)$ -coloring in sublogarithmic rounds. In *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing*, pages 465–478. ACM, 2016.
- 9 James W Hegeman and Sriram V Pemmaraju. Lessons from the congested clique applied to mapreduce. *Theoretical Computer Science*, 608:268–281, 2015.
- 10 James W Hegeman, Sriram V Pemmaraju, and Vivek B Sardeshmukh. Near-constant-time distributed algorithms on a congested clique. In *International Symposium on Distributed Computing*, pages 514–530. Springer, 2014.
- 11 Tomasz Jurdzinski and Krzysztof Nowicki. MST in  $O(1)$  rounds of congested clique. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 2620–2632, 2018.
- 12 Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. Local computation: Lower and upper bounds. *Journal of the ACM (JACM)*, 63(2):17, 2016.
- 13 Christoph Lenzen. Optimal deterministic routing and sorting on the congested clique. In *Proceedings of the 2013 ACM symposium on Principles of distributed computing*, pages 42–50. ACM, 2013.



- 14 Zvi Lotker, Boaz Patt-Shamir, Elan Pavlov, and David Peleg. Minimum-weight spanning tree construction in  $O(\log \log n)$  communication rounds. *SIAM Journal on Computing*, 35(1):120–131, 2005.
- 15 Merav Parter.  $(\Delta + 1)$  coloring in the congested clique model. In *45th International Colloquium on Automata, Languages, and Programming (ICALP 2018)*, volume 107 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 160:1–160:14. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018.
- 16 Merav Parter.  $(\Delta + 1)$  coloring in the congested clique model. *arXiv preprint*, 2018. arXiv:1805.02457.
- 17 Johannes Schneider and Roger Wattenhofer. A new technique for distributed symmetry breaking. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 257–266. ACM, 2010.

## A Missing Pseudocodes for the Subroutines of [5]

**OneShotColoring.** Each uncolored vertex  $v$  decided to participate independently with probability  $p$ . Each participating vertex  $v$  selects a color  $c(v)$  from its palette  $\Psi(v)$  uniformly at random. A participating vertex  $v$  successfully colors itself if  $c(v)$  is not chosen by any vertex in  $N^*(v)$ , where  $N^*(v) = \{u \in N(v) \mid ID(u) < ID(v)\}$ .

In the ColorBidding procedure each vertex  $v$  is associated with a parameter  $p_v \geq |\Psi(v)| - \text{outdeg}(v)$  and  $p^* = \min_v p_v$ . Let  $C$  be a constant satisfying that  $\sum_{u \in N_{out}(v)} 1/p_u \leq 1/C$ .

**ColorBidding.** Each color  $c \in \Psi(v)$  is added to  $S_v$  with probability  $C/2p_v$  independently. If there exists a color  $c^* \in S_v$  that is not selected by vertices in  $N_{out}(v)$ ,  $v$  colors itself  $c^*$ .  $N_{out}(v)$  is the set of neighbors of  $v$  that have higher priority than  $v$ .

The CLP algorithm contains two versions of the dense coloring procedures. Version 1 is used to color the small blocks and version 2 is used to color the large blocks. All vertices in  $S$  agree on a parameter  $Z_{ex}$  which is a lower bound on the number of excess colors with respect to  $S$ . Each vertex  $v \in S_j$  is associated with a parameter  $D_v$ . Let  $N'(v) = \{u \in N(v) \mid D_u \leq D_v \text{ or } D_u = D_v \text{ and } ID(u) < ID(v)\}$  to be the neighbors of  $v$  with higher priority. If  $v \in S_j$ , the parameter  $D_v$  must satisfy  $|N'(v) \cap (S \setminus S_j)| \leq D_v$ . Define  $\delta_v = D_v/Z_{ex}$ .

Procedure DenseColoringStep (Version 1)

1. Let  $\pi = \{1, \dots, |S_j|\} \rightarrow S_j$  be the permutation that lists  $S_j$  in increasing order by  $D$ -value, breaking ties by ID. For  $q$  from 1 to  $|S_j|$ , the vertex  $\pi(q)$  selects a color  $c(\pi(q))$  uniformly at random from  $\Psi(\pi(q)) \setminus \{c(\pi(q')) \mid q' < q \text{ and } \{\pi(q), \pi(q')\} \in E(G)\}$ .
2. Each  $v \in S_j$  permanently colors itself  $c(v)$  if  $c(v)$  is not selected by any vertices in  $N'(v)$ .

For version 2, for each vertex  $v \in S$ , define  $N''(v)$  to be the set of vertices  $u \in N(v) \cap S$  such that (i)  $\delta_{j'} > \delta_j$  or (ii)  $\delta_{j'} = \delta_j$  and  $ID(S_{j'}) < ID(S_j)$ , where  $v \in S_j$  and  $u \in S_{j'}$ .

**39:18 Randomized  $(\Delta + 1)$ -Coloring in  $O(\log^* \Delta)$  Congested Clique Rounds**

Procedure DenseColoringStep (Version 2)

1. Each cluster  $S_j$  selects  $(1 - \delta_j)|S_j|$  vertices u.a.r. and generates a permutation  $\pi$  of those vertices u.a.r. The vertex  $\pi(q)$  selects a color  $c(\pi(q))$  u.a.r. from

$$\Psi(\pi(q)) \setminus \{c(\pi(q')) \mid q' < q \text{ and } \{\pi(q), \pi(q')\} \in E(G)\}.$$

2. Each  $v \in S_j$  that has selected a color  $c(v)$  permanently colors itself  $c(v)$  if  $c(v)$  is not selected by any vertices  $u \in N''(v)$ .

# Congested Clique Algorithms for Graph Spanners

**Merav Parter**

Weizmann IS, Rehovot, Israel  
merav.parter@weizmann.ac.il

**Eylon Yogev**

Weizmann IS, Rehovot, Israel  
eylon.yogev@weizmann.ac.il

---

## Abstract

Graph spanners are sparse subgraphs that faithfully preserve the distances in the original graph up to small stretch. Spanners have been studied extensively as they have a wide range of applications ranging from distance oracles, labeling schemes and routing to solving linear systems and spectral sparsification. A  $k$ -spanner maintains pairwise distances up to multiplicative factor of  $k$ . It is a folklore that for every  $n$ -vertex graph  $G$ , one can construct a  $(2k - 1)$  spanner with  $O(n^{1+1/k})$  edges. In a distributed setting, such spanners can be constructed in the standard CONGEST model using  $O(k^2)$  rounds, when randomization is allowed.

In this work, we consider spanner constructions in the congested clique model, and show:

- a randomized construction of a  $(2k - 1)$ -spanner with  $\tilde{O}(n^{1+1/k})$  edges in  $O(\log k)$  rounds. The previous best algorithm runs in  $O(k)$  rounds;
- a deterministic construction of a  $(2k - 1)$ -spanner with  $\tilde{O}(n^{1+1/k})$  edges in  $O(\log k + (\log \log n)^3)$  rounds. The previous best algorithm runs in  $O(k \log n)$  rounds. This improvement is achieved by a new derandomization theorem for hitting sets which might be of independent interest;
- a deterministic construction of a  $O(k)$ -spanner with  $O(k \cdot n^{1+1/k})$  edges in  $O(\log k)$  rounds.

**2012 ACM Subject Classification** Theory of computation → Distributed algorithms

**Keywords and phrases** Distributed Graph Algorithms, Spanner, Congested Clique

**Digital Object Identifier** 10.4230/LIPIcs.DISC.2018.40

**Related Version** A full version of the paper is available at [21], <https://arxiv.org/abs/1805.05404>.

**Acknowledgements** We are grateful to Mohsen Ghaffari for earlier discussions on congested-clique spanners via streaming ideas. We thank Roei Tell for pointing out [15].

## 1 Introduction & Related Work

Graph spanners introduced by Peleg and Schäffer [23] are fundamental graph structures, more precisely, subgraphs of an input graph  $G$ , that faithfully preserve the distances in  $G$  up to small multiplicative stretch. Spanners have a wide-range of distributed applications [22] for routing [27], broadcasting, synchronizers [24], and shortest-path computations [3].

The common objective in distributed computation of spanners is to achieve the best-known existential size-stretch trade-off within small number of *rounds*. It is a folklore that for every graph  $G = (V, E)$ , there exists a  $(2k - 1)$ -spanner  $H \subseteq G$  with  $O(n^{1+1/k})$  edges. Moreover, this size-stretch tradeoff is believed to be optimal, by the girth conjecture of Erdős.

There are plentiful of distributed constructions of spanners for both the LOCAL and the CONGEST models of distributed computing [8, 2, 9, 10, 11, 25, 12, 16]. The standard setting is a synchronous message passing model where per round each node can send one



© Merav Parter and Eylon Yogev;

licensed under Creative Commons License CC-BY

32nd International Symposium on Distributed Computing (DISC 2018).

Editors: Ulrich Schmid and Josef Widder; Article No. 40; pp. 40:1–40:18

Leibniz International Proceedings in Informatics



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

message to each of its neighbors. In the LOCAL model, the message size is unbounded, while in the CONGEST model it is limited to  $O(\log n)$  bits. One of the most notable distributed randomized constructions of  $(2k - 1)$  spanners is by Baswana & Sen [2] which can be implemented in  $O(k^2)$  rounds in the CONGEST model.

Currently, there is an interesting gap between deterministic and randomized constructions in the CONGEST model, or alternatively between the deterministic construction of spanners in the LOCAL vs. the CONGEST model. Whereas the deterministic round complexity of  $(2k - 1)$  spanners in the LOCAL model is  $O(k)$  due to [10], the best deterministic algorithm in the CONGEST model takes  $O(2\sqrt{\log n \cdot \log \log n})$  rounds [13].

We consider the *congested clique* model, introduced by Lotker et al. [20]. In this model, in every round, each vertex can send  $O(\log n)$  bits to each of the vertices in the graph. The congested clique model has been receiving a lot of attention recently due to its relevance to overlay networks and large scale distributed computation [17, 14, 4].

**Deterministic local computation in the congested clique model.** Censor et al. [7] initiated the study of *deterministic* local algorithms in the congested clique model by means of derandomization of randomized LOCAL algorithms. The approach of [7] can be summarized as follows. The randomized complexity of the classical local problems is  $\text{polylog}(n)$  rounds (in both LOCAL and CONGEST models). For these randomized algorithms, it is usually sufficient that the random choices made by vertices are sampled from distributions with *bounded independence*. Hence, any round of a randomized algorithm can be simulated by giving all nodes a shared random seed of  $\text{polylog}(n)$  bits.

To completely derandomize such a round, nodes should compute (deterministically) a seed which is at least as “good”<sup>1</sup> as a random seed would be. This is achieved by estimating their “local progress” when simulating the random choices using that seed. Combining the techniques of conditional expectation, pessimistic estimators and bounded independence, leads to a simple “voting”-like algorithm in which the bits of the seed are computed *bit-by-bit*. The power of the congested clique is hence in providing some global leader that collects all votes in 1 round and broadcasts the winning bit value. This approach led to deterministic MIS in  $O(\log \Delta \log n)$  rounds and deterministic  $(2k - 1)$  spanners with  $\tilde{O}(n^{1+1/k})$  edges in  $O(k \log n)$  rounds, which also works for weighted graphs. Barenboim and Khazanov [1] presented deterministic local algorithms as a function of the graph’s *arboricity*.

**Deterministic spanners via derandomization of hitting sets.** As observed by [26, 5, 13], the derandomization of the Baswana-Sen algorithm boils down into a derandomization of  $p$ -dominating sets or *hitting-sets*. It is a well known fact that given a collection of  $m$  sets  $\mathcal{S}$ , each containing at least  $\Delta$  elements coming from a universe of size  $n$ , one can construct a hitting set  $Z$  of size  $O((n \log m)/\Delta)$ . A randomized construction of such a set is immediate by picking each element into  $Z$  with probability  $p$  and applying Chernoff. A centralized deterministic construction is also well known by the greedy approach (e.g., Lemma 2.7 of [5]).

In our setting we are interested in deterministic constructions of hitting sets in the congested clique model. In this setting, each vertex  $v$  knows a subset  $S_v$  of size at least  $\Delta$ , that consists of vertices in the  $O(k)$ -neighborhood of  $v$ , and it is required to compute a small set  $Z$  that hits (i.e., intersects) all subsets. Censor et al. [7] showed that the above mentioned randomized construction of hitting sets still holds with  $g = O(\log n)$ -wise independence,

<sup>1</sup> The random seed is usually shown to provide a large progress in expectation. The deterministically computed seed should provide a progress at least as large as the expected progress of a random seed.

■ Table 1

	Stretch	#Rounds	Type
Adaptation of Baswana & Sen [2]	$2k - 1$	$O(k)$	Randomized
This Work	$2k - 1$	$O(\log k)$	
Censor-Hillel et al. [7]	$2k - 1$	$O(k \log n)$	Deterministic
This Work	$2k - 1$	$O(\log k + (\log \log n)^3)$	
This Work	$O(k)$	$O(\log k)$	

and presented an  $O(g)$ -round algorithm that computes a hitting set deterministically by finding a good seed of  $O(g \log n)$  bits. Applying this hitting-set algorithm for computing the  $k$  levels of Baswana-Sen’s clustering yields a deterministic algorithm for  $(2k - 1)$  spanners with  $O(k \log n)$  rounds.

## Our Results and Approach in a Nutshell

We provide improved randomized and deterministic constructions of graph spanners in the congested clique model. Our randomized solution is based on an  $O(\log k)$ -round algorithm that computes the  $O(\sqrt{n})$  nearest vertices in radius  $k/2$  for every vertex  $v^2$ . This induces a partitioning of the graph into sparse and dense regions. The sparse region is solved “locally” and the dense region simulates only two phases of Baswana-Sen, leading to a total round complexity of  $O(\log k)$ . We show the following for  $n$ -vertex unweighted graphs.

► **Theorem 1.** *There exists a randomized algorithm in the congested clique model that constructs a  $(2k - 1)$ -spanner with  $\tilde{O}(k \cdot n^{1+1/k})$  edges within  $O(\log k)$  rounds w.h.p.*

Our deterministic algorithms are based on constructions of hitting-sets with short seeds. Using the pseudorandom generator of Gopalan et al. [15], we construct a hitting set with seed length  $O(\log n \cdot (\log \log n)^3)$  which yields the following for  $n$ -vertex unweighted graphs.

► **Theorem 2.** *There exists a deterministic algorithm in the congested clique model that constructs a  $(2k - 1)$ -spanner with  $\tilde{O}(k \cdot n^{1+1/k})$  edges within  $O(\log k + (\log \log n)^3)$  rounds.*

In addition, we also show that if one settles for stretch of  $O(k)$ , then a hitting-set seed of  $O(\log n)$  bits is sufficient for this purpose, yielding the following construction:

► **Theorem 3.** *There exists a deterministic algorithm in the congested clique model that constructs a  $O(k)$ -spanner with  $O(k \cdot n^{1+1/k})$  edges within  $O(\log k)$  rounds.*

A summary of our results are given in the Table 1. All results in the table are with respect to spanners with  $\tilde{O}(n^{1+1/k})$  edges for an unweighted  $n$ -vertex graph  $G$ . All these bounds are for the congested clique model<sup>3</sup>.

In what follows we provide some technical background and then present the high level ideas of these construction.

<sup>2</sup> To be more precise, the algorithm computes the  $O(n^{1/2-1/k})$  nearest vertices at distance at most  $k/2 - 1$ .

<sup>3</sup> Baswana-Sen [2] does not mention the congested clique model, but the best randomized solution in the congested clique is given by simulating [2].

**A brief exposition of Baswana-Sen [2].** The algorithm is based on constructing  $k$  levels of clustering  $\mathcal{C}_0, \dots, \mathcal{C}_{k-1}$ , where a clustering  $\mathcal{C}_i = \{C_{i,1}, \dots\}$  consists of vertex disjoint subsets which we call clusters. Every cluster  $C \in \mathcal{C}_i$  has a special node that we call *cluster center*. For each  $C \in \mathcal{C}_i$ , the spanner contains a depth- $i$  tree rooted at its center and spanning all cluster vertices. Starting with the trivial clustering  $\mathcal{C}_0 = \{\{v\}, v \in V\}$ , in each phase  $i$ , the algorithm is given a clustering  $\mathcal{C}_i$  and it computes a clustering  $\mathcal{C}_{i+1}$  by sampling the cluster center of each cluster in  $\mathcal{C}_i$  with probability  $n^{-1/k}$ . Vertices that are adjacent to the sampled clusters join them and the remaining vertices become unclustered. For the latter, the algorithm adds some of their edges to the spanner. This construction yields a  $(2k - 1)$  spanner with  $O(kn^{1+1/k})$  edges in expectation.

It is easy to see that this algorithm can be simulated in the congested clique model using  $O(k)$  rounds. As observed in [26, 16], the only randomized step in Baswana-Sen is picking the cluster centers of the  $(i + 1)^{th}$  clustering. That is, given the  $n^{1-i/k}$  cluster centers of  $\mathcal{C}_i$ , it is required to compute a subsample of  $n^{1-(i+1)/k}$  clusters without having to add too many edges to the spanner (due to unclustered vertices). This is exactly the hitting-set problem where the neighboring clusters of each vertex are the *sets* that should be covered, and the *universe* is the set of centers in  $\mathcal{C}_i$  (ideas along these lines also appear in [26, 13]).

**Our Approach.** In the following, we provide the high level description of our construction while omitting many careful details and technicalities. We note that some of these technicalities stems from the fact that we insist on achieving the (nearly) optimal spanners, as commonly done in this area. Settling for an  $O(k)$ -spanner with  $\tilde{O}(kn^{1+1/k})$  edges could considerably simplify the algorithm and its analysis. The high-level idea is simple and it is based on dividing the graph  $G$  into sparse edges and dense edges, constructing a spanner for each of these subgraphs using two different techniques. This is based on the following intuition inspired by the Baswana-Sen algorithm.

In Baswana-Sen, the vertices that are clustered in level- $i$  of the clustering are vertices whose  $i$ -neighborhood is sufficiently *dense*, i.e., contains at least  $n^{i/k}$  vertices. We then divide the vertices into *dense* vertices  $V_{dense}$  and *sparse* vertices  $V_{sparse}$ , where  $V_{dense}$  consists of vertices that have  $\Omega(\sqrt{n})$  vertices in their  $k/2$ -ball, and  $V_{sparse}$  consists of the remaining vertices. This induces a partitioning of  $G$  edges into  $E_{sparse} = (V_{sparse} \times V) \cap E(G)$  and  $E_{dense}$  that contains the remaining  $G$ -edges, i.e., edges whose both endpoints are dense.

**Collecting Topology of Closed Neighborhood.** One of the key-building blocks of our construction is an  $O(\log k)$ -round algorithm that computes for each vertex  $u$  the subgraph  $G_{k/2}(u)$  induced on its closest  $O(\sqrt{n})$  vertices within distance at most  $k/2$  in  $G$ . Hence the algorithm computes the entire  $k/2$ -neighborhoods for the sparse vertices. For the sake of the following discussion, assume that the maximum degree in  $G$  is  $O(\sqrt{n})$ . Our algorithm handles the general case as well. Intuitively, collecting the  $k/2$ -neighborhood can be done in  $O(\log k)$  rounds if the graph is sufficiently *sparse* by employing the graph exponentiation idea of [19]. In this approach, in each phase the radius of the collected neighborhood is doubled. Employing this technique in our setting gives rise to several issues. First, the input graph  $G$  is not entirely sparse but rather consists of interleaving sparse and dense regions, i.e., the  $k/2$ -neighborhood of a sparse vertex might contain dense vertices. For that purpose, in phase  $i$  of our algorithm, each vertex (either sparse or dense) should obtain a subset of its closest  $O(\sqrt{n})$  vertices in its  $2^i$  neighborhood. Limiting the amount collected information is important for being able to route this information via Lenzen's algorithm [18] in  $O(1)$  rounds in each phase.

Another technicality concerns the fact that the relation “ $u$  is in the  $\sqrt{n}$  nearest vertices to  $v$ ” is not necessarily symmetric. This entitles a problem where a given vertex  $u$  is “close”<sup>4</sup> to many vertices  $w$ , and  $u$  is not close to any of these vertices. In case where these  $w$  vertices need to receive the information from  $u$  regarding its closest neighbors (i.e., where some their close vertices are close to  $u$ ),  $u$  ends up sending too many messages in a single phase. To overcome this, we carefully set the growth of the radius of the collected neighborhood in the graph exponentiation algorithm. We let only vertices that are close to each other exchange their topology information and show that this is sufficient for computing the  $G_{k/2}(u)$  subgraphs. This procedure is the basis for our constructions as explained next.

**Handling the Sparse Region.** The idea is to let every sparse vertex  $u$  locally simulate a LOCAL spanner algorithm on its subgraph  $G_{k/2}(u)$ . For that purpose, we show that the deterministic spanner algorithm of [10] which takes  $k$  rounds in general, in fact requires only  $k/2$  rounds when running by a sparse vertex  $u$ . At the end of these  $k/2$  rounds, for each spanner edge  $(u, v)$ , at least one of the endpoints know that this edge is in the spanner. This implies that the subgraph  $G_{k/2}(u)$  contains all the information needed for  $u$  to locally simulate the spanner algorithm. This seemingly harmless approach has a subtle defect. Letting only the sparse vertices locally simulate a spanner algorithm might lead to a case where a certain edge  $(u, v)$  is not added by a sparse vertex due to a decision made by a dense vertex  $w$  in the local simulation  $u$  in  $G_{k/2}(u)$ . Since  $w$  is a dense vertex it did not run the algorithm locally and hence is not aware of adding these edges<sup>5</sup>. To overcome this, the sparse vertices notify the dense vertices about their edges added in their local simulations. We show how to do it in  $O(1)$  rounds.

**Handling the Dense Region.** In the following, we settle for stretch of  $(2k + 1)$  for ease of description. By applying the topology collecting procedure, every dense vertex  $v$  computes the set  $N_{k/2}(v)$  consisting of its closest  $\Theta(\sqrt{n})$  vertices within distance  $k/2$ . The main benefit in computing these  $N_{k/2}(v)$  sets, is that it allows the dense vertices to “skip” over the first  $k/2 - 1$  phases of Baswana-Sen, ready to apply the  $(k/2)$  phase.

As described earlier, picking the centers of the clusters can be done by computing a hitting set for the set  $\mathcal{S} = \{N_{k/2}(v) \mid v \in V_{dense}\}$ . It is easy to construct a random subset  $Z \subseteq V$  of cardinality  $O(n^{1/2})$  that hits all these sets and to cluster all the dense vertices around this  $Z$  set. This creates clusters of strong diameter  $k$  (in the spanner) that cover all the dense vertices. The final step connects each pair of adjacent clusters by adding to the spanner a single edge between each such pair, this adds  $|Z|^2 = O(n)$  edges to the spanner.

**Hitting Sets with Short Seed.** The description above used a randomized solution to the following hitting set problem: given  $n$  subsets of vertices  $S_1, \dots, S_n$ , each  $|S_i| \geq \Delta$ , find a small set  $Z$  that intersects all  $S_i$  sets. A simple randomized solution is to choose each node  $v$  to be in  $Z$  with probability  $p = O(\log n/\Delta)$ . The standard approach for derandomization is by using distributions with limited independence. Indeed, for the randomized solution to hold, it is sufficient to sample the elements from a  $\log n$ -wise distribution. However, sampling an element with probability  $p = O(\log n/\Delta)$  requires roughly  $\log n$  random bits, leading to a total seed length of  $(\log^2 n)$ , which is too large for our purposes.

<sup>4</sup> By *close* we mean being among the  $\sqrt{n}$  nearest vertices.

<sup>5</sup> If we “add” one more round and simulate  $k/2 + 1$  rounds, then there is no such problem as both endpoints of a spanner edge know that the edge is in the spanner. However, we could only collect the information up to radius  $k/2$ .



Our key observation is that for any set  $S_i$  the event that  $S_i \cap Z \neq \emptyset$  can be expressed by a *read-once DNF formula*. Thus, in order to get a short seed it suffices to have a pseudorandom generator (PRG) that can “fool” read-once DNFs. A PRG is a function that gets a short random seed and expands it to a long one which is indistinguishable from a random seed of the same length for such a formula. Luckily, such PRGs with seed length of  $O(\log n \cdot (\log \log n)^3)$  exist due to Gopalan et al. [15], leading to deterministic hitting-set algorithm with  $O((\log \log n)^3)$  rounds.

**Graph Notations.** For a vertex  $v \in V(G)$ , a subgraph  $G'$  and an integer  $\ell \in \{1, \dots, n\}$ , let  $\Gamma_\ell(v, G') = \{u \mid \text{dist}(u, v, G') \leq \ell\}$ . When  $\ell = 1$ , we omit it and simply write  $\Gamma(v, G')$ , also when the subgraph  $G'$  is clear from the context, we omit it and write  $\Gamma_\ell(v)$ . For a subset  $V' \subseteq V$ , let  $G[V']$  be the induced subgraph of  $G$  on  $V'$ . Given a disjoint subset of vertices  $C, C'$ , let  $E(C, C', G) = \{(u, v) \in E(G) \mid u \in C \text{ and } v \in C'\}$ . We say that  $C$  and  $C'$  are *adjacent* if  $E(C, C', G) \neq \emptyset$ . Also, for  $v \in V$ ,  $E(v, C, G) = \{(u, v) \in E(G) \mid u \in C\}$ . A vertex  $u$  is *incident* to a subset  $C$ , if  $E(v, C, G) \neq \emptyset$ .

**Road-Map.** Section 2 presents algorithm `NearestNeighbors` to collect the topology of nearby vertices. At the end of this section, using this collected topology, the graph is partitioned into sparse and dense subgraphs. Section 3 describes the spanner construction for the sparse regime. Section 4 considers the dense regime and is organized as follows. First, Section 4.1 describes a deterministic construction spanner given an hitting-set algorithm as a black box. Then, Section 5 fills in this missing piece and shows deterministic constructions of small hitting-sets via derandomization. Finally, Section 5.3 provides an alternative deterministic construction, with improved runtime but larger stretch.

## 2 Collecting Topology of Nearby Neighborhood

For simplicity of presentation, assume that  $k$  is even, for  $k$  odd, we replace the term  $(k/2 - 1)$  with  $\lfloor k/2 \rfloor$ . In addition, we assume  $k \geq 6$ . Note that randomized constructions with  $O(k)$  rounds are known and hence one benefits from an  $O(\log k)$  algorithm for a non-constant  $k$ . In the full version, we show the improved deterministic constructions for  $k \in \{2, 3, 4, 5\}$ .

### 2.1 Computing Nearest Vertices in the $(k/2 - 1)$ Neighborhoods

In this subsection, we present an algorithm that computes the  $n^{1/2-1/k}$  nearest vertices with distance  $k/2 - 1$  for every vertex  $v$ . This provides the basis for the subsequent procedures presented later on. Unfortunately, computing the nearest vertices of each vertex might require many rounds when  $\Delta = \omega(\sqrt{n})$ . In particular, using Lenzen’s routing<sup>6</sup>[18], in the congested clique model, the vertices can learn their 2-neighborhoods in  $O(1)$  rounds, when the maximum degree is bounded by  $O(\sqrt{n})$ . Consider a vertex  $v$  that is incident to a heavy vertex  $u$  (of degree at least  $\Omega(\sqrt{n})$ ). Clearly  $v$  has  $\Omega(n^{1/2-1/k})$  vertices at distance 2, but it is not clear how  $v$  can learn their identities. Although,  $v$  is capable of receiving  $O(n^{1/2-1/k})$  messages, the heavy neighbor  $u$  might need to send  $n^{1/2-1/k}$  messages to each of its neighbors, thus  $\Omega(n^{3/2-1/k})$  messages in total. To avoid this, we compute the  $n^{1/2-1/k}$  nearest vertices in a *lighter* subgraph  $G_{\text{light}}$  of  $G$  with maximum degree  $\sqrt{n}$ . The neighbors of heavy vertices might not learn their 2-neighborhood and would be handled slightly differently in Section 4.

<sup>6</sup> Lenzen’s routing can be viewed as a  $O(1)$ -round algorithm applied when each vertex  $v$  is a target and a sender of  $O(n)$  messages.

► **Definition 4.** A vertex  $v$  is *heavy* if  $\deg(v, G) \geq \sqrt{n}$ , the set of heavy vertices is denoted by  $V_{heavy}$ . Let  $G_{light} = G[V \setminus V_{heavy}]$ .

► **Definition 5.** For each vertex  $u \in V(G_{light})$  define  $N_{k/2-1}(u)$  to be the set of  $y(u) = \min\{n^{1/2-1/k}, |\Gamma_{k/2-1}(u, G_{light})|\}$  closest vertices at distance at most  $(k/2 - 1)$  from  $u$  (breaking ties based on IDs) in  $G_{light}$ . Define  $T_{k/2-1}(u)$  to be the truncated BFS tree rooted at  $u$  consisting of the  $u$ - $v$  shortest path in  $G_{light}$ , for every  $v \in N_{k/2-1}(u)$ .

► **Lemma 6.** *There exists a deterministic algorithm NearestNeighbors that within  $O(\log k)$  rounds, computes the truncated BFS tree  $T_{k/2-1}(u)$  for each vertex  $u \in V(G_{light})$ . That is, after running Alg. NearestNeighbors, each  $u \in V(G_{light})$  knows the entire tree  $T_{k/2-1}(u)$ .*

**Algorithm NearestNeighbors.** For every integer  $j \geq 0$ , we say that a vertex  $u$  is  $j$ -sparse if  $|\Gamma_j(u, G_{light})| \leq n^{1/2-1/k}$ , otherwise we say it is  $j$ -dense. The algorithm starts by having each non-heavy vertex compute  $\Gamma_2(u, G_{light})$  in  $O(1)$  rounds using Lenzen's algorithm. This is the only place where it is important that we work on  $G_{light}$  rather than on  $G$ . Next, in each phase  $i \geq 1$ , vertex  $u$  collects information on vertices in its  $\gamma(i+1)$ -ball in  $G_{light}$ , where:

$$\gamma(1) = 2, \text{ and } \gamma(i+1) = \min\{2\gamma(i) - 1, k/2\}, \text{ for every } i \in \{1, \dots, \lceil \log(k/2) \rceil\}.$$

At phase  $i \in \{1, \dots, \lceil \log(k/2) \rceil\}$  the algorithm maintains the invariant that a vertex  $u$  holds a partial BFS tree  $\widehat{T}_i(u)$  in  $G_{light}$  consisting of the vertices  $\widehat{N}_i(u) := V(\widehat{T}_i(u))$ , such that:

(I1) For an  $\gamma(i)$ -sparse vertex  $u$ ,  $\widehat{N}_i(u) = \Gamma_{\gamma(i)}(u)$ .

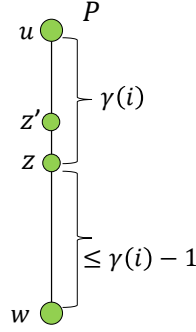
(I2) For an  $\gamma(i)$ -dense vertex  $u$ ,  $\widehat{N}_i(u)$  consists of the closest  $n^{1/2-1/k}$  vertices to  $u$  in  $G_{light}$ .

Note that in order to maintain the invariant in phase  $(i+1)$ , it is only required that in phase  $i$ , the  $\gamma(i)$ -sparse vertices would collect the relevant information, as for the  $\gamma(i)$ -dense vertices, it already holds that  $\widehat{N}_{i+1}(u) = \widehat{N}_i(u)$ . In phase  $i$ , each vertex  $v$  (regardless of being sparse or dense) sends its partial BFS tree  $\widehat{T}_i(v)$  to each vertex  $u$  only if (1)  $u \in \widehat{N}_i(v)$  and (2)  $v \in \widehat{N}_i(u)$ . This condition can be easily checked in a single round, as every vertex  $u$  can send a message to all the vertices in its set  $\widehat{N}_i(u)$ . Let  $\widehat{N}'_{i+1}(u) = \bigcup_{v \in \widehat{N}_i(u) \mid u \in \widehat{N}_i(v)} \widehat{N}_i(v)$  be the subset of all received  $\widehat{N}_i$  sets at vertex  $u$ . It then uses the distances to  $\widehat{N}_i(u)$ , and the received distances to the vertices in the  $\widehat{N}_i$  sets, to compute the shortest-path distance to each  $w \in \widehat{N}_i(v)$ . As a result it computes the partial tree  $\widehat{T}_{i+1}(u)$ . The subset  $\widehat{N}_{i+1}(u) \subseteq \widehat{N}'_{i+1}(u)$  consists of the (at most  $n^{1/2-1/k}$ ) vertices within distance  $\gamma(i+1)$  from  $u$ . This completes the description of phase  $i$ . We next analyze the algorithm and show that each phase can be implemented in  $O(1)$  rounds and that the invariant on the  $\widehat{T}_i(u)$  trees is maintained.

**Analysis.** We first show that phase  $i$  can be implemented in  $O(1)$  rounds. Note that by definition,  $|\widehat{N}_i(u)| \leq \sqrt{n}$  for every  $u$ , and every  $i \geq 1$ . Hence, by the condition of phase  $i$ , each vertex sends  $O(n)$  messages and receives  $O(n)$  messages, which can be done in  $O(1)$  rounds, using Lenzen's routing algorithm [18].

We show that the invariant holds, by induction on  $i$ . Since all vertices first collected their second neighborhood, the invariant holds<sup>7</sup> for  $i = 1$ . Assume it holds up to the beginning of phase  $i$ , and we now show that it holds in the beginning of phase  $i+1$ . If  $u$  is  $\gamma(i)$ -dense, then  $u$  should not collect any further information in phase  $i$  and the assertion holds trivially.

<sup>7</sup> This is the reason why we consider only  $G_{light}$ , as otherwise  $\gamma(1) = 0$  and we would not have any progress.



■ **Figure 1** Shown is a path  $P$  between  $u$  and  $w$  where  $z$  is the first dense vertex on the  $\gamma(i)$ -length prefix of  $P$ . If  $u \notin \widehat{N}_i(z)$  then  $u, w \in \widehat{N}_i(z')$ .

Consider an  $\gamma(i)$ -sparse vertex  $u$  and let  $N_{\gamma(i+1)}(u)$  be the target set of the  $n^{1/2-1/k}$  closest vertices at distance  $\gamma(i+1)$  from  $u$ . We will fix  $w \in N_{\gamma(i+1)}(u)$ , and show that  $w \in \widehat{N}_{i+1}(u)$  and in addition,  $u$  has computed the shortest path to  $w$  in  $G_{light}$ . Let  $P$  be  $u$ - $w$  shortest path in  $G_{light}$ . If all vertices  $z$  on the  $\gamma(i)$ -length prefix of  $P$  are  $\gamma(i)$ -sparse, then the claim holds as  $z \in \widehat{N}_i(u)$ ,  $u \in \widehat{N}_i(z)$ , and  $w \in \widehat{N}_i(z')$  where  $z'$  is the last vertex on the  $\gamma(i)$ -length prefix of  $P$ . Hence, by the induction assumption for the  $\widehat{N}_i$  sets,  $u$  can compute in phase  $i$  its shortest-path to  $w$ .

We next consider the remaining case where not all the vertices on the  $\gamma(i)$ -length path are sparse. Let  $z \in \widehat{N}_i(u)$  be the first  $\gamma(i)$ -dense vertex (closest to  $u$ ) on the  $\gamma(i)$ -length prefix of  $P$ . Observe that  $w \in \widehat{N}_i(z)$ . Otherwise,  $\widehat{N}_i(z)$  contains  $n^{1/2-1/k}$  vertices that are closer to  $z$  than  $w$ , which implies that these vertices are also closer to  $u$  than  $w$ , and hence  $w$  should not be in  $N_{\gamma(i+1)}(u)$  (as it is not among the closest  $n^{1/2-1/k}$  vertices to  $u$ ), leading to contradiction. Thus, if also  $u \in \widehat{N}_i(z)$ , then  $z$  sends to  $u$  in phase  $i$  its shortest-path to  $w$ . By the induction assumption for the  $\widehat{N}_i(u), \widehat{N}_i(z)$  sets, we have that  $u$  has the entire shortest-path to  $w$ . It remains to consider the case where the first  $\gamma(i)$ -dense vertex on  $P$ ,  $z$ , does not contain  $u$  in its  $\widehat{N}_i(z)$  set, hence it did not send its information on  $w$  to  $u$  in phase  $i$ . Denote  $x = \text{dist}(u, z, G_{light})$  and  $y = \text{dist}(z, w, G_{light})$ , thus  $x + y = |P| \leq 2\gamma(i) - 1$ . Since  $w \in \widehat{N}_i(z)$  but  $u \notin \widehat{N}_i(z)$ , we have that  $y \leq x$  and  $2y \leq |P|$ , which implies that  $y \leq \gamma(i) - 1$ . Let  $z'$  be the vertex preceding  $z$  on the  $P$  path, hence  $z'$  also appear on the  $\gamma(i)$ -length prefix of  $P$  and  $z' \in N_i(u)$ . By definition,  $z'$  is  $\gamma(i)$ -sparse and it also holds that  $u \in \widehat{N}_i(z')$ . Since  $\text{dist}(z', w, G_{light}) = y + 1 \leq \gamma(i)$ , it holds that  $w \in \widehat{N}_i(z')$ . Thus,  $u$  can compute the  $u$ - $w$  shortest-path using the  $z'$ - $w$  shortest-path it has received from  $z'$ . For an illustration, see Figure 1.

## 2.2 Dividing $G$ into Sparse and Dense Regions

During the execution of NearestNeighbors every non-heavy vertex  $v$  computes the sets  $N_{k/2-1}(v)$  and the corresponding tree  $T_{k/2-1}(v)$ . The vertices are next divided into dense vertices  $V_{dense}$  and sparse vertices  $V_{sparse}$ . Roughly speaking, the dense vertices are those that have at least  $n^{1/2-1/k}$  vertices at distance at most  $k/2 - 1$  in  $G$ . Since the subsets of nearest neighbors are computed in  $G_{light}$  rather than in  $G$ , this vertex division is more delicate.

► **Definition 7.** A vertex  $v$  is *dense* if either (1) it is heavy, (2) a neighbor of a heavy vertex or (3)  $|\Gamma_{k/2-1}(v, G_{light})| > n^{1/2-1/k}$ . Otherwise, a vertex is *sparse*. Let  $V_{dense}, V_{sparse}$  be the dense (resp., sparse) vertices in  $V$ .

► **Observation 8.** For  $k \geq 6$ , for every dense vertex  $v$  it holds that  $|\Gamma_{k/2-1}(v, G)| \geq n^{1/2-1/k}$ .

The edges of  $G$  are partitioned into:

$$E_{dense} = ((V_{dense} \times V_{dense}) \cap E(G)), \quad E_{sparse} = (V_{sparse} \times V) \cap E(G)$$

Since all the neighbors of heavy vertices are dense, it also holds that  $E_{sparse} = (V_{sparse} \times (V \setminus V_{heavy})) \cap E(G_{light})$ .

**Overview of the Spanner Constructions.** The algorithm contains two subprocedures, the first takes care of the sparse edge-set by constructing a spanner  $H_{sparse} \subseteq G_{sparse}$  and the second takes care of the dense edge-set by constructing  $H_{dense} \subseteq G$ . Specifically, these spanners will satisfy that for every  $e = (u, v) \in G_i$ ,  $\text{dist}(u, v, H_i) \leq 2k - 1$  for  $i \in \{sparse, dense\}$ . We note that the spanner  $H_{dense} \subseteq G$  rather than being contained in  $G_{dense}$ . The reason is that the spanner  $H_{dense}$  might contain edges incident to sparse vertices as will be shown later. The computation of the spanner  $H_{sparse}$  for the sparse edges,  $E_{sparse}$ , is done by letting each sparse vertex locally simulate a local spanner algorithm. The computation of  $H_{dense}$  is based on applying two levels of clustering as in Baswana-Sen. The selection of the cluster centers will be made by applying an hitting-set algorithm.

### 3 Handling the Sparse Subgraph

In the section, we construct the spanner  $H_{sparse}$  that will provide a bounded stretch for the sparse edges. As we will see, the topology collected by applying Alg. `NearestNeighbors` allows every sparse vertex to *locally* simulate a deterministic spanner algorithm in its collected subgraph, and deciding which of its edges to add to the spanner based on this local view.

Recall that for every sparse vertex  $v$  it holds that  $|\Gamma_{k/2-1}(v, G_{light})| \leq n^{1/2-1/k}$  where  $G_{light} = G[V \setminus V_{heavy}]$  and that  $E_{sparse} = (V_{sparse} \times V) \cap E(G)$ . Let  $G_{sparse}(u) = G_{sparse}[\Gamma_{k/2-1}(u, G)]$ . By applying Alg. `NearestNeighbors`, and letting sparse vertices send their edges to the sparse vertices in their  $(k/2 - 1)$  neighborhoods in  $G_{light}$ , we have:

► **Claim 9.** *There exists a  $O(\log k)$ -round deterministic algorithm, that computes for each sparse vertex  $v$  its subgraph  $G_{sparse}(v)$ .*

Our algorithm is based on an adaptation of the local algorithm of [10], which is shown to satisfy the following in our context. The proof is in the full version [21].

► **Lemma 10.** *There exists a deterministic algorithm `LocalSpanner` that constructs a  $(k - 3)$  spanner in the LOCAL model, such that every sparse vertex  $u$  decides about its spanner edges within  $k/2 - 1$  rounds. In particular,  $u$  can simulate Alg. `LocalSpanner` locally on  $G_{sparse}$  and for every edge  $(u, z)$  not added to the spanner  $H_{sparse}$ , there is a path of length at most  $(k - 3)$  in  $G_{sparse}(u) \cap H_{sparse}$ .*

A useful property of the algorithm<sup>8</sup> by Derbel et al. (Algorithm 1 in [10]) is that if a vertex  $v$  did not terminate after  $i$  rounds, then it must hold that  $|\Gamma_i(v, G)| \geq n^{i/k}$ . Thus in our context, every sparse vertex terminates after at most  $k/2 - 1$  rounds<sup>9</sup>. We also show that for

<sup>8</sup> This algorithm works only for unweighted graphs and hence our deterministic algorithms are for unweighted graphs. Currently, there are no local deterministic algorithms for weighted graphs.

<sup>9</sup> By definition we have that  $|\Gamma_{k/2-1}(u, G_{light})| \leq n^{1/2-1/k}$ . Moreover, since  $G_{sparse} \subseteq G_{light}$  it also holds that  $|\Gamma_{k/2-1}(u, G_{sparse})| \leq n^{1/2-1/k}$ .

simulating these  $(k/2 - 1)$  rounds of Alg. LocalSpanner by  $u$ , it is sufficient for  $u$  to know all the neighbors of its  $(k/2 - 2)$  neighborhood in  $G_{sparse}$  and these edges are contained in  $G_{sparse}(u)$ . The analysis of Lemma 10 appears in the full version of the paper.

We next describe Alg. SpannerSparseRegion that computes  $H_{sparse}$ . Every vertex  $u$  computes  $G_{sparse}(u)$  in  $O(\log k)$  rounds and simulate Alg. LocalSpanner in that subgraph. Let  $H_{sparse}(u)$  be the edges added to the spanner in the local simulation of Alg. LocalSpanner in  $G_{sparse}(u)$ . A sparse vertex  $u$  sends to each sparse vertex  $v \in \Gamma_{k/2-1}(u, G_{sparse})$ , the set of all  $v$ -edges in  $H_{sparse}(u)$ . Hence, each sparse vertex sends  $O(n)$  messages (at most  $\sqrt{n}$ -edges to each of its at most  $\sqrt{n}$  vertices in  $\Gamma_{k/2-1}(v, G_{sparse})$ ). In a symmetric manner, every vertex receives  $O(n)$  messages and this step can be done in  $O(1)$  rounds using Lenzen's algorithm. The final spanner is given by  $H_{sparse} = \bigcup_{u \in V_{sparse}} H_{sparse}(u)$ . The stretch argument is immediate by the correctness of Alg. LocalSpanner and the fact that all the edges added to the spanner in the local simulations are indeed added to  $H_{sparse}$ . The size argument is also immediate since we only add edges that Alg. LocalSpanner would have added when running by the entire graph.

Algorithm SpannerSparseRegion (Code for a sparse vertex  $u$ )

1. Apply Alg. NearestNeighbors to compute  $G_{sparse}(u)$  for each sparse vertex  $u$ .
2. Locally simulate Alg. LocalSpanner in  $G_{sparse}(u)$  and let  $H_{sparse}(u)$  be the edges added to the spanner in  $G_{sparse}(u)$ .
3. Send the edges of  $H_{sparse}(u)$  to the corresponding sparse endpoints.
4. Add the received edges to the spanner  $H_{sparse}$ .

#### 4 Handling the Dense Subgraph

In this section, we construct the spanner  $H_{dense}$  satisfying that  $\text{dist}(u, v, H_{dense}) \leq 2k - 1$  for every  $(u, v) \in E_{dense}$ . In this case, since the  $(k/2 - 1)$  neighborhood of each dense vertex is large then there exists a small hitting that covers all these neighborhoods. The structure of our arguments is as follows. First, we describe a deterministic construction of  $H_{dense}$  using an hitting-set algorithm as a black box. This would immediately imply a randomized spanner construction in  $O(\log k)$ -rounds. Then in Section 5, we fill in this last missing piece and show deterministic constructions of hitting sets.

**Constructing spanner for the dense subgraph via hitting sets.** Our goal is to cluster all dense vertices into small number of low-depth clusters. This translates into the following *hitting-set* problem defined in [5, 28, 13]: Given a subset  $V' \subseteq V$  and a set collection  $\mathcal{S} = \{S(v) \mid v \in V'\}$  where each  $|S(v)| \geq \Delta$  and  $\bigcup_{v \in V'} S(v) \subseteq V''$ , compute a subset  $Z \subseteq V''$  of cardinality  $O(|V''| \log n / \Delta)$  that intersects (i.e., *hits*) each subset  $S \in \mathcal{S}$ . A hitting-set of size  $O(|V''| \log n / \Delta)$  is denoted as a *small* hitting-set.

We prove the next lemma by describing the construction of the spanner  $H_{dense}$  given an algorithm  $\mathcal{A}$  that computes small hitting sets. In Section 5, we complement this lemma by describing several constructions of hitting sets. Let  $G = (V, E)$  be an  $n$ -vertex graph, and let  $\Delta \in [n]$  be a parameter. Let  $V'$  be a subset of nodes such that each node  $u \in V'$  knows a set  $S_u$  where  $|S_u| \geq \Delta$ . Let  $\mathcal{S} = \{S_u \subset V : u \in V'\}$  and suppose that  $V''$  is such that  $\bigcup S_u \subseteq V''$ .

► **Lemma 11.** *Given an algorithm  $\mathcal{A}$  for computing a small hitting-set in  $r_{\mathcal{A}}$  rounds, there exists a deterministic algorithm SpannerDenseRegion for constructing the  $(2k - 1)$  spanner  $H_{dense}$  within  $O(\log k + r_{\mathcal{A}})$  rounds.*

The next definition is useful in our context.

**$\ell$ -depth Clustering.** A *cluster* is a subset of vertices and a clustering  $\mathcal{C} = \{C_1, \dots, C_\ell\}$  consists of vertex disjoint subsets. For a positive integer  $\ell$ , a clustering  $\mathcal{C}$  is a  $\ell$ -depth clustering if for each cluster  $C \in \mathcal{C}$ , the graph  $G$  contains a tree of depth at most  $\ell$  rooted at the cluster center of  $C$  and spanning all its vertices.

## 4.1 Description of Algorithm SpannerDenseRegion

The algorithm is based on clustering the dense vertices in two levels of clustering, in a Baswana-Sen like manner. The first clustering  $\mathcal{C}_1$  is an  $(k/2 - 1)$ -depth clustering covering all the dense vertices. The second clustering,  $\mathcal{C}_2$  is an  $(k/2)$ -depth clustering that covers only a *subset* of the dense vertices. For  $k$  odd, let  $\mathcal{C}_2$  be equal to  $\mathcal{C}_1$ .

**Defining the first level of clustering.** Recall that by running Algorithm NearestNeighbors, every non-heavy vertex  $v \in G_{light}$  knows the set  $N_{k/2-1}(v)$  containing its  $n^{1/2-1/k}$  nearest neighbors in  $\Gamma_{k/2-1}(v, G_{light})$ . For every heavy vertex  $v$ , let  $N_{k/2-1}(v) = \Gamma(v, G)$ . Let  $V_{nh}$  be the set of all non-heavy vertices that are *neighbors* of heavy vertices. By definition,  $V_{nh} \subseteq V_{dense}$ . Note that for every dense vertex  $v \in V_{dense} \setminus V_{nh}$ , it holds that  $|N_{k/2-1}(v)| \geq n^{1/2-1/k}$ . The vertices  $u$  of  $V_{nh}$  are in  $G_{light}$  and hence have computed the set  $N_{k/2-1}(u)$ , however, there is in guarantee on the size of these sets.

To define the clustering of the dense vertices, Algorithm SpannerDenseRegion applies the hitting-set algorithm  $\mathcal{A}$  on the subsets  $\mathcal{S}_1 = \{N_{k/2-1}(v) \mid v \in V_{dense} \setminus V_{nh}\}$  and the universe  $V$ . Since every set in  $\mathcal{S}_1$  has size at least  $\Delta := n^{1/2-1/k}$ , the output of algorithm  $\mathcal{A}$  is a subset  $Z_1$  of cardinality  $O(n^{1/2+1/k})$  that hits all the sets in  $\mathcal{S}_1$ .

We will now construct the clusters in  $\mathcal{C}_1$  with  $Z_1$  as the cluster centers. To make sure that the clusters are vertex-disjoint and connected, we first compute the clustering in the subgraph  $G_{light}$ , and then cluster the remaining dense vertices that are not yet clustered. For every  $v \in G_{light}$  (either dense or sparse), we say that  $v$  is *clustered* if  $Z_1 \cap N_{k/2-1}(v) \neq \emptyset$ . In particular, every dense vertex  $v$  for which  $|\Gamma_{k/2-1}(v, G_{light})| \geq n^{1/2-1/k}$  is clustered (the neighbors of heavy vertices are either clustered or not). For every clustered vertex  $v \in G_{light}$  (i.e., even sparse ones), let  $c_1(v)$ , denoted hereafter the *cluster center* of  $v$ , be the closest vertex to  $v$  in  $Z_1 \cap N_{k/2-1}(v)$ , breaking shortest-path ties based on IDs. Since  $v$  knows the entire tree  $T_{k/2-1}(v)$ , it knows the distance to all the vertices in  $N_{k/2-1}(v)$  and in addition, it can compute its next-hop  $p(v)$  on the  $v$ - $c_1(v)$  shortest path in  $G_{light}$ . Each clustered vertex  $v \in G_{light}$ , adds the edge  $(v, p(v))$  to the spanner  $H_{dense}$ . It is easy to see that this defines a  $(k/2 - 1)$ -depth clustering in  $G_{light}$  that covers all dense vertices in  $G_{light}$ . In particular, each cluster  $C$  has in the spanner a tree of depth at most  $(k/2 - 1)$  that spans all the vertices in  $C$ . Note that in order for the clusters  $C$  to be connected in  $H_{dense}$ , it was crucial that all vertices in  $G_{light}$  compute their cluster centers in  $N_{k/2-1}(v)$ , if such exists, and not only the dense vertices. We next turn to cluster the remaining dense vertices. For every heavy vertex  $v$ , let  $c_1(v)$  be its closest vertex in  $\Gamma(v, G) \cap Z_1$ . It then adds the edge  $(v, c_1(v))$  to the spanner  $H_{dense}$  and broadcasts its cluster center  $c_1(v)$  to all its neighbors. Every neighbor  $u$  of a heavy vertex  $v$  that is not yet clustered, joins the cluster of  $c_1(v)$  and adds the edge  $(u, v)$  to the spanner. Overall, the clusters of  $\mathcal{C}_1$  centered at the subset  $Z_1$  cover all the dense vertices. In addition, all the vertices in a cluster  $C$  are connected in  $H_{dense}$  by a tree of depth  $k/2 - 1$ . Formally,  $\mathcal{C}_1 = \{C_1(s), \mid s \in Z_1\}$  where  $C_1(s) = \{v \mid c_1(v) = s\}$ .

**Defining the second level of clustering.** Every vertex  $v$  that is clustered in  $\mathcal{C}_1$  broadcasts its cluster center  $c_1(v)$  to all its neighbors. This allows every dense vertex  $v$  to compute the subset  $N_{k/2}(v) = \{s \in Z_1 \mid E(v, C_1(s), G) \neq \emptyset\}$  consisting of the centers of its adjacent clusters in  $\mathcal{C}_1$ . Consider two cases depending on the cardinality of  $N_{k/2}(v)$ . Every vertex  $v$  with  $|N_{k/2}(v)| \leq n^{1/k} \log n$ , adds to the spanner  $H_{dense}$  an arbitrary edge in  $E(v, C_1(s), G)$  for every  $s \in N_{k/2}(v)$ . It remains to handle the remaining vertices  $V'_{dense} = \{v \in V_{dense} \mid |N_{k/2}(v)| > n^{1/k} \log n\}$ . These vertices would be clustered in the second level of clustering  $\mathcal{C}_2$ . To compute the centers of the clusters in  $\mathcal{C}_2$ , the algorithm applies the hitting-set algorithm  $\mathcal{A}$  on the collection of subsets  $\mathcal{S}_2 = \{N_{k/2}(v) \mid v \in V'_{dense}\}$  with  $\Delta = n^{1/k} \log n$  and  $V'' = Z_1$ . The output of  $\mathcal{A}$  is a subset  $Z_2$  of cardinality  $O(|Z_1| \log n / \Delta) = O(\sqrt{n} \log n)$  that hits all the subsets in  $\mathcal{S}_2$ . The  $2^{nd}$  cluster-center  $c_2(v)$  of a vertex  $v \in V'_{dense}$  is chosen to be an arbitrary  $s \in N_{k/2}(v) \cap Z_2$ . The vertex  $v$  then adds some edge  $(v, u) \in E(v, C_1(s), G)$  to the spanner  $H_{dense}$ . Hence, the trees spanning rooted at  $s \in Z_2$  are now extended by one additional layer resulting in a  $(k/2)$ -depth clustering.

**Connecting adjacent clusters.** Finally, the algorithm adds to the spanner  $H_{dense}$  a single edge between each pairs of adjacent clusters  $C, C' \in \mathcal{C}_1 \times \mathcal{C}_2$ , this can be done in  $O(1)$  rounds as follows. Each vertex broadcasts its cluster ID in  $\mathcal{C}_2$ . Every vertex  $v \in C$  for every cluster  $C' \in \mathcal{C}_1$  picks one incident edge to each cluster  $C' \in \mathcal{C}_2$  (if such exists) and sends this edge to the corresponding center of the cluster of  $C'$  in  $\mathcal{C}_2$ . Since a vertex sends at most one message for each cluster center in  $\mathcal{C}_2$ , this can be done in  $O(1)$  rounds. Each cluster center  $r$  of the cluster  $C'$  in  $\mathcal{C}_2$  picks one representative edge among the edges it has received for each cluster  $C \in \mathcal{C}_1$  and sends a notification about the selected edge to the endpoint of the edge in  $C$ . Since the cluster center sends at most one edge for every vertex this take one round. Finally, the vertices in the clusters  $C \in \mathcal{C}_1$  add the notified edges (that they received from the centers of  $\mathcal{C}_2$ ) to the spanner. This completes the description of the algorithm. We now complete the proof of Lemma 11.

**Proof.** Recall that we assume  $k \geq 6$  and thus  $|\Gamma_{k/2-1}(v)| \geq n^{1/2-1/k}$ , for every  $v \in V_{dense}$ . We first show that for every  $(u, v) \in E_{dense}$ ,  $\text{dist}(u, v, H_{dense}) \leq 2k - 1$ . The clustering  $\mathcal{C}_1$  covers all the dense vertices. If  $u$  and  $v$  belong to the same cluster  $C$  in  $\mathcal{C}_1$ , the claim follows as  $H_{dense}$  contains an  $(k/2 - 1)$ -depth tree that spans all the vertices in  $C$ , thus  $\text{dist}(u, v, H_{dense}) \leq k - 2$ . From now on assume that  $c_1(u) \neq c_1(v)$ . We first consider the case that for both of the endpoints it holds that  $|N_{k/2}(v)|, |N_{k/2}(u)| \leq n^{1/k} \log n$ . In such a case, since  $v$  is adjacent to the cluster  $C_1$  of  $u$ , the algorithm adds to  $H_{dense}$  at least one edge in  $E(v, C_1, G)$ , let it be  $(x, v)$ . We have that  $\text{dist}(v, u, H_{dense}) \leq \text{dist}(v, x, H_{dense}) + \text{dist}(x, u, H_{dense}) \leq k - 1$  where the last inequality holds as  $x$  and  $u$  belong to the same cluster  $C_1$  in  $\mathcal{C}_1$ . Finally, it remains to consider the case where for at least one endpoint, say  $v$ , it holds that  $|N_{k/2}(v)| > n^{1/k} \log n$ . In such a case,  $v$  is clustered in  $\mathcal{C}_2$ . Let  $C_1$  be the cluster of  $u$  in  $\mathcal{C}_1$  and let  $C_2$  be the cluster of  $v$  in  $\mathcal{C}_2$ . Since  $C_1$  and  $C_2$  are adjacent, the algorithm adds an edge in  $E(C_1, C_2, G)$ , let it be  $(x, y)$  where  $x, u \in C_1$  and  $y, v \in C_2$ . We have that  $\text{dist}(u, v, H_{dense}) \leq \text{dist}(u, x, H_{dense}) + \text{dist}(x, y, H_{dense}) + \text{dist}(y, v, H_{dense}) \leq 2k - 1$ , where the last inequality holds as  $u, x$  belong to the same  $(k/2 - 1)$ -depth cluster  $C_1$ , and  $v, y$  belong to the same  $(k/2)$ -depth cluster  $C_2$ . Finally, we bound the size of  $H_{dense}$ . Since the clusters in  $\mathcal{C}_1, \mathcal{C}_2$  are vertex-disjoint, the trees spanning these clusters contain  $O(n)$  edges. For each unclustered vertex in  $\mathcal{C}_2$ , we add  $O(n^{1/k} \log n)$  edges. By the properties of the hitting-set algorithm  $\mathcal{A}$  it holds that  $|Z_1| = O(n^{1/2-1/k} \cdot \log n)$  and  $|Z_2| = O(n^{1/2} \cdot \log n)$ . Thus adding one edge between each pair of clusters adds  $|Z_1| \cdot |Z_2| = O(n^{1+1/k} \cdot \log^2 n)$  edges.  $\blacktriangleleft$



**Putting All Together: Randomized spanners in  $O(\log k)$  rounds.** We now complete the proof of Theorem 1. For an edge  $(u, v) \in E_{sparse}$ , the correctness follows by the correctness of Alg. LocalSpanner. We next consider the dense case. Let  $\mathcal{A}$  be the algorithm where each  $v \in V'$  is added into  $Z$  with probability of  $\log/\Delta$ . By Chernoff bound, we get that w.h.p.  $|Z| = O(|V'| \log n/\Delta)$  and  $Z \cap S_i \neq \emptyset$  for every  $S_i \in \mathcal{S}$ . The correctness follows by applying Lemma 11.  $\blacktriangleleft$

Algorithm SpannerDenseRegion

1. Compute an  $(k/2 - 1)$  clustering  $\mathcal{C}_1 = \{C(s) \mid s \in Z_1\}$  centered at subset  $Z_1$ .
2. For every  $v \in V_{dense}$ , let  $N_{k/2}(v) = \{s \in Z_1 \mid E(v, C_1(s), G) \neq \emptyset\}$ .
3. For every  $v \in V_{dense}$  with  $|N_{k/2}(v)| \leq n^{1/k} \log n$ , add to the spanner one edge in  $E(v, C(s), G)$  for every  $s \in N_{k/2}(v)$ .
4. Compute an  $(k/2)$  clustering  $\mathcal{C}_2$  centered at  $Z_2$  to cover the remaining dense vertices.
5. Connect (in the spanner) each pair of adjacent clusters  $C, C' \in \mathcal{C}_1 \times \mathcal{C}_2$ .

## 5 Derandomization of Hitting Sets

### 5.1 Hitting Sets with Short Seeds

The main technical part of the deterministic construction is to completely derandomize the randomized hitting-set algorithm using short seeds. We show two hitting-set constructions with different tradeoffs. The first construction is based on pseudorandom generators (PRG) for DNF formulas. The PRG will have a seed of length  $O(\log n (\log \log n)^3)$ . This would serve the basis for the construction of Theorem 2. The second hitting-set construction is based on  $O(1)$ -wise independence, it uses a small seed of length  $O(\log n)$  but yields a larger hitting-set. This would be the basis for the construction of Theorem 3.

We begin by setting up some notation. For a set  $S$  we denote by  $x \sim S$  a uniform sampling from  $S$ . For a function PRG and an index  $i$ , let  $\text{PRG}(s)_i$  the  $i^{\text{th}}$  bit of  $\text{PRG}(s)$ .

► **Definition 12** (Pseudorandom Generators). A generator  $\text{PRG}: \{0, 1\}^r \rightarrow \{0, 1\}^n$  is an  $\epsilon$ -pseudorandom generator (PRG) for a class  $\mathcal{C}$  of Boolean functions if for every  $f \in \mathcal{C}$ :

$$\left| \mathbf{E}_{x \sim \{0, 1\}^n} [f(x)] - \mathbf{E}_{s \sim \{0, 1\}^r} [f(\text{PRG}(s))] \right| \leq \epsilon.$$

We refer to  $r$  as the seed-length of the generator and say PRG is explicit if there is an efficient algorithm to compute PRG that runs in time  $\text{poly}(n, 1/\epsilon)$ .

► **Theorem 13.** For every  $\epsilon = \epsilon(n) > 0$ , there exists an explicit pseudorandom generator,  $\text{PRG}: \{0, 1\}^r \rightarrow \{0, 1\}^n$  that fools all read-once DNFs on  $n$ -variables with error at most  $\epsilon$  and seed-length  $r = O((\log(n/\epsilon)) \cdot (\log \log(n/\epsilon))^3)$ .

Using the notation above, and Theorem 13 we formulate and prove the following Lemma:

► **Lemma 14.** Let  $S$  be subset of  $[n]$  where  $|S| \geq \Delta$  for some parameter  $\Delta \leq n$  and let  $c$  be any constant. Then, there exists a family of hash functions  $\mathcal{H} = \{h: [n] \rightarrow \{0, 1\}\}$  such that choosing a random function from  $\mathcal{H}$  takes  $r = O(\log n \cdot (\log \log n)^3)$  random bits and for  $Z_h = \{u \in [n] : h(u) = 0\}$  it holds that:

$$(1) \Pr_h \left[ |Z_h| \leq \tilde{O}(n/\Delta) \right] \geq 2/3, \text{ and } (2) \Pr_h [S \cap Z_h \neq \emptyset] \geq 1 - 1/n^c.$$

**Proof.** We first describe the construction of  $\mathcal{H}$ . Let  $p = c' \log n/\Delta$  for some large constant  $c'$  (will be set later), and let  $\ell = \lceil \log 1/p \rceil$ . Let  $\text{PRG}: \{0, 1\}^r \rightarrow \{0, 1\}^{n^\ell}$  be the PRG constructed in Theorem 13 for  $r = O(\log n \ell \cdot (\log \log n \ell)^3) = O(\log n \cdot (\log \log n)^3)$  and

## 40:14 Congested Clique Algorithms for Graph Spanners

for  $\epsilon = 1/n^{10c}$ . For a string  $s$  of length  $r$  we define the hash function  $h_s(i)$  as follows. First, it computes  $y = \text{PRG}(s)$ . Then, it interprets  $y$  as  $n$  blocks where each block is of length  $\ell$  bits, and outputs 1 if and only if all the bits of the  $i^{\text{th}}$  block are 1. Formally, we define  $h_s(i) = \bigwedge_{j=(i-1)\ell+1}^{i\ell} \text{PRG}(s)_j$ . We show that properties 1 and 2 hold for the set  $Z_{h_s}$  where  $h_s \in \mathcal{H}$ . We begin with property 1. For  $i \in [n]$  let  $X_i = h_s(i)$  be a random variable where  $s \sim \{0,1\}^r$ . Moreover, let  $X = \sum_{i=1}^n X_i$ . Using this notation we have that  $|Z_{h_s}| = X$ . Thus, to show property 1, we need to show that  $\Pr_{s \sim \{0,1\}^r} [X \leq \tilde{O}(n/\Delta)] \geq 2/3$ . Let  $f_i: \{0,1\}^{n\ell} \rightarrow \{0,1\}$  be a function that outputs 1 if the  $i^{\text{th}}$  block is all 1's. That is,  $f_i(y) = \bigwedge_{j=(i-1)\ell+1}^{i\ell} y_j$ . Since  $f_i$  is a read-once DNF formula we have that

$$\left| \mathbf{E}_{y \sim \{0,1\}^{n\ell}} [f_i(y)] - \mathbf{E}_{s \sim \{0,1\}^r} [f_i(\text{PRG}(s))] \right| \leq \epsilon.$$

Therefore, it follows that

$$\begin{aligned} \mathbf{E}[X] &= \sum_{i=1}^n \mathbf{E}[X_i] = \sum_{i=1}^n \mathbf{E}_{s \sim \{0,1\}^r} [f_i(\text{PRG}(s))] \leq \\ &\sum_{i=1}^n (\mathbf{E}_{y \sim \{0,1\}^{n\ell}} [f_i(y)] + \epsilon) = n(2^{-\ell} + \epsilon) = \tilde{O}\left(\frac{n}{\Delta}\right). \end{aligned}$$

Then, by Markov's inequality we get that  $\Pr_{s \sim \{0,1\}^r} [X > 3\mathbf{E}[X]] \leq 1/3$  and thus

$$\Pr_{s \sim \{0,1\}^r} [X \leq \tilde{O}(n/\Delta)] \geq 1 - \Pr_{s \sim \{0,1\}^r} [X > 3\mathbf{E}[X]] \geq 2/3.$$

We turn to show property 2. Let  $S$  be any set of size at least  $\Delta$  and let  $g: \{0,1\}^{n\ell} \rightarrow \{0,1\}$  be an indicator function for the event that the set  $S$  is covered. That is,

$$g(y) = \bigvee_{i \in S} \bigwedge_{j=(i-1)\ell+1}^{i\ell} y_j.$$

Since  $g$  is a read-once DNF formula, and thus we have that

$$\left| \mathbf{E}_{y \sim \{0,1\}^{n\ell}} [g(y)] - \mathbf{E}_{s \sim \{0,1\}^r} [g(\text{PRG}(s))] \right| \leq \epsilon.$$

Let  $Y_i = \bigwedge_{j=(i-1)\ell+1}^{i\ell} y_j$ , and let  $Y = \sum_{i \in S} Y_i$ . Then  $\mathbf{E}[Y] = \sum_{i \in S} \mathbf{E}[Y_i] \geq \Delta 2^{-\ell} \geq \Delta p = c' \log n$ . Thus, by a Chernoff bound we have that  $\Pr[Y = 0] \leq \Pr[\mathbf{E}[Y] - Y \geq c' \log n] \leq 1/n^{2c}$ , for a large enough constant  $c'$  (that depends on  $c$ ). Together, we get that  $\Pr_s[S \cap Z_{h_s} \neq \emptyset] = \Pr_{s \sim \{0,1\}^r} [g(\text{PRG}(s))] \geq \mathbf{E}_{y \sim \{0,1\}^{n\ell}} [g(y)] - \epsilon = \Pr_{y \sim \{0,1\}^{n\ell}} [Y \geq 1] - \epsilon \geq 1 - 1/n^c$ .  $\blacktriangleleft$

We turn to show the second construction of dominating sets with short seed. In this construction the seed length is shorter, but the set is larger. By a direct application of Lemma 2.2 in [6], we get the following lemma which becomes useful for showing Theorem 3.

**► Lemma 15.** *Let  $S$  be a subset of  $[n]$  where  $|S| \geq \Delta$  for some parameter  $\Delta \leq n$  and let  $c$  be any constant. Then, there exists a family of hash functions  $\mathcal{H} = \{h: [n] \rightarrow \{0,1\}\}$  such that choosing a random function from  $\mathcal{H}$  takes  $r = O(\log n)$  random bits and for  $Z_h = \{u \in [n] : h(u) = 0\}$  it holds that: (1)  $\Pr_h[|Z_h| \leq O(n^{17/16}/\sqrt{\Delta})] \geq 2/3$ , and (2)  $\Pr_h[S \cap Z_h \neq \emptyset] \geq 1 - 1/n^c$ .*

## 5.2 Deterministic Hitting Sets in the Congested Clique

We next present a deterministic construction of hitting sets by means of derandomization. The round complexity of the algorithm depends on the number of random bits used by the randomized algorithms.

► **Theorem 16.** *Let  $G = (V, E)$  be an  $n$ -vertex graph, let  $V' \subset V$ , let  $\mathcal{S} = \{S_u \subset V : u \in V'\}$  be a set of subsets such that each node  $u \in V'$  knows the set  $S_u$  and  $|S_u| \geq \Delta$ , and let  $c$  be a constant. Let  $\mathcal{H} = \{h: [n] \rightarrow \{0, 1\}\}$  be a family of hash functions such that choosing a random function from  $\mathcal{H}$  takes  $g_{\mathcal{A}}(n, \Delta)$  random bits and for  $Z_h = \{u \in [n] : h(u) = 0\}$  it holds that: (1)  $\Pr[|Z_h| \leq f_{\mathcal{A}}(n, \Delta)] \geq 2/3$  and (2) for any  $u \in V'$ :  $\Pr[S_u \cap Z_h \neq \emptyset] \geq 1 - 1/n^c$ . Then, there exists a deterministic algorithm  $\mathcal{A}_{det}$  that constructs a hitting set of size  $O(f_{\mathcal{A}}(n, \Delta))$  in  $O(g_{\mathcal{A}}(n, \Delta)/\log n)$  rounds.*

**Proof.** Our goal is to completely derandomize the process of finding  $Z_h$  by using the method of conditional expectation. We follow the scheme of [7] to achieve this, and define two bad events that can occur when using a random seed of size  $g = g_{\mathcal{A}}(n, \Delta)$ . Let  $A$  be the event where the hitting set  $Z_h$  consists of more than  $f_{\mathcal{A}}(n, \Delta)$  vertices. Let  $B$  be the event that there exists an  $u \in V'$  such that  $S_u \cap Z_h = \emptyset$ . Let  $X_A, X_B$  be the corresponding indicator random variables for the events, and let  $X = X_A + X_B$ .

Since a random seed with  $g_{\mathcal{A}}(n, \Delta)$  bits avoids both of these events with high probability, we have that  $\mathbf{E}[X] < 1$  where the expectation is taken over a seed of length  $g$  bits. Thus, we can use the method of conditional expectations in order to get an assignment to our random coins such that no bad event occurs, i.e.,  $X = 0$ . In each step of the method, we run a distributed protocol to compute the conditional expectation. Actually, we will compute a pessimistic estimator for the conditional expectation.

Letting  $X_u$  be indicator random variable for the event that  $S_u$  is not hit by  $Z_h$ , we can write our expectation as follows:  $\mathbf{E}[X] = \mathbf{E}[X_A] + \mathbf{E}[X_B] = \Pr[X_A = 1] + \Pr[X_B = 1] = \Pr[X_A = 1] + \Pr[\bigvee_u X_u = 1]$ . Suppose we have a partial assignment to the seed, denoted by  $Y$ . Our goal is to compute the conditional expectation  $\mathbf{E}[X|Y]$ , which translates to computing  $\Pr[X_A = 1|Y]$  and  $\Pr[\bigvee_u X_u = 1|Y]$ . Notice that computing  $\Pr[X_A = 1|Y]$  is simple since it depends only on  $Y$  (and not on the graph or the subsets  $\mathcal{S}$ ). The difficult part is computing  $\Pr[\bigvee_u X_u = 1|Y]$ . Instead, we use a pessimistic estimator of  $\mathbf{E}[X]$  which avoids this difficult computation. Specifically, we define the estimator:  $\Psi = X_A + \sum_{u \in V'} X_u$ . Recall that for any  $u \in V'$  for a random  $g$ -bit length seed, it holds that  $\Pr[X_u = 1] \leq 1/n^c$  and thus by applying a union bound over all  $n$  sets, it also holds that  $\mathbf{E}[\Psi] = \Pr[X_A = 1] + \sum_u \Pr[X_u = 1] < 1$ . We describe how to compute the desired seed using the method of conditional expectation. We will reveal the assignment of the seed in chunks of  $\ell = \lceil \log n \rceil$  bits. In particular, we show how to compute the assignment of  $\ell$  bits in the seed in  $O(1)$  rounds. Since the seed has  $g$  many bits, this will yield an  $O(g/\log n)$  round algorithm.

Consider the  $i^{\text{th}}$  chunk of the seed  $Y_i = (y_1, \dots, y_{\ell})$  and assume that the assignment for the first  $i-1$  chunks  $Y_1, \dots, Y_{i-1}$  have been computed. For each of the  $n$  possible assignments to  $Y_i$ , we assign a node  $v$  that receives the conditional probability values  $\Pr[X_u = 1|Y_1, \dots, Y_i]$  from all nodes  $u \in V'$ . Notice that a node  $u$  can compute the conditional probability values  $\Pr[X_u = 1|Y_1, \dots, Y_i]$ , since  $u$  knows the IDs of the vertices in  $S_u$  and thus has all the information for this computation. The node  $v$  then sums up all these values and sends them to a global leader  $w$ . The leader  $w$  can easily compute the conditional probability  $\Pr[X_A = 1|Y]$ , and thus using the values it received from all the nodes it can compute  $\mathbf{E}[X|Y]$  for of the possible  $n$  assignments to  $Y_i$ . Finally,  $w$  selects the assignment  $(y_1^*, \dots, y_{\ell}^*)$  that minimizes the pessimistic estimator  $\Psi$  and broadcasts it to all nodes in the graph. After  $O(g/\log n)$  rounds  $Y$  has been completely fixed such that  $X < 1$ . Since  $X_A$  and  $X_B$  get binary values, it must be the case that  $X_A = X_B = 0$ , and a hitting set has been found. ◀

Combining Lemma 14 and Lemma 15 with Theorem 16, yields:

► **Corollary 17.** *Let  $G = (V, E)$  be an  $n$ -vertex graph, let  $V', V'' \subseteq V$ , let  $\mathcal{S} = \{S_u \subseteq V : u \in V'\}$  be a set of subsets such that each node  $u \in V'$  knows the set  $S_u$ , such that  $|S_u| \geq \Delta$  and  $\bigcup S_u \subseteq V''$ . Then, there exists deterministic algorithms  $\mathcal{A}_{det}, \mathcal{A}'_{det}$  in the congested clique model that construct a hitting set  $Z$  for  $\mathcal{S}$  such that: (1)  $|Z| = \tilde{O}(|V''|/\Delta)$  and  $\mathcal{A}_{det}$  runs in  $O((\log \log n)^3)$  rounds. (2)  $|Z| = O(|V''|^{17/16}/\sqrt{\Delta})$  and  $\mathcal{A}'_{det}$  runs in  $O(1)$  rounds.*

**Deterministic construction in  $O(\log k + O((\log \log n)^3))$  Rounds.** Theorem 2 follows by plugging Corollary 17(1) into Lemma 11.

### 5.3 Deterministic $O(k)$ -Spanners in $O(\log k)$ Rounds

In this subsection, we provide a proof sketch of Theorem 3. The complete proof appears in the full version. Let  $k \geq 10$ . According to Section 3, it remains to consider the construction of  $H_{dense}$  for the dense edge set  $E_{dense}$ . Recall that for every dense vertex  $v$ , it holds that  $|\Gamma_{k/2}(v, G)| \geq n^{1/2-1/k}$ . Similarly to the proof of Lemma 11, we construct a  $(k/2 - 1)$  dominating set  $Z$  for the dense vertices. However, to achieve the desired round complexity, we use the  $O(1)$ -round hitting set construction of Corollary 17(2) with parameters of  $\Delta = n^{1/2-1/k}$  and  $V' = V$ . The output is then a hitting set  $Z$  of cardinality  $O(n^{13/16+1/(2k)})$  that hits all the  $(k/2 - 1)$  neighborhoods of the dense vertices. Then, as in Alg. `SpannerDenseRegion`, we compute a  $(k/2 - 1)$ -depth clustering  $\mathcal{C}_1$  centered at  $Z$ . The key difference to Alg. `SpannerDenseRegion` is that  $|Z|$  is too large for allowing us to add an edge between each pair of adjacent clusters, as this would result in a spanner of size  $O(|Z|^2)$ . Instead, we essentially contract the clusters of  $\mathcal{C}_1$  (i.e., contracting the intra-cluster edges) and construct the spanner recursively in the resulting contracted graph  $G''$ . Every contracted node in  $G''$  corresponds to a cluster with a small strong diameter in the spanner. Specifically,  $G''$  is decomposed into sparse and dense regions. Handling the sparse part is done deterministically by applying Alg. `SpannerSparseRegion`. To handle the dense case, we apply the hitting-set algorithm of Corollary 17(2) to cluster the dense nodes (which are in fact, contracted nodes) into  $|V(G'')|/\sqrt{\Delta}$  clusters for  $\Delta = n^{1/2-1/k}$ . After  $O(1)$  repetitions of the above, we will be left with a contracted graph with  $o(\sqrt{n})$  vertices. At this point, we connect each pair of clusters (corresponding to these contracted nodes) in the spanner.

A naïve implementation of such an approach would yield a spanner with stretch  $k^{O(1)}$ , as the diameter of the clusters induced by the contracted nodes is increased by a  $k$ -factor in each of the phases. To avoid this blow-up in the stretch, we enjoy the fact that already after the first phase, the contracted graph  $G'$  has  $O(n^{13/16+o(1)})$  nodes and hence we can allow to compute a  $(2k' - 1)$  spanner for  $G'$  with  $k' = 8$  as this would add  $O(n)$  edges to the final spanner. Since in each of the phases (except for the first one) the stretch parameter is *constant*, the stretch will be bounded by  $O(k)$ , and the number of edges by  $O(k \cdot n^{1+1/k})$ .

---

#### References

- 1 Leonid Barenboim and Victor Khazanov. Distributed symmetry-breaking algorithms for congested cliques. *arXiv preprint arXiv:1802.07209*, 2018.
- 2 Surender Baswana and Sandeep Sen. A simple and linear time randomized algorithm for computing sparse spanners in weighted graphs. *Random Structures and Algorithms*, 30(4):532–563, 2007.

- 3 Ruben Becker, Andreas Karrenbauer, Sebastian Krinninger, and Christoph Lenzen. Near-optimal approximate shortest paths and transshipment in distributed and streaming models. In *DISC*, 2017.
- 4 Soheil Behnezhad, Mahsa Derakhshan, and MohammadTaghi Hajiaghayi. Brief announcement: Semi-mapreduce meets congested clique. *arXiv preprint arXiv:1802.10297*, 2018.
- 5 Arnab Bhattacharyya, Elena Grigorescu, Kyomin Jung, Sofya Raskhodnikova, and David P Woodruff. Transitive-closure spanners. *SIAM Journal on Computing*, 41(6):1380–1425, 2012.
- 6 L Elisa Celis, Omer Reingold, Gil Segev, and Udi Wieder. Balls and bins: Smaller hash families and faster evaluation. *SIAM Journal on Computing*, 42(3):1030–1050, 2013.
- 7 Keren Censor-Hillel, Merav Parter, and Gregory Schwartzman. Derandomizing local distributed algorithms under bandwidth restrictions. In *31 International Symposium on Distributed Computing*, 2017.
- 8 Bilel Derbel and Cyril Gavoille. Fast deterministic distributed algorithms for sparse spanners. *Theoretical Computer Science*, 2008.
- 9 Bilel Derbel, Cyril Gavoille, and David Peleg. Deterministic distributed construction of linear stretch spanners in polylogarithmic time. In *DISC*, pages 179–192. Springer, 2007.
- 10 Bilel Derbel, Cyril Gavoille, David Peleg, and Laurent Viennot. On the locality of distributed sparse spanner construction. In *PODC*, pages 273–282, 2008.
- 11 Bilel Derbel, Cyril Gavoille, David Peleg, and Laurent Viennot. Local computation of nearly additive spanners. In *DISC*, 2009.
- 12 Bilel Derbel, Mohamed Mosbah, and Akka Zemmari. Sublinear fully distributed partition with applications. *Theory of Computing Systems*, 47(2):368–404, 2010.
- 13 Mohsen Ghaffari. An improved distributed algorithm for maximal independent set. *Manuscript*, 2018.
- 14 Mohsen Ghaffari, Themis Gouleakis, Slobodan Mitrović, and Ronitt Rubinfeld. Improved massively parallel computation algorithms for mis, matching, and vertex cover. *PODC*, 2018.
- 15 Parikshit Gopalan, Raghu Meka, Omer Reingold, Luca Trevisan, and Salil P. Vadhan. Better pseudorandom generators from milder pseudorandom restrictions. In *53rd Annual IEEE Symposium on Foundations of Computer Science, FOCS 2012, New Brunswick, NJ, USA, October 20-23, 2012*, pages 120–129, 2012.
- 16 Ofer Grossman and Merav Parter. Improved deterministic distributed construction of spanners. In *DISC*, 2017.
- 17 James W Hegeman and Sriram V Pemmaraju. Lessons from the congested clique applied to mapreduce. *Theoretical Computer Science*, 608:268–281, 2015.
- 18 Christoph Lenzen. Optimal deterministic routing and sorting on the congested clique. In *the Proc. of the Int'l Symp. on Princ. of Dist. Comp. (PODC)*, pages 42–50, 2013.
- 19 Christoph Lenzen and Roger Wattenhofer. Brief announcement: exponential speed-up of local algorithms using non-local communication. In *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing, PODC 2010, Zurich, Switzerland, July 25-28, 2010*, pages 295–296, 2010.
- 20 Zvi Lotker, Elan Pavlov, Boaz Patt-Shamir, and David Peleg. MST construction in  $O(\log \log n)$  communication rounds. In *the Proceedings of the Symposium on Parallel Algorithms and Architectures*, pages 94–100. ACM, 2003.
- 21 Merav Parter and Eylon Yogev. Congested clique algorithms for graph spanners. *arXiv preprint*, 2018. [arXiv:1805.05404](https://arxiv.org/abs/1805.05404).
- 22 David Peleg. *Distributed Computing: A Locality-sensitive Approach*. SIAM, 2000.
- 23 David Peleg and Alejandro A Schäffer. Graph spanners. *Journal of graph theory*, 13(1):99–116, 1989.

## 40:18 Congested Clique Algorithms for Graph Spanners

- 24 David Peleg and Jeffrey D Ullman. An optimal synchronizer for the hypercube. *SIAM Journal on computing*, 18(4):740–747, 1989.
- 25 Seth Pettie. Distributed algorithms for ultrasparse spanners and linear size skeletons. *Distributed Computing*, 22(3):147–166, 2010.
- 26 Liam Roditty, Mikkel Thorup, and Uri Zwick. Deterministic constructions of approximate distance oracles and spanners. In *International Colloquium on Automata, Languages, and Programming*, pages 261–272. Springer, 2005.
- 27 Mikkel Thorup and Uri Zwick. Compact routing schemes. In *Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures*, pages 1–10. ACM, 2001.
- 28 Virginia Vassilevska Williams. Graph algorithms – Fall 2016, MIT, lecture notes 5, 2016. URL: <http://theory.stanford.edu/~virgi/cs267/lecture5.pdf>.

# Lattice Agreement in Message Passing Systems

**Xiong Zheng**

University of Texas at Austin, Austin, TX 78712, USA  
zhengxiongty@utexas.edu

**Changyong Hu**

University of Texas at Austin, Austin, TX 78712, USA  
colinhu9@utexas.edu

**Vijay K. Garg**

University of Texas at Austin, Austin, TX 78712, USA  
garg@ece.utexas.edu

---

## Abstract

This paper studies the lattice agreement problem and the generalized lattice agreement problem in distributed message passing systems. In the lattice agreement problem, given input values from a lattice, processes have to non-trivially decide output values that lie on a chain. We consider the lattice agreement problem in both synchronous and asynchronous systems. For synchronous lattice agreement, we present two algorithms which run in  $\log(f)$  and  $\min\{O(\log^2 h(L)), O(\log^2 f)\}$  rounds, respectively, where  $h(L)$  denotes the height of the input sublattice  $L$ ,  $f < n$  is the number of crash failures the system can tolerate, and  $n$  is the number of processes in the system. These algorithms have significant better round complexity than previously known algorithms. The algorithm by Attiya et al. [Attiya et al. DISC, 1995] takes  $\log(n)$  synchronous rounds, and the algorithm by Mavronicolas [Mavronicolas, 2018] takes  $\min\{O(h(L)), O(\sqrt{f})\}$  rounds. For asynchronous lattice agreement, we propose an algorithm which has time complexity of  $2 * \min\{h(L), f + 1\}$  message delays which improves on the previously known time complexity of  $O(n)$  message delays.

The generalized lattice agreement problem defined by Faleiro et al in [Faleiro et al. PODC, 2012] is a generalization of the lattice agreement problem where it is applied for the replicated state machine. We propose an algorithm which guarantees liveness when a majority of the processes are correct in asynchronous systems. Our algorithm requires  $\min\{O(h(L)), O(f)\}$  units of time in the worst case which is better than  $O(n)$  units of time required by the algorithm in [Faleiro et al. PODC, 2012].

**2012 ACM Subject Classification** Theory of computation → Distributed algorithms

**Keywords and phrases** Lattice Agreement, Replicated State Machine, Consensus

**Digital Object Identifier** 10.4230/LIPIcs.DISC.2018.41

**Related Version** A full version of the paper is available at <https://arxiv.org/abs/1807.11557>.

**Funding** Supported by NSF CNS-1812349, NSF CNS-1563544, NSF CNS-1346245, Huawei Inc., and the Cullen Trust for Higher Education Endowed Professorship.

**Acknowledgements** We want to thank John Kaippallimalil for providing some useful application cases for CRDT and generalized lattice agreement.



© Xiong Zheng, Changyong Hu, and Vijay K. Garg;  
licensed under Creative Commons License CC-BY

32nd International Symposium on Distributed Computing (DISC 2018).

Editors: Ulrich Schmid and Josef Widder; Article No. 41; pp. 41:1–41:17

Leibniz International Proceedings in Informatics



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



## 1 Introduction

Lattice agreement, introduced in [2] to solve the atomic snapshot problem [1] in shared memory, is an important decision problem in distributed systems. In this problem, processes start with input values from a lattice and need to decide values which are comparable to each other. Lattice agreement problem is a weaker decision problem than consensus. In synchronous systems, consensus cannot be solved in fewer than  $f + 1$  rounds [6], but lattice agreement can be solved in  $\log f$  rounds (shown by an algorithm we propose). In asynchronous systems, the consensus problem cannot be solved even with one failure [8], whereas the lattice agreement problem can be solved in asynchronous systems when a majority of processes is correct [7].

In synchronous message passing systems, a  $\log n$  rounds recursive algorithm based on “branch-and-bound” approach is proposed in [2] to solve the lattice agreement problem with message complexity of  $O(n^2)$ . It can tolerate at most  $n - 1$  process failures. Later, [12] gave an algorithm with round complexity of  $\min\{1 + h(L), \lfloor (3 + \sqrt{8f + 1})/2 \rfloor\}$ , for any execution where at most  $f < n$  processes may crash. 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. Our first algorithm, for synchronous lattice agreement,  $LA_\alpha$ , requires  $\log h(L)$  rounds. It assumes that the height of the input lattice is known to all processes. By applying this algorithm as a building block, we give an algorithm,  $LA_\beta$ , which requires only  $\log f$  rounds without the height assumption in  $LA_\alpha$ . Instead of directly trying to decide on the comparable output values which are from a lattice with an unknown height, this algorithm first performs lattice agreement on the failure set known by each process by using  $LA_\alpha$ . Then each process removes values from *faulty* processes they know and outputs the join of all the remaining values. Our third algorithm,  $LA_\gamma$ , has round complexity of  $\min\{O(\log^2 h(L)), O(\log^2 f)\}$ , which depends on the height of the input lattice but does not assume that the height is known. This algorithm iteratively guesses the actual height of the input lattice and applies  $LA_\alpha$  with the guessed height as input, until all processes terminate.

Lattice agreement in asynchronous message passing systems is useful due to its applications in atomic snapshot objects and fault-tolerant replicated state machines. 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 (examples of algorithms in message passing systems based on snapshot objects can be found in [18], [13] and [4]). As shown in [2], any algorithm for lattice agreement can be applied to solve the atomic snapshot problem in a shared memory system. We note that [3] does not directly use lattice agreement to solve the atomic snapshot problem, but their idea of producing comparable *views* for processes is essentially lattice agreement. Thus, by using the same transformation techniques in [2] and [3], algorithms for lattice agreement problem can be directly applied to implement atomic snapshot objects in crash-prone message passing systems. We give an algorithm for asynchronous lattice agreement problem which requires  $\min\{O(h(L)), O(f)\}$  message delays. Then, by applying the technique in [3], our algorithm can be used to implement atomic snapshot objects on top of crash-prone asynchronous message passing systems and achieve time complexity of  $O(f)$  message delays in the worst case. Our result significantly improves the message delays in the previous work by Delporte-Gallet, Fauconnier et al [5]. The algorithm in [5] directly implements an atomic snapshot object on top of crash-prone message passing systems and requires  $O(n)$  message delays in the worst case.

Another related work for lattice agreement in asynchronous systems is by Faleiro et al. [7]. They solve the lattice agreement problem in asynchronous systems by giving a Paxos style protocol [10, 11], in which each proposer keeps proposing a value until it gets *accept* messages from a majority of acceptors. The acceptor only *accepts* a proposal when the proposal has a bigger value than its accepted value. Their algorithm requires  $O(n)$  message delays. Our asynchronous lattice agreement algorithm does not have Paxos style. Instead, it runs in *round-trips*. Each round-trip is composed of sending a message to all and getting  $n - f$  acknowledgements back. Our algorithm guarantees termination in  $\min\{O(h(L)), O(f)\}$  message delays which is a significant improvement over  $O(n)$  message delays.

Generalized lattice agreement problem defined in [7] is a generalization of the lattice agreement problem in asynchronous systems. It is applied to implement a specific class of replicated state machines. In conventional replicated state machine approach [14], consensus based mechanism is used to implement strong consistency. Due to performance reasons, many systems relax the strong consistency requirement and support eventual consistency [17], i.e, all copies are eventually consistent. However, there is no guarantee on when this eventual consistency happens. Also, different copies could be in an inconsistent state before this eventual situation happens. Conflict-free replicated data types (CRDT) [15, 16] is a data structure which supports such eventual consistency. In CRDT, all operations are designed to be commutative such that they can be concurrently executed without coordination. As shown in [7] by applying generalized lattice agreement on top of CRDT, the states of any two copies can be made comparable and thus provide linearizability guarantee [9] for CRDT.

The following example from [7] motivates generalized lattice agreement. Consider a replicated set data structure which supports adds and reads. Suppose there are two concurrent updates,  $add(a)$  and  $add(b)$ , and two concurrent reads on copy one and two respectively. By using CRDT, it could happen that the two reads return  $\{a\}$  and  $\{b\}$  respectively. This execution is not linearizable [9], because if  $add(a)$  appears before  $add(b)$  in the linear order, then no read can return  $\{b\}$ . On the other hand, if we use conventional consensus replicated state machine technique, then all operations would be coordinated including the two reads. This greatly impacts the throughput of the system. By applying generalized lattice agreement on top of CRDT, all operations can be concurrently executed and any two reads always return comparable views of the system. In the above example, the two reads return either (i)  $\{a\}$  and  $\{a, b\}$  or (ii)  $\{b\}$  and  $\{a, b\}$  which is linearizable. Therefore, generalized lattice agreement can be applied on top of CRDT to provide better consistency guarantee than CRDT and better availability than conventional replicated state machine technique.

Since the generalized lattice agreement problem has applications in building replicated state machines, it is important to reduce the message delays for a value to be learned. Faleiro et al. [7] propose an algorithm for the generalized lattice agreement by using their algorithm for the lattice agreement problem as a building block. Their generalized lattice agreement algorithm satisfies safety and liveness assuming  $f < \lceil \frac{n}{2} \rceil$ . A value is eventually learned in their algorithm after  $O(n)$  message delays in the worst case. Our algorithm guarantees that a value is learned in  $\min\{O(h(L)), O(f)\}$  message delays.

In summary, this paper makes the following contributions:

- We present an algorithm,  $LA_\alpha$  to solve the lattice agreement in synchronous system in  $\log h(L)$  rounds assuming  $h(L)$  is known. Using  $LA_\alpha$ , we propose an algorithm,  $LA_\beta$  to solve the standard lattice agreement problem in  $\log f$  rounds. This bound is significantly better than the previously known upper bounds of  $\log n$  by [3] and  $\min\{1 + h(L), \lceil (3 + \sqrt{8f + 1})/2 \rceil\}$  by [12] (and solves the open problem posed there). We also give an algorithm,  $LA_\gamma$  which runs in  $\min\{O(\log^2 h(L)), O(\log^2 f)\}$  rounds.

■ **Table 1** Previous Work and Our Results.

Problem	Protocol	Time	Message
LA sync	[3]	$O(\log n)$	$O(n^2)$
	[12]	$\min\{O(h(L)), O(\sqrt{f})\}$	$n^2 \cdot \min\{O(h(L)), O(\sqrt{f})\}$
	$LA_\alpha$	$O(\log h(L))$	$O(n^2 \log h(L))$
	$LA_\beta$	$O(\log f)$	$O(n^2 \log f)$
	$LA_\gamma$	$\min\{O(\log^2 h(L)), O(\log^2 f)\}$	$n^2 \cdot \min\{O(\log^2 h(L)), O(\log^2 f)\}$
LA async	[7]	$O(n)$	$O(n^3)$
	$LA_\delta$	$\min\{O(h(L)), O(f)\}$	$n^2 \cdot \min\{O(h(L)), O(f)\}$
GLA async	[7]	$O(n)$	$O(n^3)$
	$GLA_\alpha$	$\min\{O(h(L)), O(f)\}$	$n^2 \cdot \min\{O(h(L)), O(f)\}$

- For the lattice agreement problem in asynchronous systems, we give an algorithm,  $LA_\delta$  which requires  $2 \cdot \min\{h(L), f + 1\}$  message delays which improves the  $O(n)$  bound by [7].
- Based on the asynchronous lattice agreement algorithm, we present an algorithm,  $GLA_\alpha$ , to solve the generalized lattice agreement with time complexity  $\min\{O(h(L)), O(f)\}$  message delays which improves the  $O(n)$  bound by [7].

Related previous work and our results are summarized in Table 1.  $LA_{sync}$  and  $LA_{async}$  represent lattice agreement in synchronous systems and asynchronous systems, respectively.  $GLA_{async}$  represents generalized lattice agreement in asynchronous systems.  $LA_\alpha$  is designed to solve the lattice agreement problem with the assumption that the height of the input lattice is given. It serves as a building block for  $LA_\beta$  and  $LA_\gamma$ . For synchronous systems, the time complexity is given in terms of synchronous rounds. For asynchronous system, the time complexity is given in terms of message delays. The message column represents the total number of messages sent by all processes in one execution. For generalized lattice agreement problem, the message complexity is given in terms of the number of messages needed for a value to be learned.

## 2 System Model and Problem Definitions

### 2.1 System Model

We assume a distributed message passing system with  $n$  processes in a completely connected topology, denoted as  $p_1, \dots, p_n$ . We consider both synchronous or asynchronous systems. Synchronous means that message delays and the duration of the operations performed by the process have an upper bound on the time. Asynchronous means that there is no upper bound on the time for a message to reach its destination. The model assumes that processes may have crash failures but no Byzantine failures. The model parameter  $f$  denotes the maximum number of processes that may crash in a run. We assume that the underlying communication system is reliable but the message channel may not be FIFO. We say a process is *faulty* in a run if it crashes and *correct* or *non-faulty* otherwise. In our following algorithms, when a process sends a message to all, it also sends this message to itself.

### 2.2 Lattice Agreement

Let  $(X, \leq, \sqcup)$  be a finite join semi-lattice with a partial order  $\leq$  and join  $\sqcup$ . 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 a 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.

In the lattice agreement problem [2], each process  $p_i$  can propose a value  $x_i$  in  $X$  and must decide on some output  $y_i$  also in  $X$ . An algorithm is said to solve the lattice agreement problem if the following properties are satisfied:

**Downward-Validity:** For all  $i \in [1..n]$ ,  $x_i \leq y_i$ .

**Upward-Validity:** For all  $i \in [1..n]$ ,  $y_i \leq \sqcup\{x_1, \dots, x_n\}$ .

**Comparability:** For all  $i \in [1..n]$  and  $j \in [1..n]$ , either  $y_i \leq y_j$  or  $y_j \leq y_i$ .

In this paper, all the algorithms that we propose apply join operation to some subset of input values. Therefore, it is sufficient to focus on the join-closed subset of  $X$  that includes all input values. Let  $L$  be the join-closed subset of  $X$  that includes all input values.  $L$  is also a join semi-lattice. We call  $L$  the *input sublattice* of  $X$ . All algorithms proposed in this paper are based on  $L$ . Since the complexity of our algorithms depend on the height of lattice  $L$ , we give the formal definitions as below:

► **Definition 1.** The height of a value  $v$  in a lattice  $X$  is the length of longest path from any minimal value to  $v$ , denoted as  $h_X(v)$  or  $h(v)$  when it is clear.

► **Definition 2.** The height of a lattice  $X$  is the height of its largest value, denoted as  $h(X)$ .

Each process proposes a value from a boolean lattice. Thus, the largest value in this lattice is the set consists of all the  $n$  values. From the definition 2, we have  $h(L) \leq n$ .

### 2.3 Generalized Lattice Agreement

In generalized lattice agreement problem, each process may receive a possibly infinite sequence of values as inputs that belong to a lattice at any point of time. Let  $x_i^p$  denote the  $i$ th value received by process  $p$ . The aim is for each process  $p$  to learn a sequence of output values  $y_j^p$  which satisfies the following conditions:

**Validity:** Any learned value  $y_j^p$  is a join of some set of received input values.

**Stability:** The value learned by any process  $p$  is non-decreasing:  $j < k \implies y_j^p \leq y_k^p$ .

**Comparability:** Any two values  $y_j^p$  and  $y_k^q$  learned by any two process  $p$  and  $q$  are comparable.

**Liveness:** Every value  $x_i^p$  received by a correct process  $p$  is eventually included in some learned value  $y_k^q$  of every correct process  $q$ : i.e,  $x_i^p \leq y_k^q$ .

## 3 Lattice Agreement in Synchronous Systems

### 3.1 Lattice Agreement with Known Height

In this section, we first consider a simpler version of the standard lattice agreement problem by assuming that the height of the input sublattice  $L$  is known in advance, i.e,  $h(L)$  is given. We propose an algorithm,  $LA_\alpha$ , to solve this problem in  $\log h(L)$  synchronous rounds. In section 3.2, we give an algorithm to solve the lattice agreement problem when the height is not given using this algorithm.

Algorithm  $LA_\alpha$  runs in synchronous rounds. At each round, by calling a *Classifier* procedure (described below), processes within a same group (to be defined later) are classified into different groups. The algorithm guarantees that any two processes within the same group have equal values and any two processes in different groups have comparable values at the end. Thus, values of all processes are comparable to each other at the end. We present the algorithm by first introducing the fundamental *Classifier* procedure.

### 3.1.1 The Classifier Procedure

The *Classifier* procedure is inspired by the Classifier procedure given by Attiya and Rachman in [3], called *AR-Classifier*, where it is applied to solve the atomic snapshot problem in the shared memory system. The intuition behind the *Classifier* procedure is to classify processes to *master* or *slave* and ensure all *master* processes have values greater than all *slave* processes.

The pseudo-code for *Classifier* is given in Figure 1. It takes two parameters: the input value  $v$  and the threshold value  $k$ . The output is composed of three items: the output value, the classification result and the decision status. The process which calls the *Classifier* procedure should update their value to be the output value. The classification result is either *master* or *slave*. The decision status is a Boolean value which is used to inform whether the invoking process can decide on the output value or not. The main functionality of the *Classifier* procedure is either to tell the invoking process to decide, or to classify the invoking process as a *master* or a *slave*. Details of the *Classifier* procedure are shown below:

Line 1-3: The invoking process sends a message with its input value  $v$  and the threshold value  $k$  to all. It then collects all the received values associated with the threshold value  $k$  in a set  $U$ .

Line 5-6: It checks whether all values in  $U$  are comparable to the input value. If they are comparable, it terminates the *Classifier* procedure and returns the input value as the output value and *true* as the decision status.

Line 8-12: It performs classification based on received values. Let  $w$  be the join of all received values associated with the threshold value  $k$ . If the height of  $w$  in lattice  $L$  is greater than the threshold value  $k$ , then the *Classifier* returns  $w$  as the output value, *master* as the classification result and *true* as the decision status. Otherwise, it returns the input value as the output value, *slave* as the classification result and *false* as the decision status. From the classification steps, it is easy to see that the processes classified as *master* have values greater than those classified as *slave* because  $w$  is the join of all values in  $U$ .

There are four main differences between the *AR-Classifier* and our *Classifier*: 1) The *AR-Classifier* is based on the shared memory model whereas our algorithm is based on synchronous message passing. 2) The *AR-Classifier* does not allow early termination. 3) Each process in the *AR-Classifier* needs values from all processes whereas our *Classifier* uses values only from processes within its group. 4) The *AR-Classifier* procedure requires the invoking process to read values of all processes again if the invoking process is classified as *master* whereas our algorithm needs to receive values from all processes only once.

### 3.1.2 Algorithm $LA_\alpha$

Algorithm  $LA_\alpha$  (shown in Figure 2) runs in at most  $\log h(L)$  rounds. It assumes knowledge of  $H = h(L)$ , the height of the input lattice. Let  $x_i$  denote the initial input value of process  $i$ ,  $v_i^r$  denote the value held by process  $i$  at the beginning of round  $r$ , and  $class$  denote the classification result of the *Classifier* procedure. The  $class$  indicates whether the process is classified as a *master* or a *slave*. The  $decided$  variable shows whether the process has decided or not. Each process  $i$  has a label denoted as  $l_i$ . This label is updated at each round. Processes which have the same label  $l$  are said to be in the same group with label  $l$ . The definitions of *label* and *group* are formally given as:

► **Definition 3 (label).** Each process has a *label*, which serves as a knowledge threshold and is passed as the threshold value  $k$  whenever the process calls the *Classifier* procedure.

► **Definition 4 (group).** A *group* is a set of processes which have the same label. The label of a group is the label of the processes in this group.

```

Classifier( $v, k$ ):
 $v$ : input value     $k$ : threshold value
1: Send ( $v, k$ ) to all
2: Receive messages of the form ( $-, k$ )
3: Let  $U$  be values contained in received messages
3:
4: /* Early Termination */
5: if  $|U| = 0$  or  $\forall u \in U : v \leq u \vee u \leq v$ 
6:   return ( $v, -, true$ )
6:
7: /* Classification */
8: Let  $w := \sqcup\{u : u \in U\}$ 
9: if  $h(w) > k$ 
10:  return ( $w, master, false$ )
11: else
12:  return ( $v, slave, false$ )

```

■ **Figure 1** *Classifier*.

```

LA $_{\alpha}$ ( $H, x_i$ ) for  $p_i$ :
 $H$ : given height     $x_i$ : input value
1:  $v_i^1 := x_i$  // value at round 1
2:  $l_i := \frac{H}{2}$  // label
3:  $decided := false$ 
4:
5: for  $r := 1$  to  $\log H + 1$ 
6:   ( $v_i^{r+1}, class, decided$ )
   := Classifier( $l_i, v_i^r$ )
7:   if  $decided$ 
8:     return  $v_i^{r+1}$ 
9:   else if  $class = master$ 
10:     $l_i := l_i + \frac{H}{2^{r+1}}$ 
11:   else
12:     $l_i := l_i - \frac{H}{2^{r+1}}$ 
13: end for

```

■ **Figure 2** Algorithm *LA* $_{\alpha}$ .

A process has *decided* if it has set its decision status to true. Otherwise, it is undecided. At each round  $r$ , an undecided process invokes the *Classifier* procedure with its current value and its current label  $l_i$  as parameters  $v$  and  $k$ , respectively. Since each process passes its label as the threshold value  $k$  when invoking the *Classifier* procedure, line 2 of the *Classifier* is equivalent to receiving messages from processes within the same group; that is, at each round, a process performs the *Classifier* procedure within its group. Processes which are in different groups do not affect each other. At round  $r$ , by invoking the *Classifier* procedure, each process  $i$  sets  $v_i^{r+1}$ , *class* and *decided* to the returned output value, the classification result and the decision status. Each process first checks the value of *decided*. If it is true, process  $i$  decides on  $v_i^{r+1}$  and terminates the algorithm. Otherwise, if it is classified as a *master*, it increases its label by  $\frac{H}{2^{r+1}}$ . If it is classified as a *slave*, it decreases its label by  $\frac{H}{2^{r+1}}$ . Now we show how the *Classifier* procedure combined with this label update mechanism makes any two processes have comparable values at the end.

Let  $G$  be a group of processes at round  $r$ . Let  $M(G)$  and  $S(G)$  be the group of processes which are classified as *master* and *slave*, respectively, when they run the *Classifier* procedure in group  $G$ . We say that  $G$  is the parent of  $M(G)$  and  $S(G)$ . Thus,  $M(G)$  and  $S(G)$  are both groups at round  $r + 1$ . Process  $i \in M(G)$  or  $i \in S(G)$  indicates that  $i$  does not decide in group  $G$  at round  $r$ . Initially, all process have the same label  $\frac{H}{2}$  and are in the same group with label  $\frac{H}{2}$ . When they execute the *Classifier*, they will be classified into different groups. We can view the execution as processes traversing through a binary tree. Initially, all of them are at the root of the tree. As the program executes, if they are classified as *master*, then they go to the right child. Otherwise, they go to the left child.

Before we prove the correctness of the given algorithm, we first give some useful properties satisfied by the *Classifier* procedure. Although Lemma 5 is similar to a lemma given in [5], it is discussed here in message passing systems and the proofs are different.

► **Lemma 5.** *Let  $G$  be a group at round  $r$  with label  $k$ . Let  $L$  and  $R$  be two nonnegative integers such that  $L \leq k \leq R$ . If  $L < h(v_i^r) \leq R$  for every process  $i \in G$ , and  $h(\sqcup\{v_i^r : i \in G\}) \leq R$ , then*



## 41:8 Lattice Agreement in Message Passing Systems

- (p1) for each process  $i \in M(G)$ ,  $k < h(v_i^{r+1}) \leq R$
- (p2) for each process  $i \in S(G)$ ,  $L < h(v_i^{r+1}) \leq k$
- (p3)  $h(\sqcup\{v_i^{r+1} : i \in M(G)\}) \leq R$
- (p4)  $h(\sqcup\{v_i^{r+1} : i \in S(G)\}) \leq k$ , and
- (p5) for each process  $i \in M(G)$ ,  $v_i^{r+1} \geq \sqcup\{v_i^{r+1} : i \in S(G)\}$

**Proof.**

(p1)–(p3): Immediate from the *Classifier* procedure.

(p4): Since  $S(G)$  is a group of processes which are at round  $r + 1$ , all processes in  $S(G)$  are correct (non-faulty) at round  $r$ . So, all processes in  $S(G)$  must have received values of each other in the *Classifier* procedure at round  $r$  in group  $G$ . Thus,  $h(\sqcup\{v_i^{r+1} : i \in S(G)\}) \leq k$ , otherwise all of them should be in group  $M(G)$  instead of  $S(G)$ , according to the condition at *line 9* of the *Classifier* procedure.

(p5): Since all processes in  $S(G)$  are correct at round  $r$ , all processes in  $M(G)$  must have received values of all processes in  $S(G)$  in the *Classifier* procedure at round  $r$ . Any process which proceeds to group  $M(G)$  takes the join of all received values at round  $r$ , according to *line 10*. Thus, for every process  $i \in M(G)$ ,  $v_i^{r+1} \geq \sqcup\{v_i^{r+1} : i \in S(G)\}$ . ◀

► **Lemma 6.** *Let  $x$  be a value from a lattice  $L$ , and  $V$  be a set of values from  $L$ . Let  $U$  be any subset of  $V$ . If  $x$  is comparable with  $\forall v \in V$ , then  $x$  is comparable with  $\sqcup\{u \mid u \in U\}$ .*

**Proof.** If  $\forall u \in U : u \leq x$ , then  $\sqcup\{u \mid u \in U\} \leq x$ . Otherwise,  $\exists y \in U : x \leq y$ . Since  $y \leq \sqcup\{u \mid u \in U\}$ , so  $x \leq \sqcup\{u \mid u \in U\}$ . ◀

► **Lemma 7.** *If process  $i$  decides at round  $r$  on value  $y_i$ , then  $y_i$  is comparable with  $v_j^r$  for any correct process  $j$ .*

**Proof.** Let process  $i$  decide in group  $G$  at round  $r$ . Consider the two cases below:

Case 1:  $j \notin G$ . Let  $G'$  be a group at the maximum round  $r'$  such that both  $i$  and  $j$  belong to  $G'$ . Then, either  $i \in M(G') \wedge j \in S(G')$  or  $j \in M(G') \wedge i \in S(G')$ . We only consider the case  $i \in M(G') \wedge j \in S(G')$ . The other case is similar. From (p5) of Lemma 5, we have  $\sqcup\{v_p^r : p \in S(G')\} \leq y_i$ . Since  $j \in S(G')$ , then  $v_j^r \leq \sqcup\{v_p^r : p \in S(G')\}$ . Thus,  $v_j^r \leq y_i$ . For the other case, we have  $y_i \leq v_j^r$ . Therefore,  $y_i$  is comparable with  $v_j^r$ .

Case 2:  $j \in G$ , since process  $j$  is correct, then  $i$  must have received  $v_j^r$  at round  $r$ . Thus, by *line 5* of the *Classifier* procedure, we have that  $y_i$  is comparable with  $y_j^r$ . ◀

Now we show that any two processes decide on comparable values.

► **Lemma 8.** (*Comparability*) *Let process  $i$  and  $j$  decide on  $y_i$  and  $y_j$ , respectively. Then  $y_i$  and  $y_j$  are comparable.*

**Proof.** Let process  $i$  and  $j$  decide at round  $r_i$  and  $r_j$ , respectively. Without loss of generality, assume  $r_i \leq r_j$ . At round  $r_i$ , from Lemma 7 we have  $y_i$  is comparable with  $v_k^r$  for any correct undecided process  $k$ . Let  $V = \{v_k^{r_i} \mid \text{process } k \text{ undecided and correct}\}$ . Since  $r_j \geq r_i$ ,  $y_j$  is at most the join of a subset of  $V$ . Thus, from Lemma 6 we have  $y_i$  and  $y_j$  are comparable. ◀

Now we prove that all processes decide within  $\log H + 1$  rounds by showing all processes in the same group at the beginning of round  $\log H + 1$  have equal values, given by Lemma 9 and Lemma 10. Since Lemma 9 and Lemma 10 and the corresponding proofs are similar to the ones given in [3], the proofs are omitted here and can be found in the full paper. Proof of Lemma 9 is based on (p1-p4) of Lemma 5 by induction. Proof of Lemma 10 is based on Lemma 9.



► **Lemma 9.** *Let  $G$  be a group of processes at round  $r$  with label  $k$ . Then*

- (1) *for each process  $i \in G$ ,  $k - \frac{H}{2^r} < h(v_i^r) \leq k + \frac{H}{2^r}$*   
 (2)  *$h(\sqcup\{v_i^r : i \in G\}) \leq k + \frac{H}{2^r}$*

► **Lemma 10.** *Let  $i$  and  $j$  be two processes that are within the same group  $G$  at the beginning of round  $r = \log H + 1$ . Then  $v_i^r$  and  $v_j^r$  are equal.*

► **Lemma 11.** *All processes decide within  $\log H + 1$  rounds.*

**Proof.** From Lemma 10, we know any two processes which are in the same group at the beginning of round  $\log H + 1$  have equal values. Then, the condition in *line 5* of *Classifier* procedure is satisfied. Thus, all undecided processes decide at round  $\log H + 1$ . ◀

► **Remark 12.** Since at the beginning of round  $\log H + 1$  all undecided processes have comparable values,  $LA_\alpha$  only needs  $\log H$  rounds. For simplicity, one more round is executed to make all processes decide at *line 5* of the *Classifier* procedure.

► **Theorem 13.** *Algorithm  $LA_\alpha$  solves lattice agreement problem in  $\log H$  rounds and can tolerate  $f < n$  failures.*

**Proof.** *Downward-Validity* follows from the fact that the value of each process is non-decreasing at each round. For *Upward-Validity*, according to the *Classifier* procedure, each process either keeps its value unchanged or takes the join of the values proposed by other processes which could never be greater than  $\sqcup\{x_1, \dots, x_n\}$ . For *Comparability*, from Lemma 8, we know for any two process  $i$  and  $j$ , if they decide, then their decision values must be comparable. From Lemma 11, we know all processes decide. Thus, comparability holds. ◀

**Complexity.** Time complexity is  $\log H$  rounds. For message complexity, since each process sends  $n$  messages per round,  $\log H$  rounds results in  $n^2 \log H$  messages in total. Notice that the number of messages can be further reduced by keeping a set of processes which are not in its group. If a process  $p$  receives a message from process  $q$  with a threshold value different from its own threshold value, it knows that  $q$  is not in its group. Each process does not send messages to the processes in this set.

Algorithm  $LA_\beta$  runs in  $\log \text{height}(L)$  rounds by assuming that  $\text{height}(L)$  is given. However, in order to know that actual height of input lattice, we need to know how many distinct values all process propose which needs extra effort. For this reason, in following sections, we introduce algorithms to solve the lattice agreement problem without this assumption.

## 3.2 Lattice Agreement with Unknown Height

In this section, we consider the standard lattice agreement in which the height of the lattice is not known to any process. We propose algorithm,  $LA_\beta$ , (shown in Figure 3) based on algorithm  $LA_\alpha$ .

### 3.2.1 Algorithm $LA_\beta$

Algorithm  $LA_\beta$  runs in  $\log f + 1$  synchronous rounds. It makes use of algorithm  $LA_\alpha$  as a building block. Instead of directly agreeing on input values which are taken from a lattice with unknown height, we first do lattice agreement on the failure set that each process knows *after one round of broadcast*. The set of all failure sets forms a boolean lattice with union as the join operation and with height equal to  $f$  (since there are at most  $f$  failures). The algorithm consists of two phases. At *Phase A*, all processes exchange their values. Process  $i$

<p><b><math>LA_\beta</math> for <math>p_i</math></b></p> <pre> 1: <math>V_i := \{x_i\}</math> // set of values, initially <math>x_i</math> 2: <math>F_i := \emptyset</math> // set of known failure processes 3: <math>f :=</math> the maximum number of failures 4: /* Exchange Values and Record Failures */ 5: <b>Phase A:</b> 6: Send <math>V_i</math> to all 7: <b>for</b> <math>j := 1</math> to <math>n</math> 8:   <b>if</b> <math>V_j</math> is received from process <math>j</math> 9:     <math>V_i := V_i \cup V_j</math> 10:  <b>else</b> 11:    <math>F_i := F_i \cup j</math> 12:  <b>end for</b> 13: /* LA with Known Height <math>f</math> */ 14: <b>Phase B:</b> 15: <math>F'_i := LA_\alpha(f, F_i)</math> 16: <math>U_i :=</math> values from processes in <math>F'_i</math> in <i>Phase A</i> 17: <math>C_i := V_i - U_i</math> // set of correct values 18: <math>y_i := \sqcup \{v : v \in C_i\}</math> </pre>	<p><b><math>LA_\gamma</math> for <math>p_i</math></b></p> <pre> 1: <math>v_i := x_i</math> // input value 2: <math>decided := false</math> 3: /* Exchange Values */ 4: <b>Phase A:</b> 5: Send <math>v_i</math> to all 6: <b>for</b> <math>j := 1</math> to <math>n</math> 7:   receive <math>v_j</math> from <math>p_j</math> 8:   <math>v_i := v_i \sqcup v_j</math> 9: <b>end for</b> 10: /* Guessing Height */ 11: <b>Phase B:</b> 12: <math>guess := 2</math> // guess height 13: <b>while</b> (<math>\neg decided</math>) 14:   <math>v_i := LA_\alpha(guess, v_i)</math> 15:   <math>guess := 2 * guess</math> 16: <b>end while</b> 17: <math>y_i := v_i</math> </pre>
---	---

■ **Figure 3** Algorithm  $LA_\beta$ .

■ **Figure 4** Algorithm  $LA_\gamma$ .

includes  $j$  into its failure set if it does not receive value from process  $j$  at the first phase. After the first phase, each process has a failure set which contains failed processes it knows. Then in *phase B*, they invoke algorithm  $LA_\alpha$  with  $f$  as the height and its failure set as input. After that, each process *decides* on a failure set which satisfies lattice agreement properties. The new failure set of any two process  $i$  and  $j$  are comparable to each other, i.e.,  $F'_i$  is comparable to  $F'_j$ . Equipped with this comparable failure set, each process removes values it received from processes which are in its failure set and decides on the join of the remaining values.

The following lemma shows that any two processes decide on comparable values. We only give the sketch of proof, and the detailed proof is available in the full paper.

► **Lemma 14.** (*Comparability*) *Let process  $i$  and  $j$  decide on  $y_i$  and  $y_j$ , respectively. Then  $y_i$  and  $y_j$  are comparable.*

**Proof sketch.** According to comparability of  $LA_\alpha$ , all processes have comparable failure sets. Then, the set of values they received at *Phase A* from correct processes must be comparable, i.e.,  $C_i$  is comparable with  $C_j$ . Therefore,  $y_i$  and  $y_j$  are comparable. ◀

► **Theorem 15.**  *$LA_\beta$  solves lattice agreement problem in  $\log f + 1$  rounds, where  $f < n$  is the maximum number of failures the system can tolerate.*

**Proof.** *Downward-Validity.* Initially, for correct process  $i$ ,  $v_i = x_i$ . After *Phase A*, since  $i$  is correct, so  $i$  is not in any failure set of any process. At *Phase B*, process  $i$  invokes algorithm  $LA_\alpha$  with failure set as the input value. Thus, according to the *Upward-Validity* of  $LA_\alpha$ ,  $i$

is not included in  $F'_i$ . So,  $x_i \in C_i$ . Therefore,  $x_i \leq y_i$ . *Upward-Validity* is immediate from the fact that each process receives at most all values by all processes. *Comparability* follows from Lemma 14. ◀

### 3.2.2 Algorithm $LA_\gamma$

Algorithm  $LA_\beta$  solves lattice agreement in  $\log f + 1$  rounds whereas Algorithm  $LA_\alpha$  solves lattice agreement in  $\log h(L)$  rounds assuming  $h(L)$  is given. We now propose an algorithm to solve lattice agreement which has round complexity related to  $h(L)$  even when  $h(L)$  is not known. This algorithm called  $LA_\gamma$  (shown in Figure 4), solves the standard lattice agreement in  $O(\min\{\log^2 h(L), \log^2 f\})$  rounds. The basic idea is to “guess” the height of  $L$  and apply algorithm  $LA_\alpha$  using the guessed height as input. The algorithm is composed of two phases. At *Phase A*, each process simply broadcasts its value and takes the join of all received values. *Phase B* is the guessing phase which invokes algorithm  $LA_\alpha$  repeatedly. Notice that *decided* variable is updated at *line 6* of  $LA_\alpha$ .

Let  $w_i$  denote the value of  $v_i$  after *Phase A*. Let  $\Psi$  denote the sublattice formed by values of all correct processes after *Phase A*, i.e.,  $\Psi = \{u \mid (u \in L) \wedge (\exists i : w_i \leq u)\}$ . Since there are at most  $f$  failures, we have  $h(\Psi) \leq f$ . Now we show that *Phase B* terminates in at most  $\lceil \log h(\Psi) \rceil$  executions of  $LA_\alpha$ . We call the  $i$ -th execution of  $LA_\alpha$  as iteration  $i$ . Notice that the guessed height of iteration  $i$  is  $2^i$ .

► **Lemma 16.** *After iteration  $\lceil \log h(\Psi) \rceil$  of  $LA_\alpha$  at *Phase B*, all processes decide.*

**Proof.** Since  $2^{\lceil \log h(\Psi) \rceil} \geq h(\Psi)$ , Lemma 9 still holds which implies Lemma 10. Thus, all undecided processes have equal values at the last round of iteration  $\lceil \log h(\Psi) \rceil$ . Therefore, all undecided processes decide after iteration  $\lceil \log h(\Psi) \rceil$ . ◀

We now show that two processes decide on comparable values irrespective of whether they both decide on the same iteration of  $LA_\alpha$ .

► **Lemma 17.** (*Comparability*) *Let  $i$  and  $j$  be any two processes that decide on value  $y_i$  and  $y_j$ , respectively. Then  $y_i$  and  $y_j$  are comparable.*

**Proof.** Assume process  $i$  decides on  $G_i$  at round  $r_i$  of execution  $e_i$  of  $LA_\alpha$  and process  $j$  decides on  $G_j$  at round  $r_j$  of execution  $e_j$  of  $LA_\alpha$ . If  $e_i = e_j$ , then  $y_i$  and  $y_j$  are comparable by Lemma 8. Otherwise,  $e_i \neq e_j$ . Without loss of generality, suppose  $e_i < e_j$ . Consider round  $r_i$  of execution  $e_i$  of  $LA_\alpha$ . Since  $i$  decides on value  $y_i$  at this round, then from Lemma 7, we have that  $y_i$  is comparable with  $v_k^r$  for any correct process  $k$ . Let  $V = \{v_k^r \mid k \text{ is correct}\}$ . Then,  $y_i$  is at most the join of a subset of  $V$ . From Lemma 6, it follows that  $y_i$  is comparable with  $y_j$ . ◀

► **Theorem 18.**  *$LA_\gamma$  solves the lattice agreement problem and can tolerate  $f < n$  failures.*

**Proof.** *Downward-Validity* follows from that fact that the value of each process is non-decreasing along the execution. *Upward-Validity* follows since each process can receive at most all values from all processes. *Comparability* holds by Lemma 17. ◀

**Complexity.** From Lemma 16, we know *Phase B* terminates in at most  $\lceil \log h(\Psi) \rceil$  executions of  $LA_\alpha$ . Thus, *Phase B* takes  $\log 2 + \log 4 + \dots + \lceil \log h(\Psi) \rceil = \frac{(\lceil \log h(\Psi) \rceil + 1) * (\lceil \log h(\Psi) \rceil)}{2}$  rounds in worst case. Since  $h(\Psi) \leq f$  and  $h(\Psi) \leq h(L)$ ,  $LA_\gamma$  has round complexity of  $\min\{O(\log^2 h(L)), O(\log^2 f)\}$ . Each process sends  $n$  messages at each round, thus message complexity is  $n^2 \cdot \min\{O(\log^2 h(L)), O(\log^2 f)\}$ .

<p><b><math>LA_\delta</math></b> for <math>p_i</math></p> <p><math>acceptVal := x_i</math> // accept value  <math>learnedVal := \perp</math> // learned value</p> <p><b>on receiving</b> <math>prop(v_j, r)</math> from <math>p_j</math>:</p> <p><b>if</b> <math>v_j \geq acceptVal</math>      Send <math>ACK("accept", -, r)</math>      <math>acceptVal := v_j</math></p> <p><b>else</b>      Send <math>ACK("reject", acceptVal, r)</math></p>	<p><b>for</b> <math>r := 1</math> <b>to</b> <math>f + 1</math></p> <p><math>val := acceptVal</math>  Send <math>prop(val, r)</math> to all  <b>wait for</b> <math>n - f</math> <math>ACK(-, -, r)</math> messages  let <math>V_r</math> be values contained in <i>reject</i> <math>ACKs</math>  let <math>tally</math> be number of <i>accept</i> <math>ACKs</math>  <b>if</b> <math>tally &gt; \frac{n}{2}</math>      <math>learnedVal := val</math>      <b>break</b></p> <p><b>else</b>      <math>acceptVal := acceptVal \sqcup \{v \mid v \in V_r\}</math></p> <p><b>end for</b></p>
--	--

■ **Figure 5** Algorithm  $LA_\delta$ .

## 4 Lattice Agreement in Asynchronous Systems

In this section, we discuss the lattice agreement problem in asynchronous systems. The algorithm proposed in [7] requires  $O(n)$  units of time, whereas our algorithm ( $LA_\delta$  shown in Figure 5) requires only  $O(f)$  units of time. We first note that

► **Theorem 19.** *The lattice agreement problem cannot be solved in asynchronous message systems if  $f \geq \frac{n}{2}$ .*

**Proof.** The proof follows from the standard partition argument. If two partitions have incomparable values then they can never decide on comparable values. ◀

### 4.1 Algorithm $LA_\delta$

On account of Theorem 19, we assume that  $f < \frac{n}{2}$ . The algorithm proceeds in *round-trips*. A single round-trip is composed of sending messages to all and getting  $n - f$  acknowledgement messages back. At each round-trip, a process sends a *prop* message to all, with its current accepted value as the proposal value, and waits for  $n - f$   $ACK$  messages. If majority of these  $ACK$  messages are *accept*, then it decides on its current proposed value. Otherwise, it updates its current accept value to be the join of all values received and starts next round-trip. Whenever a process receives a proposal, i.e., a *prop* message, if the proposal has a value at least as big as its current value, then it sends back an  $ACK$  message with *accept* and updates its current accept value to be the received proposal value. Otherwise, it sends back an  $ACK$  message with *reject*.

Let  $acceptVal_i^r$  denote the accept value (variable  $acceptVal$ ) held by  $p_i$  at the beginning of round-trip  $r$ . Let  $L^{(r)} = \{u \mid (u \in L) \wedge (\exists i : acceptVal_i^r \leq u)\}$ , i.e.,  $L^{(r)}$  denotes the join-closed subset of  $L$  that includes the accept values held by all undecided processes at the beginning of the round-trip  $r$ . Notice that  $L^{(1)} = L$ .

► **Lemma 20.** *For any round-trip  $r$ ,  $h(L^{(r+1)}) < h(L^{(r)})$ .*

**Proof.** If a process decides at round-trip  $r$ , its value is not in  $L^{(r+1)}$ . So, we only need to prove that  $h(acceptVal_i^r) < h(acceptVal_i^{r+1})$  for any process  $i$  which does not decide at round-trip  $r$ . The fact that process  $i$  does not decide at round-trip  $r$  implies that  $i$  must have received at least one *reject*  $ACK$  with a greater value. Since  $acceptVal_i^{r+1}$  is the join of all values received at round-trip  $r$ ,  $acceptVal_i^r < acceptVal_i^{r+1}$ . Hence,  $h(acceptVal_i^r) < h(acceptVal_i^{r+1})$  for any undecided process  $i$ . Therefore,  $h(L^{(r+1)}) < h(L^{(r)})$ . ◀

► **Lemma 21.** *All process decide within  $\min\{h(L), f + 1\}$  asynchronous round-trips.*

**Proof.** We first show that  $h(L^{(2)}) \leq f$ . At the first round-trip, each process receives  $n - f$  ACKs, which is equivalent to receiving  $n - f$  values. Therefore,  $h(L^{(2)}) \leq f$ . Let  $r_{min} = \min\{h(L), f + 1\}$ . Combining the fact that  $h(L^{(2)}) \leq f$  with Lemma 20, we have  $h(L^{(r_{min})}) \leq 1$ . This means that undecided correct processes have the same value. Thus, all of them receive  $n - f$  ACK messages with *accept* and decide. Therefore, all processes decide within  $\min\{h(L), f + 1\}$  round-trips. ◀

We note here that the algorithm in [7] takes  $O(n)$  message delays for a value to be learned in the worst case. A crucial difference between  $LA_\delta$  and the algorithm in [7] is that  $LA_\delta$  starts with the accepted value as the input value. Hence, after the first round-trip, there is a significant reduction in the height of the sublattice, from  $n$  initially (in the worst case) to  $f$ . In [7], acceptors start with the accepted value as null. Hence, there is reduction of height by only 1 in the worst case. Since in their algorithm, acceptors are different from proposers (in the style of Paxos), acceptors do not have access to the proposed values.

► **Theorem 22.** *Algorithm  $LA_\delta$  solves the lattice agreement problem in  $\min\{h(L), f + 1\}$  round-trips.*

**Proof.** *Down-Validity* holds since the accept value is non-decreasing for any process  $i$ . *Upward-Validity* follows because each learned value must be the join of a subset of all initial values which is at most  $\sqcup\{x_1, \dots, x_n\}$ . For *Comparability*, suppose process  $i$  and  $j$  decide on values  $y_i$  and  $y_j$ . There must be at least one process that has accepted both  $y_i$  and  $y_j$ . Since each process can only *accept* comparable values. Thus, we have either  $y_i \leq y_j$  or  $y_j \leq y_i$ . ◀

**Complexity.** From Lemma 21, we know that  $LA_\delta$  takes at most  $\min\{h(L), f + 1\}$  round-trips, which results in  $2 \cdot \min\{h(L), f + 1\}$  message delays, since one round-trip takes two message delays. At each round-trip, each process sends out at most  $2n$  messages. Thus, the number of messages for all processes is at most  $2 \cdot n^2 \cdot \min\{h(L), f + 1\}$ .

## 5 Generalized Lattice Agreement

In this section, we discuss the generalized lattice agreement problem as defined in Section 2.3. Since it is easy to adapt algorithms for lattice agreement in synchronous systems to solve generalized lattice agreement problem, we only consider asynchronous systems. We show how to adapt  $LA_\delta$  to solve the generalized lattice agreement problem (algorithm  $GLA_\alpha$  shown in Figure 6) in  $\min\{O(h(L)), O(f)\}$  units of time.

### 5.1 Algorithm $GLA_\alpha$

$GLA_\alpha$  invokes the `Agree()` procedure to learn a new value multiple times. The `Agree()` procedure is an execution of  $LA_\delta$  with some modifications (to be given later). A sequence number is associated with each execution of the `Agree()` procedure, thus each correct process has a learned value for each sequence number. The basic idea of  $GLA_\alpha$  is to let all processes sequentially execute  $LA_\delta$  to learn values, and make sure: 1) any two learned values for the same sequence number are comparable, 2) any learned value for a bigger sequence number is at least as big as any learned value for a smaller sequence number. The first goal can be simply achieved by invoking  $LA_\delta$  with the sequence number. In order to achieve the second goal, the key idea is to make any proposal for sequence number  $s + 1$  to be at least as big

as the largest learned value for sequence number  $s$ . Notice that at each round-trip of  $LA_\delta$  execution, a process waits for  $n - f$  *ACKs*, and any two set of  $n - f$  processes have at least one process in common. Thus, the second goal can be achieved by making sure at least  $n - f$  processes know the largest learned value after execution of  $LA_\delta$  for a sequence number.

Upon receiving a value  $v$  from client in a message tagged with *ClientValue*, a process adds  $v$  into its buffer and sends a *ServerValue* message with  $v$  to all other processes. The process can start to learn new values only when it succeeds at its current proposal. Otherwise,  $LA_\alpha$  may not terminate, as shown by an example in [7]. Upon receiving a *ServerValue* message with value  $v$ , a process simply adds  $v$  to its buffer.

The *Agree()* procedure is automatically executed when the guard condition is satisfied; that is, it is not currently proposing a value and it has some value in its buffer or it has seen a sequence number bigger than its current sequence number. Inside the *Agree()* procedure, a process first updates its *acceptVal* to be the join of current *acceptVal* and *buffVal*. Then, it starts an adapted  $LA_\delta$  execution. The original  $LA_\delta$  and adapted  $LA_\delta$  differ in the following ways: 1) Each message in the adapted  $LA_\delta$  is associated with a sequence number. 2) A process can also decide on a value for a sequence number if it receives any *decide ACK* message for that sequence number. 3) On receiving a *prop* message associated with a sequence number  $s'$ , if  $s'$  is smaller than its current sequence number which means it has learned a value for  $s'$ , then it simply sends *ACK* message with its learned value for  $s'$  back. If  $s'$  is greater than its current sequence number, it updates its *maxSeq* and waits until its current sequence number matches  $s'$ . After that it sends back *ACK* message with *accept* or *reject* based on whether the proposal value is bigger than its current *accept* value or not. The reason a process keeps track of the maximal sequence number it has ever seen, is to make sure each process has a learned value for each sequence number. When the maximum sequence number is bigger than its current sequence number, it has to invoke *Agree()* procedure even if it does not have any new value to propose. After execution of adapted  $LA_\delta$ , a process increases its current sequence number.

We next show the correctness of  $GLA_\alpha$ . Let  $acceptVal_s^p$  denote the *acceptVal* of process  $p$  at the end of *Agree()* procedure for sequence number  $s$ . Let  $LV_p$  denote the map of sequence number to learned value (variable *LV*) for process  $p$  and  $m_s = \sqcup\{LV_p[s] : p \in [1..n]\}$ , i.e.,  $m_s$  denotes the join of all learned values for sequence number  $s$ . Let  $LP_s = \{p \mid (p \in [1..n]) \wedge (m_s \leq acceptVal_s^p)\}$ , i.e.,  $LP_s$  is the set of processes which have *acceptVal* greater than the join of all learned values for the sequence number  $s$ . Notice that a process has two ways to learn a value for its current sequence number in the *Agree()* procedure: 1) by receiving a majority of *accept ACKs*. 2) by receiving some *decide ACKs*.

The following lemma proves that the adapted  $LA_\alpha$  satisfies the first goal.

► **Lemma 23.** *For any sequence number  $s$ ,  $LV_p[s]$  is comparable with  $LV_q[s]$  for any two processes  $p$  and  $q$ .*

**Proof.** We only need to show that any two processes which learn by the first way must learn comparable values, since processes which learn by the second way simply learn values from processes which learn by the first way. By the same reasoning as *Comparability* of Theorem 22, we know this is true. ◀

From Lemma 23, we know that  $m_s$  is the largest learned value for sequence number  $s$ .

► **Lemma 24.** *For any sequence number  $s$ ,  $|LP_s| > \frac{n}{2}$ .*

<p><b><math>GLA_\alpha</math></b> for <math>p_i</math></p> <p><math>s := 0</math> // sequence number  <math>maxSeq := -1</math> // max seq number seen  <math>buffVal := \perp</math> // received values  /* map from seq to learned value */  <math>LV := \perp</math>  <math>acceptVal := \perp</math>  <math>active := false</math></p> <p><b>on receiving</b> <math>ClientValue(v)</math>:  <math>buffVal := buffVal \sqcup v</math>  Send <math>ServerValue(v)</math> to all</p> <p><b>on receiving</b> <math>ServerValue(v)</math>:  <math>buffVal := buffVal \sqcup v</math></p> <p><b>on receiving</b> <math>prop(v_j, r, s')</math> from <math>p_j</math>:  <b>if</b> <math>s' &lt; s</math>  Send <math>ACK("decide", LV[s'], r, s')</math>  <b>break</b>  <b>if</b> <math>s' &gt; s</math>  <math>maxSeq := \max\{s', maxSeq\}</math>  <b>wait until</b> <math>s = s'</math>  <b>if</b> <math>v_j \geq acceptVal</math>  Send <math>ACK("accept", -, r, s')</math>  <math>acceptVal := v_j</math>  <b>else</b>  Send <math>ACK("reject", acceptVal, r, s')</math></p>	<p><b>Procedure Agree():</b>  <b>guard:</b> (<math>active = false</math>)  <math>\wedge (buffVal \neq \perp \vee maxSeq \geq s)</math>  <b>effect:</b>  <math>active := true</math>  <math>acceptVal := buffVal \sqcup acceptVal</math>  <math>buffVal := \perp</math></p> <p>/* <math>LA_\delta</math> with sequence number */  <b>for</b> <math>r := 1</math> <b>to</b> <math>f + 1</math>  <math>val := acceptVal</math>  Send <math>prop(val, r, s)</math> to all  <b>wait for</b> <math>n - f</math> <math>ACK(-, -, r, s)</math>  let <math>V</math> be values in <i>reject ACKs</i>  let <math>D</math> be values in <i>decide ACKs</i>  let <math>tally</math> be number of <i>accept ACKs</i>  <b>if</b> <math> D  &gt; 0</math>  <math>val := \sqcup\{d \mid d \in D\}</math>  <b>break</b>  <b>else if</b> <math>tally &gt; \frac{n}{2}</math>  <b>break</b>  <b>else</b>  <math>acceptVal := acceptVal \sqcup \{v \mid v \in V\}</math>  <b>end for</b>  <math>LV[s] := val</math>  <math>s := s + 1</math>  <math>active := false</math></p>
--	---

■ **Figure 6** Algorithm  $GLA_\alpha$ .

**Proof.** Consider Agree() procedure for  $s$ . Since  $m_s$  is the largest learned value for sequence number  $s$ , there must exist a process  $p$  which learns  $m_s$  by the first way. Thus,  $p$  must have received a majority of *accept ACKs*, which means at least a majority of processes have *acceptVal* greater than  $m_s$  after Agree() procedure for  $s$ . Therefore,  $|LP_s| > \frac{n}{2}$ . ◀

The lemma below shows that  $GLA_\alpha$  achieves the second goal.

► **Lemma 25.**  $m_s \leq LV_p[s + 1]$  for any process  $p$  and any sequence number  $s$ .

**Proof.** From Lemma 24, we know for sequence number  $s$  at least a majority of processes have *acceptVal* greater than  $m_s$ . To decide on  $LV_p[s + 1]$ , process  $p$  must get majority *accept*. Since any two majority has at least one process in common,  $m_s \leq LV_p[s + 1]$ . ◀

► **Theorem 26.** Algorithm  $GLA_\alpha$  solves generalized lattice agreement when a majority of processes is correct.



**Proof.** *Validity* holds since any learned value is the join of a subset of values received.

*Stability.* From Lemma 25 and the fact that  $LV_p[s] \leq m_s$ , we have that  $LV_p[s] \leq LV_p[s+1]$  for any process  $p$  and any sequence number  $s$ , which implies *Stability*.

*Comparability.* We need to show that  $LV_p[s]$  and  $LV_q[s']$  are comparable for any two processes  $p$  and  $q$ , and for any two sequence number  $s$  and  $s'$ . If  $s = s'$ , this is immediate from Lemma 23. Now consider the case when  $s \neq s'$ . Without loss of generality, assume  $s < s'$ . From Lemma 25, we can conclude that  $LV_p[s] \leq LV_q[s']$ . Thus, *comparability* holds.

*Liveness.* Any received value  $v$  is eventually included in some proposal, i.e., *prop* message. From Theorem 22, we know that in at most  $2 \cdot \min\{h(L), f + 1\}$  message delays that proposal value will be included in some learned value. Thus,  $v$  will be learned eventually. ◀

**Complexity.** For time complexity, from the analysis for *liveness* in Theorem 26, we know that a received value is learned in at most  $2 \cdot \min\{h(L), f + 1\}$  message delays. For message complexity, since each process sends out  $n$  messages per *round-trip*, the total number of messages needed to learn a value is  $2 \cdot n^2 \cdot \min\{h(L), f + 1\}$ .

## 6 Conclusions

We have presented algorithms for the lattice agreement problem and the generalized lattice agreement problem. These algorithms achieve significantly better time complexity than previous algorithms. For future work, we would like to know the answers to the following two questions: 1) Is  $\log f$  rounds a lower bound for lattice agreement in synchronous message passing systems? 2) Is  $O(f)$  message delays optimal for the lattice agreement and generalized lattice agreement problem in asynchronous message passing systems?

---

### 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  $\mathcal{O}(n \log n)$  operations. *SIAM Journal on Computing*, 27(2):319–340, 1998.
- 4 Hagit Attiya and Jennifer Welch. *Distributed computing: fundamentals, simulations, and advanced topics*, volume 19. John Wiley & Sons, 2004.
- 5 Carole Delporte-Gallet, Hugues Fauconnier, Sergio Rajsbaum, and Michel Raynal. Implementing snapshot objects on top of crash-prone asynchronous message-passing systems. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 341–355. Springer, 2016.
- 6 Danny Dolev and H Raymond Strong. Authenticated algorithms for byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983.
- 7 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.
- 8 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.
- 9 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.

- 10 Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.
- 11 Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.
- 12 Marios Mavronicolas. A bound on the rounds to reach lattice agreement, 2000. URL: <http://www.cs.ucy.ac.cy/~mavronic/pdf/lattice.pdf>.
- 13 Michel Raynal. *Concurrent programming: algorithms, principles, and foundations*. Springer Science & Business Media, 2012.
- 14 Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.
- 15 Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems*, pages 386–400. Springer, 2011.
- 16 Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Convergent and commutative replicated data types. *Bulletin-European Association for Theoretical Computer Science*, 104:67–88, 2011.
- 17 Andrew S Tanenbaum and Maarten Van Steen. *Distributed systems: principles and paradigms*. Prentice-Hall, 2007.
- 18 Gadi Taubenfeld. *Synchronization algorithms and concurrent programming*. Pearson Education, 2006.



# Brief Announcement: Local Distributed Algorithms in Highly Dynamic Networks

**Philipp Bamberger**

University of Freiburg, Georges-Köhler-Allee 106, 79110 Freiburg, Germany  
philipp.bamberger@cs.uni-freiburg.de

**Fabian Kuhn**

University of Freiburg, Georges-Köhler-Allee 106, 79110 Freiburg, Germany  
kuhn@cs.uni-freiburg.de

**Yannic Maus**

University of Freiburg, Georges-Köhler-Allee 106, 79110 Freiburg, Germany  
yannic.maus@cs.uni-freiburg.de

---

## Abstract

We define a generalization of local distributed graph problems to (synchronous round-based) dynamic networks and present a framework for developing algorithms for these problems. We require two properties from our algorithms: (1) They should satisfy non-trivial guarantees in every round. The guarantees should be stronger the more stable the graph has been during the last few rounds and they coincide with the definition of the static graph problem if no topological change appeared recently. (2) If a constant neighborhood around some part of the graph is stable during an interval, the algorithms quickly converge to a solution for this part of the graph that remains unchanged throughout the interval.

We demonstrate our generic framework with two classic distributed graph, namely (*degree+1*)-*vertex coloring* and *maximal independent set (MIS)*.

**2012 ACM Subject Classification** Networks → Network algorithms

**Keywords and phrases** dynamic networks, distributed graph algorithms, MIS, vertex coloring

**Digital Object Identifier** 10.4230/LIPIcs.DISC.2018.42

**Funding** Second and third author were supported by ERC Grant No. 336495 (ACDC).

## 1 Introduction

Many modern computer systems are built on top of large-scale networks such as the Internet, the world wide web, wireless ad hoc and sensor networks, or peer-to-peer networks. Often, the network topology of such systems is inherently dynamic: nodes can join or leave at any time and (e.g., in the context of overlay networks or mobile wireless networks) communication links might appear and disappear constantly. As a consequence, we aim to develop distributed algorithms that can cope with a potentially *highly dynamic network topology* and to understand what can and what cannot be computed in a dynamic network. In particular, we investigate techniques to develop distributed dynamic network solutions for *distributed graph problems* and more specifically for solving *local distributed graph problems* such as computing a graph coloring or a maximal independent set (MIS) of the network graph (see, e.g., [1, 5, 6]).

Most previous work on solving distributed graph problems in the dynamic setting is of the following flavor [3, 2]: After one or more topology changes, the algorithm has a *recovery period* to fix its output and the network does not undergo any changes during this recovery period. If further dynamic changes occur while recovering from a previous change such an



© Philipp Bamberger, Fabian Kuhn, and Yannic Maus;  
licensed under Creative Commons License CC-BY

32nd International Symposium on Distributed Computing (DISC 2018).

Editors: Ulrich Schmid and Josef Widder; Article No. 42; pp. 42:1–42:4

Leibniz International Proceedings in Informatics



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

algorithm loses its guarantees and it might even fail to provide any guarantees at all. We therefore follow a different approach. Our randomized algorithms constantly adapt to a changing environment. They always satisfy non-trivial guarantees, no matter how dynamic the network is. The guarantees become stronger if the network is less dynamic. In particular, if the network becomes static in a constant neighborhood around some part of the network, the solution of that part also converges to a static solution after a short time.

In the present paper, we develop a framework to build distributed dynamic network algorithms with the aforementioned properties and apply the framework to two of the classic distributed graph problems, namely, the problem of computing a maximal independent set (MIS) and the problem of computing a vertex coloring of the network graph. We however note that the general framework also applies to various additional graph problems. For example, it seems particularly suitable to convert classic covering or packing optimization problems to the dynamic setting. Examples for such problems are minimum dominating set, minimum vertex cover, or maximum matching. For the coloring problem, our algorithm guarantees that after two nodes are joined by an edge, they can only have the same color for a short time. Further, the total number of colors used is still essentially upper bounded by the maximum degree of the network as in the static version of the problem. In the context of dynamic networks, the degree of some node  $v$  at a time  $t$  is defined to be the number of distinct neighbors  $v$  has had during the last few rounds.

## 2 Model, Contribution & Techniques

We model a dynamic network as a synchronous system over a set  $V$  of  $n$  potential nodes. Time is divided into rounds and in each round  $r = 0, 1, 2, \dots$ , there is a communication graph  $G_r = (V_r, E_r)$ . We generally assume that nodes can wake up gradually, however for the purpose of this summary, we assume that all nodes wake up initially and we thus have  $V_r = V$  for all  $r \geq 1$ . We consider graph problems that can be decomposed into two parts that are given by a *packing* and a *covering* graph property. Essentially, a packing property is a graph property that remains true when removing edges and a covering property is a graph property that remains true when adding edges. In addition, we assume that the validity of a solution can be checked locally, i.e., by evaluating it in the constant neighborhood of every node [4]. For example, the problem of finding an MIS on a graph  $G$  can be decomposed into the problem of finding a subset  $S$  of the nodes such that no two neighbors are in  $S$  (packing property) and  $S$  is a dominating set of  $G$  (covering property). For the (degree+1)-coloring problem, the requirement that the vertex coloring is proper is a packing property and the requirement that the color of a node  $v$  is from  $\{1, \dots, \deg(v) + 1\}$  is a covering property. For a given graph problem and an integer parameter  $T \geq 1$ , we say that a given solution is a *T-dynamic solution* at time  $r$  if a) the solution satisfies the packing property for the *intersection graph*  $G_r^{T \cap} = G_{r-T+1} \cap G_{r-T+1} \cap \dots \cap G_r$  (i.e., the graph that contains all edges that have been present throughout the last  $T$  rounds), and b) the solution satisfies the covering property for the *union graph*  $G_r^{T \cup} = G_{r-T+1} \cup G_{r-T+1} \cup \dots \cup G_r$  (i.e., the graph that contains all edges that have been present at least once in the last  $T$  rounds). We believe that this provides a natural generalization of a static graph problem to the dynamic context. Note that the dynamic guarantees are stronger the less dynamic the graph is and if the graph has been static during rounds  $r - T + 1, \dots, r$ , a  $T$ -dynamic solution at time  $r$  is the same as a static solution for the given graph problem for the graph  $G_r$  in round  $r$ .

When designing a distributed algorithm for a given dynamic graph problem, we require that for some  $T \geq 1$ , the algorithm outputs a  $T$ -dynamic solution after each round  $r$ . Assume

that we can construct an algorithm  $\mathcal{A}$  such that if all nodes start  $\mathcal{A}$  in round 1, after round  $T$ ,  $\mathcal{A}$  outputs a  $T$ -dynamic solution w.r.t. to the first  $T$  graphs (i.e., a solution that satisfies the packing property for  $G_T^{T\cap}$  and the covering property for  $G_T^{T\cup}$ ). Given such an algorithm  $\mathcal{A}$ , we can in principle design an algorithm that always outputs a  $T$ -dynamic solution by just starting a new instance of  $\mathcal{A}$  in every round and outputting the solution of an instance started in round  $r + 1$  after round  $r + T$ . However, such a solution would not be satisfactory because especially if  $\mathcal{A}$  is randomized, the output might change completely from round to round even if the graph is only mildly dynamic or even static. Thus we refine this approach and define two *abstract* types of algorithms to deal with dynamic graph problems.

For two positive integers  $T$  and  $\alpha$ , we say that an algorithm  $\mathcal{A}_1$  is a  $(T, \alpha)$ -*network-static algorithm* for a given dynamic graph problem if it satisfies the following properties. At the end of each round  $r \geq 1$ , the algorithm outputs a valid *partial* solution for the graph  $G_r$  (In a partial solution, nodes are allowed to output  $\perp$  and for each node  $v$  that outputs a value  $\neq \perp$ , there is an extension of the partial solution such that the packing property for  $v$  is satisfied and the covering property for  $v$  is satisfied for all extensions of the partial solution). In addition, if the  $\alpha$ -neighborhood of some node  $v$  remains static in some interval  $[r, r_2]$ ,  $v$  must output a fixed value  $\neq \perp$  throughout the interval  $[r + T, r_2]$ . Further, for a positive integer  $T$ , we say that an algorithm  $\mathcal{A}_2$  is a  $T$ -*dynamic algorithm* for a given dynamic graph problem if it satisfies the following property. Let  $r \geq 1$  be some round and assume that we are given a valid partial solution for  $G_r$ . If  $\mathcal{A}_2$  is started in round  $r + 1$ , at the end of round  $r + T - 1$ , it outputs a  $T$ -dynamic solution that extends the given partial solution for  $G_r$ . The following theorem shows that a  $T_1$ -dynamic algorithm and a  $(T_2, \alpha)$ -network-static algorithm can be combined to obtain a distributed algorithm that always outputs a  $T_1$ -dynamic solution while behaving well if the graph is locally static for sufficiently long. Our framework thus allows to separate the two tasks of (1) always outputting a  $T$ -dynamic solution and (2) providing a locally stable output if the network is locally static.

► **Theorem 1.** *Let  $T_1$  and  $T_2$  be positive integers,  $\mathcal{P}$  a packing, and  $\mathcal{C}$  a covering problem. Given a  $T_1$ -dynamic algorithm and a  $(T_2, \alpha)$ -network-static algorithm for  $(\mathcal{P}, \mathcal{C})$ , one can combine both algorithms to an algorithm such that:*

1. (*dynamic solution*) *Its output in round  $r$  is a  $T_1$ -dynamic solution for  $(\mathcal{P}, \mathcal{C})$ .*
2. (*locally static*) *If the graph is static in the  $\alpha$ -neighborhood of a node  $v \in V_r$  in all rounds in an interval  $[r, r_2]$ , then the output of  $v$  does not change for all rounds in  $[r + T_1 + T_2, r_2]$ .*

### 3 Two Sample Problems: MIS & Vertex-Coloring

We show how to apply the above framework to two of the classic local symmetry breaking problems: computing a vertex coloring and computing an MIS of the network graph. In both cases, we only slightly adapt existing randomized algorithms (e.g., [5, 1, 6]) to obtain the results. We see the relatively simple adaptation – compared to a huge and heavy machinery – of existing static algorithms to the dynamic case as a strength of the framework in terms of practicability. Of course, some of the existing proofs need additional care and some algorithms, e.g., the MIS algorithm by Ghaffari [5], need some (crucial) modifications to assure termination in the dynamic setting.

► **Corollary 2.** *There is a  $T = O(\log n)$  and an algorithm that, w.h.p., outputs a  $T$ -dynamic solution for (degree+1)-coloring (for MIS) in every round and the output of any node  $v$  is static in all rounds in the interval  $[r + 2T, r_2]$  if the 2-neighborhood of  $v$  is static in all rounds in the interval  $[r, r_2]$ .*

---

**References**

---

- 1 N. Alon, L. Babai, and A. Itai. A fast and simple randomized parallel algorithm for the maximal independent set problem. *J. of Algorithms*, 7(4):567–583, 1986.
- 2 S. Assadi, K. Onak, B. Schieber, and S. Solomon. Fully dynamic maximal independent set with sublinear update time. In *Proc. 50th ACM Symp. on Theory of Comp. (STOC)*, 2018.
- 3 K. Censor-Hillel, E. Haramaty, and Z.S. Karnin. Optimal dynamic distributed MIS. In *Proc. 35th ACM Symp. on Principles of Distr. Computing (PODC)*, pages 217–226, 2016.
- 4 P. Fraigniaud, A. Korman, and D. Peleg. Local distributed decision. In *Proc. 52nd Symp. on Foundations of Computer Sc. (FOCS)*, pages 708–717. IEEE Computer Society, 2011.
- 5 M. Ghaffari. An improved distributed algorithm for maximal independent set. In *Proc. 27th ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 270–277, 2016.
- 6 M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM J. Comp.*, 15:1036–1053, 1986.



# Brief Announcement: Randomized Blind Radio Networks

**Artur Czumaj**

University of Warwick, Coventry, UK

A.Czumaj@warwick.ac.uk

**Peter Davies**

University of Warwick, Coventry, UK

P.W.Davies@warwick.ac.uk

---

## Abstract

Radio networks are a long-studied model for distributed system of devices which communicate wirelessly. When these devices are mobile or have limited capabilities, the system is best modeled by the ad-hoc variant, in which the devices do not know the structure of the network. Much work has been devoted to designing algorithms for the ad-hoc model, particularly for fundamental communications tasks such as broadcasting. Most of these algorithms, however, assume that devices have some network knowledge (usually bounds on the number of nodes in the network  $n$ , and the diameter  $D$ ), which may not be realistic in systems with weak devices or gradual deployment. Little is known about what can be done without this information.

This is the issue we address in this work, by presenting the first *randomized* broadcasting algorithms for *blind* networks in which nodes have no prior knowledge whatsoever. We demonstrate that lack of parameter knowledge can be overcome at only a small increase in running time. Specifically, we show that in networks without collision detection, broadcast can be achieved in  $O(D \log \frac{n}{D} \log^2 \log \frac{n}{D} + \log^2 n)$  time, almost reaching the  $\Omega(D \log \frac{n}{D} + \log^2 n)$  lower bound. We also give an even faster algorithm for directed networks with collision detection.

**2012 ACM Subject Classification** Theory of computation → Distributed algorithms, Networks → Network algorithms

**Keywords and phrases** Broadcasting, Randomized Algorithms, Radio Networks

**Digital Object Identifier** 10.4230/LIPIcs.DISC.2018.43

**Funding** Research partially supported by the Centre for Discrete Mathematics and its Applications (DIMAP), by EPSRC award EP/D063191/1, and by EPSRC award EP/N011163/1.

## 1 Model and problem

In the *ad-hoc multi-hop radio network* model, a communications network is represented as a graph, with nodes corresponding to devices with wireless capability. A directed edge  $(u, v)$  in the graph means that device  $u$  can reach device  $v$  via direct transmission. Efficiency of algorithms is measured in terms of number of nodes  $n$  in the network, and eccentricity  $D$  (maximum distance between any pair of nodes). The defining feature of radio networks is the rule for how nodes can communicate: time is divided into discrete synchronous steps, in which each node can choose whether to transmit a message or listen for messages. A listening node in a given time-step then hears a message iff exactly one of its in-neighbors transmits. In the model *with collision detection*, a listening node can distinguish between the cases of having 0 in-neighbors transmit and having more than one, but in the model *without collision detection* these scenarios are indistinguishable.



© Artur Czumaj and Peter Davies;

licensed under Creative Commons License CC-BY

32nd International Symposium on Distributed Computing (DISC 2018).

Editors: Ulrich Schmid and Josef Widder; Article No. 43; pp. 43:1–43:3

Leibniz International Proceedings in Informatics



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

While, in the ad-hoc model, the underlying graph is unknown to the nodes, it is usual to assume that nodes do know the values of  $n$  and  $D$ . We do not make this assumption, and thus are dealing with a more restrictive model, which we call *blind* radio networks, in which nodes have no prior network knowledge whatsoever.

We design *randomized algorithms* for the task of *broadcasting*, in which a single designated *source* node starts with a message, and must inform all nodes in the network via transmissions. We assume that all nodes except the source begin in an *inactive* state, and become *active* when they receive a transmission. Our algorithms are Monte-Carlo algorithms succeeding with *high probability* (i.e., their failure probability is at most  $n^{-c}$  for some  $c > 0$ ).

## 1.1 Related work

Broadcasting is possibly the most studied problem in radio networks, and has a wealth of literature in various settings. In networks without collision detection, optimal broadcasting was achieved by Czumaj and Rytter [3], and Kowalski and Pelc [5], who gave randomized algorithms that complete the task in  $O(D \log \frac{n}{D} + \log^2 n)$  time with high probability. This matched a known  $\Omega(D \log \frac{n}{D} + \log^2 n)$  lower bound for the task [1, 6]. However, their algorithms *intrinsically require parameter knowledge*, and algorithms that do not require such knowledge have been little studied. The closest analogue in the literature is the work of Jurdzinski and Stachowiak [4], who give algorithms for wake-up in single-hop radio networks under a wide range of node knowledge assumptions. Their Use-Factorial-Representation algorithm is the most relevant; the running time is given as  $O((\log n \log \log n)^3)$  for single-hop networks, but a similar analysis as we present would demonstrate that the algorithm also performs broadcasting in multi-hop networks in  $O((D + \log n) \log^2 \frac{n}{D} \log^3 \log \frac{n}{D})$  time.

## 1.2 New results

We present a randomized algorithm for broadcasting in blind (directed or undirected) networks without collision detection which succeeds with high probability within time  $O(D \log \frac{n}{D} \log^2 \log \frac{n}{D} + \log^2 n)$ . This improves over the running time of [4] and comes within a poly-log log factor of the  $\Omega(D \log \frac{n}{D} + \log^2 n)$  lower bound [1, 6]. We also present an  $O(D \log \frac{n}{D} \log \log \log \frac{n}{D} + \log^2 n)$ -time algorithm for directed networks with collision detection.

## 2 Algorithms

The main idea of our randomized algorithms in blind radio networks is as follows: when considering a particular node  $v$  we wish to inform, all of its active in-neighbors will be transmitting with some probability. We wish to make the sum of these probabilities approximately constant (say  $\frac{1}{2}$ ), since then we can show that  $v$  will be informed with good probability. However, we do not know the size of  $v$ 's active in-neighborhood, so choosing appropriate probabilities is difficult. To do so, we have the source node generate a global random variable from some distribution  $Y$  for each time-step, which will function as a 'guess' of in-neighborhood size. By appending these variables to the source message, we can ensure that all active nodes are aware of them. Then, based on these global variables and upon local randomness, the active nodes decide whether to transmit.

By choosing and analyzing the distribution  $Y$  we can obtain some bound on the probability that a node with active neighbors is informed, in each time-step. We then show a recipe for converting these probabilities to a running time for broadcasting.

**Algorithm 1** Broadcast Framework.

---

```

for  $t = 1$  to  $\infty$  do
  let  $T = 2^t$ .
  for each  $j \in [T]$ ,  $s$  generates a random variable  $x_j$  from distribution  $Y$ .
   $s$  appends variables  $x_j$  to the source message.
  for  $j$  from 1 to  $T$ , in time-step  $j$ , do active nodes  $v$  transmit with probability  $2^{-x_j}$ .
end for

```

---

**2.1 Blind radio networks without collision detection**

In networks without collision detection, we take  $Y$  to be the sum of two *components*, which account for different network conditions. Under most circumstances, we use **General-Broadcast**; where the source “guesses” a neighborhood size from 1 to  $\infty$  in each time-step, with a probability that decreases in neighborhood size in order to converge. In low diameter networks, we improve upon this with **Shallow-Broadcast** component, which quickly informs networks of low diameter using  $T$  to approximate in-neighborhood size.

► **Theorem 1.** *Broadcasting can be performed in networks without collision detection in  $O(D \log \frac{n}{D} \log^2 \log \frac{n}{D} + \log^2 n)$  time, succeeding with high probability.*

**2.2 Directed blind radio networks with collision detection**

When collision detection (and a global clock) is available, nodes can learn their exact distance  $d_v$  from the source node within  $O(D)$  time, via a process known as *beep waves* [2]. The local transmission probabilities that nodes use during our broadcasting algorithm can then depend on  $d_v$ , as well as  $T$  and the global randomness provided by the source. We add two new components to the two already defined which exploit this: **Deep-Broadcast**, which quickly informs nodes far from the source, and **Semi-Shallow-Broadcast**, which removes a running-time bottleneck when  $D$  is small.

► **Theorem 2.** *Broadcasting can be performed in networks with collision detection in  $O(D \log \frac{n}{D} \log \log \log \frac{n}{D} + \log^2 n)$  time, succeeding with high probability.*

**References**

- 1 N. Alon, A. Bar-Noy, N. Linial, and D. Peleg. A lower bound for radio broadcast. *Journal of Computer and System Sciences*, 43(2):290–298, 1991.
- 2 A. Czumaj and P. Davies. Communicating with beeps. In *Proceedings of the 19th International Conference on Principles of Distributed Systems (OPODIS)*, pages 1–16, 2015.
- 3 A. Czumaj and W. Rytter. Broadcasting algorithms in radio networks with unknown topology. In *Proceedings of the 44th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 492–501, 2003.
- 4 T. Jurdziński and G. Stachowiak. Probabilistic algorithms for the wakeup problem in single-hop radio networks. *Theory of Computing Systems*, 38(3):347–367, 2005.
- 5 D. Kowalski and A. Pelc. Broadcasting in undirected ad hoc radio networks. *Distributed Computing*, 18(1):43–57, 2005.
- 6 E. Kushilevitz and Y. Mansour. An  $\Omega(D \log(N/D))$  lower bound for broadcast in radio networks. *SIAM Journal on Computing*, 27(3):702–712, 1998.



# Brief Announcement: Deterministic Contention Resolution on a Shared Channel

**Gianluca De Marco**

Dipartimento di Informatica, University of Salerno, Italy  
demarco@dia.unisa.it

**Dariusz R. Kowalski**

Department of Computer Science, University of Liverpool, UK  
d.kowalski@liverpool.ac.uk

**Grzegorz Stachowiak**

Institute of Computer Science, University Wrocław, Poland  
gst@cs.uni.wroc.pl

---

## Abstract

A shared channel, also called multiple-access channel, is one of the fundamental communication models. Autonomous entities communicate over a shared medium, and one of the main challenges is how to efficiently resolve collisions occurring when more than one entity attempts to access the channel at the same time. In this work we explore the impact of asynchrony, knowledge (or linear estimate) of the number of contenders, and acknowledgments, on both latency and channel utilization for the Contention resolution problem with non-adaptive deterministic algorithms.

**2012 ACM Subject Classification** Theory of computation → Distributed algorithms

**Keywords and phrases** Shared channel, multiple-access channel, distributed algorithm

**Digital Object Identifier** 10.4230/LIPIcs.DISC.2018.44

## 1 Introduction

The formal model that is taken as the basis for theoretical studies is defined as follows, *cf.* the surveys by Gallager [9] and Chlebus [4] and the recent works [5, 7, 8]. A set of  $k$  stations, also called *nodes*, are connected to the same transmission medium (called a *shared channel*) and can communicate by transmitting and receiving messages on the shared channel in synchronous *rounds*. The stations have distinct ids in the range  $[N] = \{0, 1, \dots, N - 1\}$ . A *contention resolution* algorithm is a distributed algorithm that schedules the transmissions for each of the  $k$  stations possessing a packet, guaranteeing that every station eventually transmits individually (*i.e.*, without interfering with other stations) on the channel.

All the literature on this problem (with the exception of recent papers [3, 7, 8]) either assumed the (simplified) *static* situation in which the  $k$  stations are all activated at the very beginning (and therefore start simultaneously their transmitting schedules) [1, 11, 12, 13] or that the activation times are restricted to statistical or adversarial-queuing models [2, 6, 10]. In the dynamic scenario, considered in this paper, Bender et al. [3] gave a very efficient randomized adaptive algorithm with collision detection. In contrast, our work deals with deterministic non-adaptive algorithms, *i.e.*, protocols that are not using any channel feedback.

Inspired by the inherently decentralized nature of the multiple access model, and adopting the model from [8] developed in the context of randomized solutions, in this paper we focus on a more general *dynamic* scenario, in which the stations with packets could get awake (*i.e.*, start their local executions) in *arbitrary times*, *i.e.*, the sequence of activation times, also called a *wake-up schedule*, is totally determined by a worst-case adversary. This scenario,



© Gianluca De Marco, Dariusz R. Kowalski, and Grzegorz Stachowiak;  
licensed under Creative Commons License CC-BY

32nd International Symposium on Distributed Computing (DISC 2018).

Editors: Ulrich Schmid and Josef Widder; Article No. 44; pp. 44:1–44:3

Leibniz International Proceedings in Informatics



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

also called asynchronous, reflects the more realistic situation in which the stations are geographically far apart or totally independent (from themselves and/or from the scheduler which injects packets to the underlying communication protocol), and consequently each activation time is locally determined and cannot be known or predicted.

Although the communication is in synchronous rounds, we assume *no global clock* and *no system-based synchronization*: each station starts its local clock in the round when it wakes up, without knowing anything about the round numbers of other stations' clock. In the static model there is no distinction between the model with a global clock and that without it. Indeed, one can assume that a global clock is always available in the latter: all stations start simultaneously and therefore their clocks, will always tick the same rounds. In this sense, the dynamic model considered in this work is more general and challenging than the static one.

We measure the efficiency of a station in terms of its *latency*, *i.e.*, the number of rounds necessary for the station to transmit its packet successfully, measured since its activation time. The complexity of an algorithm, called an *algorithm latency* (or simply a *latency* if it is understood from the context) is defined as the maximum latency over all awoken stations. A *channel utilization*, sometimes called a *throughput*, is defined as the worst-case ratio between the contention size  $k$  (which corresponds to the absolute minimum number of rounds needed for all the awoken stations to transmit successfully to the channel) and the algorithm latency.

## 2 Our contribution

Our first result shows that if the number of contenders  $k$  (or a linear upper bound of it) is known and the stations switch-off after the acknowledgment of their successful transmissions, the channel admits efficient solutions: there exists a deterministic non-adaptive distributed algorithm working in  $O(k \log k \log N)$  time. This is close to the known lower bound  $\Omega(k \log(N/k))$ . In terms of channel utilization, the algorithm achieves throughput  $\Omega(1/(\log k \log N))$ , which is close to the upper bound  $O(1/\log(N/k))$ .

In a nutshell, we first generate a randomized schedule that succeeds with very high probability and then we use the probabilistic method to show that a schedule allowing every station to transmit must exist. Since we know  $k$ , the schedule for any station can be organized in such a way that we start with a probability of transmission  $O(1/k)$  and double it every  $O(k \log N)$  rounds until the station transmits with constant probability. For any fixed station  $v$ , the rounds  $t$  at which  $v$  has a good (constant) probability of successful transmission are those with the sum of all transmission probabilities at  $t$  being a constant. The main challenge is to show that there are sufficiently many rounds with such a favorable property, in order to get a sufficiently high probability of short latency allowing derandomization.

In the same settings but when  $k$  is unknown, we show an  $\Omega(k^2/\log k)$  lower bound, which points out that the ignorance of contention  $k$  makes the channel nearly quadratically less efficient, even if the stations could switch-off after acknowledgments. In very broad terms, the proof is organized as follows. We start by defining a randomly generated wake-up pattern for the  $k$  stations. Then we prove that in such a (worst-case) random pattern no station is able to successfully transmit after  $\Omega(k^2/\log k)$  rounds with a high probability. The probabilistic method is finally used to show that such a wake-up pattern exists.

In our final result we nearly match the above mentioned complexity (for unknown  $k$ ) by presenting an upper bound of  $O(k^2 \log N)$ , which is achieved even if acknowledgments are not provided. In terms of channel utilization, the algorithm achieves throughput  $\Omega(1/(k \log N))$ , which is close to the upper bound  $O((\log k)/k)$ . The high level approach follows the lines of the first result. Here the additional challenge is the ignorance of parameter  $k$ , which

complicates the design of the random schedule. In particular, in this case the transmission probabilities cannot depend on the unknown  $k$ . Thus, we let them start from a constant value and decrease (contrary to what we did when  $k$  was known). One of the main issues is the right choice of the decrease factor. On one hand it should be relatively fast in order to guarantee the sought latency. On the other hand, it cannot be too fast since avoiding collisions becomes harder in absence of switch-off's (due to no acknowledgements). We found out that starting from a constant probability and decreasing it every  $O(\ln N)$  rounds from  $1/\sqrt{j}$  to  $1/\sqrt{j+1}$  for  $j = 4, 5, 6, \dots$ , allows us to balance both challenges.

Surprisingly, our results imply that the knowledge of the contention size has an exponential impact on the deterministic utilization of an asynchronous channel, while it is known that for synchronized channels this feature does not influence asymptotically the channel utilization. The second implication concerns the impact of acknowledgments – our results exponentially improve deterministic channel utilization if (some estimate of)  $k$  is known, unlike the case of randomized algorithms where the corresponding improvement is only polynomial.

---

## References

- 1 A. Fernández Anta, M. A. Mosteiro, and J. Ramon Mu noz. Unbounded contention resolution in multiple-access channels. *Algorithmica*, 67:295–314, 2013.
- 2 M. A. Bender, M. Farach-Colton, S. He, B. C. Kuszmaul, and C. E. Leiserson. Adversarial contention resolution for simple channels. In *Proceedings, 17th ACM Symp. on Parallel Algorithms (SPAA)*, pages 325–332, New York, NY, USA, 2005. ACM.
- 3 M. A. Bender, T. Kopelowitz, S. Pettie, and M. Young. Contention resolution with log-logstar channel accesses. In *Proceedings, 48th ACM Symp. on Theory of Computing (STOC)*, pages 499–508, Cambridge, MA, USA, 2016. ACM.
- 4 B. S. Chlebus. Randomized communication in radio networks. In P. M. Pardalos, S. Rajasekaran, J. H. Reif, and J. D. P. Rolim, editors, *Handbook on Randomized Computing*, pages 401–456. Springer, New York, NY, USA, 2001.
- 5 B. S. Chlebus, G. De Marco, and D. R. Kowalski. Scalable wake-up of multi-channel single-hop radio networks. *Theor. Comput. Sci.*, 615:23–44, 2016. doi:10.1016/j.tcs.2015.11.046.
- 6 B. S. Chlebus, D. R. Kowalski, and M. A. Rokicki. Adversarial queuing on the multiple access channel. *ACM Trans. on Algorithms*, 8:5:1–5:31, 2012.
- 7 G. De Marco and D. R. Kowalski. Contention resolution in a non-synchronized multiple access channel. *Theor. Comput. Sci.*, 689:1–13, 2017. doi:10.1016/j.tcs.2017.05.014.
- 8 G. De Marco and G. Stachowiak. Asynchronous shared channel. In *Proceedings, ACM Symp. on Principles of Distributed Computing, PODC 2017, Washington, DC, USA, July 25-27, 2017*, pages 391–400, 2017. doi:10.1145/3087801.3087831.
- 9 Robert G. Gallager. A perspective on multiaccess channels. *IEEE Trans. Information Theory*, 31(2):124–142, 1985.
- 10 L. A. Goldberg, P. D. MacKenzie, M. Paterson, and A. Srinivasan. Contention resolution with constant expected delay. *J. ACM*, 47(6):1048–1096, 2000.
- 11 A. G. Greenberg and A. S. Winograd. Lower bound on the time needed in the worst case to resolve conflicts deterministically in multiple access channels. *J. ACM*, 32:589–596, 1985.
- 12 J. Komlós and A. G. Greenberg. An asymptotically optimal nonadaptive algorithm for conflict resolution in multiple-access channels. *IEEE Trans. on Information Theory*, 31:302–306, 1985.
- 13 D. Kowalski. On selection problem in radio networks. In *Proceedings, 24th ACM Symp. on Principles of Distributed Computing (PODC)*, pages 158–166, Las Vegas, NV, USA, 2005. ACM.





# Brief Announcement: Generalising Concurrent Correctness to Weak Memory

Simon Doherty<sup>1</sup>

Department of Computer Science, University of Sheffield, UK  
s.doherty@sheffield.ac.uk

Brijesh Dongol<sup>2</sup>

Department of Computer Science, University of Surrey, Guildford, UK

Heike Wehrheim

Department of Computer Science, Paderborn University, Paderborn, Germany

John Derrick

Department of Computer Science, University of Sheffield, UK

---

## Abstract

Correctness conditions like linearizability and opacity describe some form of atomicity imposed on concurrent objects. In this paper, we propose a correctness condition (called causal atomicity) for concurrent objects executing in a weak memory model, where the histories of the objects in question are partially ordered. We establish compositionality and abstraction results for causal atomicity and develop an associated refinement-based proof technique.

**2012 ACM Subject Classification** Theory of computation → Concurrency, Theory of computation → Shared memory algorithms

**Keywords and phrases** Weak Memory, Concurrent Object, Execution Structure

**Digital Object Identifier** 10.4230/LIPIcs.DISC.2018.45

## 1 Foundations

Correctness conditions like *linearizability* for data structures and *opacity* for software transactional memory (STM) specify atomicity for concurrent objects. They are developed under the assumption that the underlying memory model is sequentially consistent. Our objective here is to generalise such notions to weak memory models. We develop a new notion of correctness: *causal atomicity*<sup>3</sup>, which we show is compositional and ensures client abstraction. Causal atomicity applies to both concurrent objects and transactional memory, thus it encompasses both linearizability and opacity. Causal linearizability is covered earlier [1]; here we specialise causal atomicity to *causal opacity* and develop a proof technique for it.

Our framework is based on object executions given as *execution structures* [4]. They describe the usual precedence order between events as well as a communication relation. Informally, a communication relation between two events arises when some low-level operation of the first event synchronises with some operation belonging to the second event. Events describe e.g. transactions in an STM or operations on a data structure, and their

---

<sup>1</sup> Simon Doherty and John Derrick are funded by EPSRC Grant EP/M017044/1.

<sup>2</sup> Funded by EPSRC Grant EP/R019045/1.

<sup>3</sup> This notion is related to, but different from causal atomicity defined by Farzan and Madhusudan [3]. For example, their notion is not compositional.



implementations give arise to the execution of low-level operations. In prior work [1], we have used this technique to define a causal version of linearizability. Here, we generalise the methods and define a notion of atomicity that is also applicable to transactional memory.

► **Definition 1.** An *execution structure* is a tuple  $(E, \rightarrow, \dashrightarrow)$  consisting of a finite set of events  $E$ , a strict order  $\rightarrow \subseteq E \times E$  and a relation  $\dashrightarrow \subseteq E \times E$  satisfying:

1. If  $t_1 \rightarrow t_2$ , then both  $t_1 \dashrightarrow t_2$  and  $\neg(t_2 \dashrightarrow t_1)$ .
2. If  $t_1 \rightarrow t_2 \dashrightarrow t_3$  or  $t_1 \dashrightarrow t_2 \rightarrow t_3$ , then  $t_1 \dashrightarrow t_3$ .
3. If  $t_1 \rightarrow t_2 \dashrightarrow t_3 \rightarrow t_4$ , then  $t_1 \rightarrow t_4$ .

Like other concurrent correctness conditions, we employ some sequential object on a fixed alphabet to compare the executions of the concurrent object against. A sequential object  $\mathbb{S}$  specifies a set of *legal* sequential executions denoted  $legal_{\mathbb{S}}$ . Syntactically, each element of  $legal_{\mathbb{S}}$  is a sequence of events that are labelled with operations that the object provides.

## 2 Contributions

**Causal atomicity.** When comparing concurrent executions against sequential ones, some key orders of the concurrent execution need to be preserved. In our case, these are the relations  $\rightarrow$  and  $\dashrightarrow$  of the execution structure. We say  $<$  is a *logical order* of an execution structure  $\mathbb{E} = (E, \rightarrow, \dashrightarrow)$  iff  $< \subseteq E \times E$  is a strict order such that  $\rightarrow \subseteq < \subseteq \dashrightarrow$ . For a partial order  $< \subseteq E \times E$ , we let  $LE(<) = \{w \in E^* \mid < \subseteq \dashrightarrow_w\}$  be the set of *linear extensions* of  $<$ , where  $\dashrightarrow_w$  is the total order corresponding to the (total) order on the elements of  $w$ .

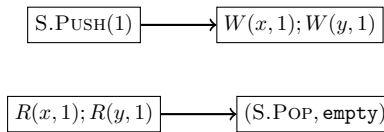
► **Definition 2.** Let  $\mathbb{S}$  be a sequential object. An execution structure  $\mathbb{E}$  is *causally atomic* w.r.t.  $\mathbb{S}$  iff there exists a logical order  $<$  of  $\mathbb{E}$  such that  $LE(<) \subseteq legal_{\mathbb{S}}$ .

The logical order at least needs to contain the *precedence order*  $\rightarrow$  and at most the *communication relation*  $\dashrightarrow$ . Note that the logical order (induced by a specification) can introduce additional communication in an implementation (see example below).

**Compositionality.** A key requirement on every correctness condition is *compositionality*: when a program employs operations from two different concurrent objects, these objects individually satisfy the correctness condition if, and only if, their combined usage also satisfies the correctness condition. Technically, we use a composition operator  $\otimes$  on sequential objects to compute the interleavings of sequential executions, and an operator  $|$  to restrict execution structures to events from a given alphabet.

► **Theorem 3.** Let  $\mathbb{S}_1$  and  $\mathbb{S}_2$  be sequential objects with disjoint alphabets  $\Sigma_1$  and  $\Sigma_2$  and let  $\mathbb{E} = (E, \rightarrow, \dashrightarrow)$  be an execution structure. Then  $\mathbb{E}|_{\Sigma_1}$  and  $\mathbb{E}|_{\Sigma_2}$  are causally atomic w.r.t.  $\mathbb{S}_1$  and  $\mathbb{S}_2$ , respectively, iff  $\mathbb{E}$  is causally atomic w.r.t.  $\mathbb{S}_1 \otimes \mathbb{S}_2$ .

For example, consider the execution structure below, which comprises a stack object  $S$  and an STM, where we assume memory values are initialised to 0. The event  $W(x, 1); W(y, 1)$  corresponds to a transaction consisting of a write to  $x$  followed by a write to  $y$ . Similarly,  $R(x, 1); R(y, 1)$  corresponds to a transaction comprising a read of  $x$  followed by a read of  $y$ .



The execution should not be allowed since there is no total ordering of events that respects the existing precedence order  $\rightarrow$ . Causal atomicity necessitates a communication relation  $W(x, 1); W(y, 1) \dashrightarrow R(x, 1); R(y, 1)$  which, together with axiom **A3** induces order

S.PUSH(1)  $\rightarrow$  (S.POP, empty). Thus, when considering the full execution structure the composition execution restricted to the stack object is not causally atomic. A more detailed example composition of concurrent objects is given in [1].

**Abstraction.** Another property of causal atomicity is *abstraction*, which formalises a notion of substitutability: when a client uses the operations that an object provides in its interface, then it should not be able to distinguish the implementation from its sequential specification. To formalise this, we represent a client  $\mathcal{C}$  as a set of *client executions*, each of which is simply a partial order over events labelled with operations from the alphabet of the object and other client-local events such as reads and writes to client variables. Suppose  $\mathbb{O}$  is a set of execution structures over  $\Sigma$ . The *client-object composition* of  $\mathcal{C}$  and  $\mathbb{O}$  is the set  $\mathcal{C}[\mathbb{O}] = \{\prec \in \mathcal{C} \mid \exists ((\text{dom}(\prec) \cup \text{ran}(\prec)) \cap \Sigma, \rightarrow, \dashrightarrow) \in \mathbb{O}. \prec|_{\Sigma} \subseteq \rightarrow\}$ .

Given a sequential object  $\mathbb{S}$ , we let  $CA[\mathbb{S}]$  be the set of execution structures that are causally atomic w.r.t.  $\mathbb{S}$ . Furthermore, we lift sequential objects to sets of execution structures by letting  $E[\mathbb{S}]$  be the set of execution structures such that there is some  $w \in \text{legal}_{\mathbb{S}}$  where *both* the precedence order and communication relation is the total order  $\rightarrow_w$ . Our goal is to compare, for any client  $\mathcal{C}$ , the client-object composition  $\mathcal{C}[CA[\mathbb{S}]]$  with the corresponding composition with the sequential object,  $\mathcal{C}[E[\mathbb{S}]]$ . To do so, we define a notion of *observational refinement*, denoted  $\sqsubseteq$ , such that  $\mathcal{C}[CA[\mathbb{S}]] \sqsubseteq \mathcal{C}[E[\mathbb{S}]]$  holds if for every execution in  $\mathcal{C}[CA[\mathbb{S}]]$ , there is an *observationally equivalent* execution in  $\mathcal{C}[E[\mathbb{S}]]$ . Our notion of observational equivalence requires that any pair of equivalent executions must have compatible orders when restricted to client-local events. We then prove the following.

► **Theorem 4.** *If  $\mathcal{C}$  is a client and  $\mathbb{S}$  a sequential object, then  $\mathcal{C}[CA[\mathbb{S}]] \sqsubseteq \mathcal{C}[E[\mathbb{S}]]$ .*

**Transactional Memory.** We use causal atomicity to obtain a correctness condition for transactional memory, which we call *causal opacity*. To do so, we first define a *transactional sequential object*, denoted  $\mathbb{T}$ , whose alphabet is made up of entire *blocks* of reads and writes, and whose semantics requires that each block executes atomically. Causal opacity itself is a condition on transactional operations rather than atomic blocks, thus allowing concurrent transactions. It is defined in terms a transformation called  $\mu$ -*abstraction* [4], that combines these operations into a block from the alphabet of  $\mathbb{T}$ . An execution structure is said to be causally opaque whenever its  $\mu$ -abstraction is causally atomic w.r.t.  $\mathbb{T}$ .

We show how to verify causal opacity using a sequential object,  $\mathbb{C}$ , such that the  $\mu$ -abstraction of every execution structure in  $CA[\mathbb{C}]$  is causally atomic w.r.t.  $\mathbb{T}$ . The object  $\mathbb{C}$  is adapted from TMS2 [2], and like its predecessor, is given in an operational fashion which enables simulation-based refinement proofs.

---

## References

- 1 S. Doherty, B. Dongol, H. Wehrheim, and J. Derrick. Making Linearizability Compositional for Partially Ordered Executions. In *iFM*, volume 11023 of *LNCS*, 2018.
- 2 S. Doherty, L. Groves, V. Luchangco, and M. Moir. Towards formally specifying and verifying transactional memory. *Formal Asp. Comput.*, 25(5):769–799, 2013.
- 3 A. Farzan and P. Madhusudan. Causal atomicity. In *CAV*, volume 4144 of *LNCS*, pages 315–328. Springer, 2006.
- 4 L. Lamport. On interprocess communication. part I: basic formalism. *Distributed Computing*, 1(2):77–85, 1986.



# Brief Announcement: Exact Size Counting in Uniform Population Protocols in Nearly Logarithmic Time

**David Doty**

Department of Computer Science, University of California, Davis  
doty@ucdavis.edu

**Mahsa Eftekhari**

Department of Computer Science, University of California, Davis  
mhseftekhari@ucdavis.edu

**Othon Michail**

Department of Computer Science, University of Liverpool, UK  
Othon.Michail@liverpool.ac.uk

**Paul G. Spirakis**

Department of Computer Science, University of Liverpool, UK and Computer Technology Institute & Press “Diophantus” (CTI), Patras, Greece  
P.Spirakis@liverpool.ac.uk

**Michail Theofilatos**

Department of Computer Science, University of Liverpool, UK  
michail.theofilatos@liverpool.ac.uk

---

## Abstract

We study population protocols: networks of anonymous agents whose pairwise interactions are chosen uniformly at random. The *size counting problem* is that of calculating the exact number  $n$  of agents in the population, assuming no leader (each agent starts in the same state). We give the first protocol that solves this problem in sublinear time.

The protocol converges in  $O(\log n \log \log n)$  time and uses  $O(n^{60})$  states ( $O(1) + 60 \log n$  bits of memory per agent) with probability  $1 - O(\frac{\log \log n}{n})$ . The time to converge is also  $O(\log n \log \log n)$  in expectation. Crucially, unlike most published protocols with  $\omega(1)$  states, our protocol is *uniform*: it uses the same transition algorithm for any population size, so does not need an estimate of the population size to be embedded into the algorithm.

**2012 ACM Subject Classification** Theory of computation → Distributed algorithms

**Keywords and phrases** population protocol, counting, leader election, polylogarithmic time

**Digital Object Identifier** 10.4230/LIPIcs.DISC.2018.46

**Funding** DD, ME: NSF grant CCF-1619343. OM, PS, MT: EEE/CS initiative NeST. MT: Leverhulme Research Centre for Functional Materials Design.

## 1 Introduction

*Population protocols* [4] are networks that consist of computational entities called *agents* with no control over the schedule of interactions with other agents. In a population of  $n$  agents, repeatedly a random pair of agents is chosen to interact, each observing the state of the other agent before updating its own state.



© David Doty, Mahsa Eftekhari, Othon Michail, Paul G. Spirakis, and Michail Theofilatos; licensed under Creative Commons License CC-BY

32nd International Symposium on Distributed Computing (DISC 2018).

Editors: Ulrich Schmid and Josef Widder; Article No. 46; pp. 46:1–46:3

Leibniz International Proceedings in Informatics



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

The (*parallel*) time for some event to happen in a protocol is a random variable, defined as the number of interactions, divided by  $n$ , until the event happens. A recent blitz of impressive results in population protocol has shown that leader election [1, 9, 7, 8] and exact majority [3, 2] can be solved in  $\text{polylog}(n)$  time using  $\text{polylog}(n)$  states. Most of the protocols with  $\omega(1)$  states use a *nonuniform* model: given  $n$ , the state set  $Q_n$  and transition function  $\delta_n : Q_n \times Q_n \rightarrow Q_n \times Q_n$  are allowed to depend arbitrarily on  $n$ , other than the constraint that  $|Q_n| \leq f(n)$  for some function  $f$  growing as  $\text{polylog}(n)$ . This nonuniformity is used in most of the cited protocols to encode a value such as  $\lfloor \log n \rfloor$  into each agent.

We define a *uniform* variant of the model: the same transition algorithm is used for all populations, though the number of states may vary with the population size. A uniform protocol can be deployed into *any* population without knowing in advance the size, or even a rough estimate of the size. The original,  $O(1)$ -state model [4, 5, 6], is uniform since there is a single transition function. Because we allow memory to grow with  $n$ , our model's power exceeds that of the original, but is strictly less than that of the nonuniform model of most papers using  $\omega(1)$  states.

## 2 Algorithm

The problem of counting the number of agents and storing this number in each agent is clearly solvable by an  $O(n)$  time protocol using a straightforward leader election: Agents initially assume they are leaders and the count is 1. When two leaders meet, one agent sums their counts while the other becomes a follower, and followers propagate by epidemic the maximum count. No faster protocol was previously known. Our main result improves this.

► **Theorem 2.1.** *There is a leaderless, uniform population protocol solving the exact size counting problem with probability 1. With probability at least  $1 - \frac{10+5 \log \log n}{n}$ , the convergence time is at most  $6 \ln n \log \log n$ , and each agent uses  $17 + 60 \log n$  bits of memory. The expected time to convergence is at most  $7 \ln n \log \log n$ .*

Key to our technique is a protocol, due to Mocquard et al. [10] (and similar to that of Alistarh and Gelashvili [3]), that counts the exact difference between the number  $b$  of “blue” and  $r$  of “red” agents in the initial population. The protocol assumes that each agent initially stores  $n$  exactly (so is nonuniform). Blue agents start with an integer value  $-M$ , while red agents start with  $M$ . When two agents meet, they average their values, one rounding up and the other down if the sum is odd. This eventually converges to all agents sharing the population-wide average  $(b - r) \frac{M}{n}$ , and the estimates of this average get close enough for the output to be correct within  $O(\log n)$  time [10]. Our protocol essentially inverts this, starting with one blue agent (a leader) and  $n - 1$  red agents, we compute the population size as a function of the average. (See below for details.) However, for this to work, our protocol requires a leader and for each agent to share a value  $M \geq 3n^3$ , which are not present initially. Four sub-protocols are used in total (although all agents run in parallel, each subprotocol runs sequentially within each agent whenever it interacts): UNIQUEID, ELECTLEADER, AVERAGING, and TIMER.

UNIQUEID eventually assigns to every agent a unique ID, represented as a binary string. Agents start with ID  $\epsilon$  (empty string), and whenever two agents with the same ID meet, all agents double the length of their IDs with uniformly random bits (appending a single bit when two  $\epsilon$ 's meet). This protocol requires  $\Omega(n)$  time to converge, but within only  $O(\log n \log \log n)$  time can be used by the next subprotocol to elect a leader.

ELECTLEADER propagates the lexicographically largest ID (considered the ID of the leader) by epidemic (via transition of the form  $x, y \rightarrow y, y$  if  $y > x$  lexicographically). The length of the leader's ID is used as a polynomial-factor upper bound on  $3n^3$ .



AVERAGING uses a fast averaging protocol [10, 3]. We assume the initial configuration of this protocol is one leader and  $n - 1$  followers. (This protocol and the next (TIMER) are restarted each time the UNIQUEID protocol discovers two agents shared an ID; so eventually AVERAGING will be restarted with a unique leader.) Each agent stores the value  $M$ , and the leader initializes an integer field `ave` to be  $M$ , with followers initializing `ave` to be 0. When two agents meet, they average their `ave` fields, with one rounding up and the other rounding down if the sum is odd. Thus the population-wide sum is always  $M$ . Eventually all agents have `ave` =  $\lceil \frac{M}{n} \rceil$  or  $\lfloor \frac{M}{n} \rfloor$ , so  $n = \lfloor \frac{M}{\text{ave}} + \frac{1}{2} \rfloor$  (i.e.,  $\frac{M}{\text{ave}} + \frac{1}{2}$  rounded to the nearest integer). It could take linear time for `ave` to converge this closely to  $\frac{M}{n}$ , but as long as  $M \geq 3n^3$  and `ave` is within  $n$  of  $\frac{M}{n}$ ,  $\lfloor \frac{M}{\text{ave}} + \frac{1}{2} \rfloor$  is the correct population size  $n$ ; we show that in  $O(\log n)$  time all `ave` fields are within  $n$  of  $\frac{M}{n}$ .

Since UNIQUEID continues restarting beyond the  $O(\log n \log \log n)$  time required for initialize convergence to a correct output, TIMER is used to detect when AVERAGING has likely converged, waiting to write output into the `output` field of the agent. Timer is a phase clock [6] that ensures after the correct value is written, on subsequent restarts of AVERAGING, the incorrect values that exist before AVERAGING re-converges will not overwrite the correct value recorded into `output` during the earlier restart.

### 3 Conclusion

$\Omega(n)$  is a clear lower bound on the number of states required for any protocol computing the exact population size, since  $\log n$  bits are required merely to write the number  $n$ . (Note that our protocol uses  $60 \log n$  bits.) It is an open question if there exists a uniform polylog-time,  $O(n)$ -state population protocol for exact size computation.

---

#### References

- 1 Dan Alistarh, James Aspnes, David Eisenstat, Rati Gelashvili, and Ronald L Rivest. Time-space trade-offs in population protocols. In *SODA*, 2017.
- 2 Dan Alistarh, James Aspnes, and Rati Gelashvili. Space-optimal majority in population protocols. In *SODA*, 2018.
- 3 Dan Alistarh and Rati Gelashvili. Polylogarithmic-time leader election in population protocols. In *ICALP*, 2015.
- 4 Dana Angluin, James Aspnes, Zoë Diamadi, Michael J. Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed Computing*, 18(4):235–253, 2006.
- 5 Dana Angluin, James Aspnes, and David Eisenstat. Stably computable predicates are semilinear. In *PODC*, 2006.
- 6 Dana Angluin, James Aspnes, and David Eisenstat. Fast computation by population protocols with a leader. *Distributed Computing*, 21(3):183–199, 2008.
- 7 Petra Berenbrink, Dominik Kaaser, Peter Kling, and Lena Otterbach. Simple and Efficient Leader Election. In *SOSA*, 2018.
- 8 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 *PODC*, 2017.
- 9 Leszek Gąsieniec and Grzegorz Stachowiak. Fast space optimal leader election in population protocols. In *SODA*, 2018.
- 10 Yves Mocquard, Emmanuelle Anceaume, James Aspnes, Yann Busnel, and Bruno Sericola. Counting with population protocols. In *NCA*, 2015.




# Brief Announcement: A Tight Lower Bound for Clock Synchronization in Odd-Ary $M$ -Toroids

Reginald Frank<sup>1</sup>

Texas A&M University, College Station, TX, USA

reginaldfrank77@tamu.edu

 <https://orcid.org/0000-0002-0423-1071>

Jennifer L. Welch<sup>2</sup>

Texas A&M University, College Station, TX, USA

welch@cse.tamu.edu

---

## Abstract

In this paper we show a tight closed-form expression for the optimal clock synchronization in  $k$ -ary  $m$ -cubes with wraparound, where  $k$  is odd. This is done by proving a lower bound of  $\frac{1}{4}um(k - \frac{1}{k})$ , where  $k$  is the (odd) number of processes in each of the  $m$  dimensions, and  $u$  is the uncertainty in delay on every link. Our lower bound matches the previously known upper bound.

**2012 ACM Subject Classification** Theory of computation → Distributed algorithms

**Keywords and phrases** Clock synchronization, Lower bound,  $k$ -ary  $m$ -toroid

**Digital Object Identifier** 10.4230/LIPIcs.DISC.2018.47

**Related Version** Find the full paper at <https://arxiv.org/abs/1807.05139>.

## 1 Introduction

Synchronizing clocks in a distributed system in which processes communicate through messages with uncertain delays is subject to inherent errors. A body of work has sought bounds on how closely the clocks can be synchronized when there is no drift in the hardware clocks and there are no failures. Lundelius and Lynch [5] showed that, in an  $n$ -process clique with the same uncertainty  $u$  on every link, the best synchronization possible is  $u(1 - \frac{1}{n})$ . Subsequently, Halpern et al. [4] considered arbitrary topologies in which each link may have a different uncertainty and showed that the optimal clock synchronization is the solution of an optimization problem. This work was generalized by [1, 6] in which algorithms were given for finding the optimal clock synchronization in any given execution. In contrast to the more general lower bounds of [4, 1, 6], Biaz and Welch [3] gave a collection of *closed-form* upper and lower bounds on the optimal clock synchronization in the worst case for  $k$ -ary  $m$ -cubes ( $m$ -dimensional hypercubes with  $k$  processes in every dimension), both with and without wraparound, in which every link has the same uncertainty,  $u$ . When there is no wraparound, the tight bound is  $\frac{1}{2}um(k - 1)$ . When there is wraparound and  $k$  is even, the tight bound is  $\frac{1}{4}umk$ . However, when there is wraparound and  $k$  is odd, there is a gap between the upper bound of  $\frac{1}{4}um(k - \frac{1}{k})$  and the lower bound of  $\frac{1}{4}um(k - 1)$ .

---

<sup>1</sup> Supported in part by CRA-W and CDC's DREU program and NSF grant CNS-0540631.

<sup>2</sup> Supported in part by NSF grant 1526725.



© Reginald Frank and Jennifer L. Welch;

licensed under Creative Commons License CC-BY

32nd International Symposium on Distributed Computing (DISC 2018).

Editors: Ulrich Schmid and Josef Widder; Article No. 47; pp. 47:1–47:3

Leibniz International Proceedings in Informatics



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

In this paper, we consider  $k$ -ary  $m$ -cubes with wraparound (“ $m$ -toroids”) and odd  $k$ . We show a lower bound of  $\frac{1}{4}um(k - \frac{1}{k})$ , which matches the previously known upper bound. We use the same shifting technique from previous lower bounds for clock synchronization (e.g., [5, 4, 3]). The key insight in our improved lower bound is to exploit the fact that the graph is a collection of rings in each dimension and to use multiple shifted executions instead of one.

## 2 Preliminaries

We first present our model and problem statement (following [5, 2, 3]). We consider a graph of  $k^m$  processes, where  $k \geq 3$  is odd and  $m \geq 1$ , in which each process id is a tuple  $\langle p_0, p_1, \dots, p_{m-1} \rangle$  where each  $p_i \in \{0, 1, \dots, k-1\}$ . There are links in both directions between any two processes  $\vec{p}$  and  $\vec{q}$  if and only if their ids differ in exactly one component, say the  $i$ -th, such that  $p_i = q_i + 1$  (addition on process id components is modulo  $k$  throughout). Each process  $\vec{p}$  has a *hardware clock* modeled as a function  $H_{\vec{p}}$  from reals (real time) to reals (clock time). We assume there is no drift, so  $H_{\vec{p}}(t) = t + c_{\vec{p}}$  for some constant  $c_{\vec{p}}$ . Each *process* is modeled as a state machine whose transition function takes as input the current state, current value of the hardware clock, and current *event* (receipt of a message or some internal occurrence), and produces a new state and a message to send over each incident link.

A *history* of process  $\vec{p}$  is a sequence of alternating states and pairs of the form (event, hardware clock value), beginning with  $\vec{p}$ 's initial state. Each state must follow correctly from the previous one according to  $\vec{p}$ 's transition function and the hardware clock values must increase. A *timed history* of  $\vec{p}$  is a history together with an assignment of a real time  $t$  to each pair  $(e, T)$  in the history such that  $H_{\vec{p}}(t) = T$ . An *execution* is a set of  $k^m$  timed histories, one per process, with a bijection for each link between the set of messages sent over the link and the set of messages received over the link. The *delay* of a message is the difference between the real time when it is received and the real time when it is sent. An execution is *admissible* if every message has delay in  $[0, u]$  where  $u$  is a fixed value called the *uniform uncertainty*.

We assume each process  $\vec{p}$  has a local variable  $adj_{\vec{p}}$  as part of its state and we define its *adjusted clock*  $A_{\vec{p}}(t)$  to be equal to  $H_{\vec{p}}(t) + adj_{\vec{p}}(t)$ . An execution has *terminated* once all processes have stopped changing their *adj* variables. We say the algorithm *achieves  $\epsilon$ -synchronized clocks* if every admissible execution eventually terminates with  $|A_{\vec{p}}(t) - A_{\vec{q}}(t)| \leq \epsilon$  for all processes  $\vec{p}$  and  $\vec{q}$  and all times  $t$  after termination.

“Shifting” an execution changes the real times at which events occur [5]. Let  $\mathbf{x}$  be an  $m$ -dimensional matrix of real numbers with  $k$  elements in each dimension, which we call a *shift matrix*; elements of  $\mathbf{x}$  are indexed by process ids. Define  $shift(\alpha, \mathbf{x})$  be the result of adding  $x_{\vec{p}}$  to the real time associated with each event in  $\vec{p}$ 's timed history in  $\alpha$ . Shifting changes the hardware clocks and message delays as follows [5, 2]:

- **Lemma 1.** *Let  $\alpha$  be an execution with hardware clocks  $H_{\vec{p}}$  and let  $\mathbf{x}$  be a shift matrix. Then  $shift(\alpha, \mathbf{x})$  is a (not necessarily admissible) execution in which*
- the hardware clock of each  $\vec{p}$ , denoted  $H'_{\vec{p}}(t)$ , equals  $H_{\vec{p}}(t) - x_{\vec{p}}$  and*
  - every message from  $\vec{p}$  to  $\vec{q}$  has delay  $\delta - x_{\vec{p}} + x_{\vec{q}}$ , where  $\delta$  is the message's delay in  $\alpha$ .*

## 3 Lower Bound

► **Theorem 2.** *For any algorithm that achieves  $\epsilon$ -synchronized clocks in a  $k$ -ary  $m$ -toroid with uniform uncertainty  $u$ , where  $k$  is odd, it must be that  $\epsilon \geq \frac{1}{4}um(k - \frac{1}{k})$ .*

The complete proof and an example for the  $k = 5$  case are in the full paper.

**Proof sketch.** Let  $\mathcal{A}$  be any algorithm that achieves  $\epsilon$ -synchronized clocks in a  $k$ -ary  $m$ -toroid with uniform uncertainty  $u$ , where  $k = 2r + 1$  for some integer  $r \geq 1$ . Let  $\alpha$  be the admissible execution of  $\mathcal{A}$  in which  $H_{\vec{p}}(t) = t$  for each process  $\vec{p}$ , every message from  $\vec{p}$  to  $\vec{q}$ , where  $\vec{q}$  is  $\vec{p}$ 's neighbor in the  $h$ -th dimension such that  $q_h = p_h + 1$ , has the same fixed delay  $\delta_{\vec{p},\vec{q}}$ , which is 0 if  $0 \leq p_h < r$  and is  $u$  if  $r \leq p_h < k$ , and every message from  $\vec{q}$  to  $\vec{p}$  has the same fixed delay  $\delta_{\vec{q},\vec{p}} = u - \delta_{\vec{p},\vec{q}}$ .

For  $0 \leq i < k$ , define  $\alpha^i = \text{shift}(\alpha, \mathbf{x}^i)$ , where the  $\vec{p}$ -th element of the shift matrix  $\mathbf{x}^i$ , denoted  $x_{\vec{p}}^i$ , is defined as  $\sum_{j=0}^{m-1} \mathbf{W}_{p_j}^i$ , where  $\mathbf{W}$  is defined as follows:

range of $i \in \{0, \dots, m-1\}$			
$0 \leq i < r$		$r \leq i < k$	
range of $p_j$	$\mathbf{W}_{p_j}^i$	range of $p_j$	$\mathbf{W}_{p_j}^i$
$0 \leq p_j \leq i$	0	$0 \leq p_j \leq i - r$	$p_j u$
$i < p_j \leq r$	$(p_j - i)u$	$i - r < p_j \leq r$	$(i - r)u$
$r < p_j \leq r + i + 1$	$(r - i)u$	$r < p_j \leq i$	$(i - p_j)u$
$r + i + 1 < p_j \leq 2r$	$(2r - p_j + 1)u$	$i < p_j \leq 2r$	0

The idea behind the shift amounts in  $\mathbf{W}$  is to cause two processes that are farthest apart in the graph to be shifted as far apart in real time as possible – thus achieving a large skew between their adjusted clocks – while maintaining valid message delays between all neighbors. By considering multiple shifted executions, we can cancel out terms involving adjusted clocks, leaving behind only terms that involve the system parameters  $\epsilon$  and  $u$ , and the graph parameters  $k$  and  $m$ .

In the full paper we show that all shifted executions are admissible, i.e., that all message delays are in  $[0, u]$ :

► **Lemma 3.** *For all  $i$ ,  $0 \leq i < k$ ,  $\alpha^i$  is admissible.*

Fix any  $i$  with  $0 \leq i < r$ . We focus on two processes that are maximally far away from each other. Since  $\alpha^i$  is admissible by Lemma 3,  $\mathcal{A}$  must ensure that  $A_{\langle i, \dots, i \rangle}^i - A_{\langle i+r+1, \dots, i+r+1 \rangle}^i \leq \epsilon$ , where  $A_{\vec{p}}^i$  denotes the adjusted clock of process  $\vec{p}$  after termination in  $\alpha^i$ . By definition of  $\alpha^i$  and Lemma 1(a),  $A_{\langle i, \dots, i \rangle}^i = A_{\langle i, \dots, i \rangle}$  and  $A_{\langle i+r+1, \dots, i+r+1 \rangle}^i = A_{\langle i+r+1, \dots, i+r+1 \rangle} - m(r-i)u$ . Thus by substituting we get  $A_{\langle i, \dots, i \rangle} - A_{\langle i+r+1, \dots, i+r+1 \rangle} \leq -m(r-i)u + \epsilon$ , for  $0 \leq i < r$ . Similarly, we can show  $A_{\langle i, \dots, i \rangle} - A_{\langle i-r, \dots, i-r \rangle} \leq -m(i-r)u + \epsilon$ , for  $r \leq i < k$ .

Adding together these  $k$  inequalities and simplifying gives  $\epsilon \geq \frac{1}{4}um(k - \frac{1}{k})$ . ◀

## References

- 1 Hagit Attiya, Amir Herzberg, and Sergio Rajsbaum. Optimal clock synchronization under different delay assumptions. *SIAM J. Comput.*, 25(2):369–389, 1996.
- 2 Hagit Attiya and Jennifer L. Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics, Second Edition*. John Wiley & Sons, Hoboken, NJ, 2004.
- 3 Saad Biaz and Jennifer L. Welch. Closed form bounds for clock synchronization under simple uncertainty assumptions. *Inf. Process. Lett.*, 80(3):151–157, 2001.
- 4 Joseph Y. Halpern, Nimrod Megiddo, and Ashfaq A. Munshi. Optimal precision in the presence of uncertainty. *J. Complexity*, 1(2):170–196, 1985.
- 5 Jennifer Lundelius and Nancy Lynch. An upper and lower bound for clock synchronization. *Inform. Control*, 62(2/3):190–204, 1984.
- 6 Boaz Patt-Shamir and Sergio Rajsbaum. A theory of clock synchronization (extended abstract). In *Proc. 26th Annual ACM Symp. Theory of Comput.*, pages 810–819, 1994.



# Brief Announcement: On Simple Back-Off in Unreliable Radio Networks

**Seth Gilbert**

National University of Singapore  
seth.gilbert@comp.nus.edu.sg

**Nancy Lynch**

Massachusetts Institute of Technology, Cambridge, MA 02139, USA  
lynch@csail.mit.edu

**Calvin Newport**

Georgetown University, Washington, D.C., USA  
cnewport@cs.georgetown.edu

**Dominik Pajak**

Massachusetts Institute of Technology, Cambridge, MA 02139, USA  
pajak@csail.mit.edu

---

## Abstract

In this paper, we study local broadcast in the dual graph model, which describes communication in a radio network with both reliable and unreliable links. Existing work proved that efficient solutions to these problems are impossible in the dual graph model under standard assumptions. In real networks, however, simple back-off strategies tend to perform well for solving these basic communication tasks. We address this apparent paradox by introducing a new set of constraints to the dual graph model that better generalize the slow/fast fading behavior common in real networks. We prove that in the context of these new constraints, simple back-off strategies now provide efficient solutions to local broadcast in the dual graph model. These results provide theoretical foundations for the practical observation that simple back-off algorithms tend to work well even amid the complicated link dynamics of real radio networks.

**2012 ACM Subject Classification** Theory of computation → Distributed algorithms, Networks → Ad hoc networks

**Keywords and phrases** radio networks, broadcast, unreliable links, distributed algorithm, robustness

**Digital Object Identifier** 10.4230/LIPIcs.DISC.2018.48

**Related Version** The full version is available at <https://arxiv.org/abs/1803.02216>.

## 1 Introduction

Existing papers proved that it is impossible to solve standard broadcast problems efficiently in the dual graph model without the addition of strong extra assumptions [3]. In real radio networks, however, which suffer from the type of link dynamics abstracted by the dual graph model, simple back-off strategies tend to perform quite well.

These dueling realities seem to imply a dispiriting gap between theory and practice: basic communication tasks that are easily solved in real networks are impossible when studied in abstract models of these networks.

*What explains this paradox?* This paper tackles this fundamental question.



© Seth Gilbert, Nancy Lynch, Calvin Newport, and Dominik Pajak;  
licensed under Creative Commons License CC-BY

32nd International Symposium on Distributed Computing (DISC 2018).

Editors: Ulrich Schmid and Josef Widder; Article No. 48; pp. 48:1–48:3

Leibniz International Proceedings in Informatics



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



As detailed below, we focus our attention on the *adversary* entity that decides which unreliable links to include in the network topology in each round of an execution in the dual graph model. We introduce a new type of adversary with constraints that better generalize the dynamic behavior of real radio links. We then reexamine simple back-off strategies originally introduced in the standard radio network model (which has only reliable links) [1], and prove that for reasonable parameters, these simple strategies *now do* guarantee efficient communication in the dual graph model combined with our new, more realistic adversary.

**Dual Graph Model.** This model describes the network topology with two graphs  $G = (V, E)$  and  $G' = (V, E')$ , where  $E \subseteq E'$ . The  $n = |V|$  vertices in  $V$  correspond to the wireless devices in the network, which we call *nodes* in the following. The edge in  $E$  describe reliable links (which maintain a consistently high quality), while the edges in  $E' \setminus E$  describe unreliable links (which have quality that can vary over time). For a given dual graph, we use  $\Delta$  to describe the maximum degree in  $G'$ , and  $D$  to describe the diameter of  $G$ .

Time proceeds in synchronous rounds that we label  $1, 2, 3, \dots$ . For each round  $r \geq 1$ , the network topology is described by  $G_r = (V, E_r)$ , where  $E_r$  contains all edges in  $E$  plus a subset of the edges in  $E' \setminus E$ . The subset of edges from  $E' \setminus E$  are selected by an *adversary*. The graph  $G_r$  can be interpreted as describing the high quality links during round  $r$ . That is, if  $\{u, v\} \in E_r$ , this mean the link between  $u$  and  $v$  is strong enough that  $u$  could deliver a message to  $v$ , or garble another message being sent to  $v$  at the same time.

With the topology  $G_r$  established for the round, behavior proceeds as in the standard radio network model. That is, each node  $u \in V$  can decide to transmit or receive. If  $u$  receives and exactly one neighbor  $v$  of  $u$  in  $E_r$  transmits, then  $u$  receives  $v$ 's message. If  $u$  receives and two or more neighbors in  $E_r$  transmit,  $u$  receives nothing as the messages are lost due to collision. If  $u$  receives and no neighbor transmits,  $u$  also receives nothing. We assume  $u$  does not have collision detection, meaning it cannot distinguish between these last two cases.

**The Fading Adversary.** We parameterize the adversary with a *stability factor* that we represent with an integer  $\tau \geq 1$ . In each round, the adversary must draw the subset of edges (if any) from  $E' \setminus E$  to include in the topology from a distribution defined over these edges. The adversary selects which distributions it uses and it can change this distribution at most once every  $\tau$  rounds.

**Problem.** In this paper, we study the *local* broadcast problem. The problem assumes a set  $B \subseteq V$  of nodes are provided with a message. Let  $R \subseteq V$  be the set of nodes in  $V$  that neighbor at least one node in  $B$  in  $E$ . The problem is solved once every node in  $R$  has received at least one message from a node in  $B$ .

**Uniform Algorithms.** In this paper focus on *uniform algorithms*, which require nodes to make their probabilistic transmission decisions according to a predetermined sequence of broadcast probabilities that we express as a repeating cycle,  $(p_1, p_2, \dots, p_k)$  of  $k$  probabilities in synchrony.

**Our results.** In standard Dual Graph Model, where the adversary can arbitrarily change the state of all the unreliable edges in every step, the time of local broadcast can be lower bounded by  $\Omega(n/\log n)$  [3]. On the other hand, in reliable networks, *decay* algorithm solves local broadcast in time  $O(\log \Delta \log(n/\varepsilon))$  [1] with probability at least  $1 - \varepsilon$  and this time is optimal [2]. Thus there is an exponential gap between the reliable model and worst-case

unreliable model. Our fading adversary can be (for large  $\tau$ ) seen as an average-case unreliable model. For smaller  $\tau$  the model becomes similar to the standard dual graph model (in particular, for  $\tau = 1$  model with fading adversary is stronger than the dual graph model).

We show that for  $\tau \geq \log \Delta$ , the optimal time of local broadcast for reliable networks can be achieved in the model with fading adversary. Secondly we prove a tradeoff between the optimal time of local broadcast in the model with fading adversary and the value of  $\tau$ . We show that factor  $\Delta^{1/\tau}$  is necessary in the time complexity of any uniform local broadcast algorithm. This shows how quickly the optimal time increases between both extremes depending on  $\tau$ .

## 2 Results

Our algorithm is a simple back-off style strategy inspired by the *decay* routine from [1]. We use notation  $\bar{\tau} = \min\{\lceil \log_{2e} \Delta/2 \rceil, \tau\}$ .

<pre> 1 <b>Procedure:</b> Uniform(<math>k, p_1, p_2, \dots, p_k</math>) 2 <b>for</b> <math>i = 1, 2, \dots, k</math> <b>do</b> 3   <b>if</b> has message <b>then</b> 4       with prob. <math>p_i</math> <b>Transmit</b> <b>else Listen</b> 5   <b>else Listen</b> // without a message        listen </pre>	<pre> 1 <b>Algorithm:</b> FRLB(<math>r</math>) 2 <b>for</b> <math>i \leftarrow 1</math> <b>to</b> <math>\bar{\tau}</math> <b>do</b> <math>p_i \leftarrow \frac{\log_{2e} \Delta}{\Delta^{i/\bar{\tau}}}</math> 3 <b>repeat</b> <math>r</math> times 4     Uniform(<math>\bar{\tau}, p_1, p_2, \dots, p_{\bar{\tau}}</math>) </pre>
--	--

► **Theorem 1.** For any error bound  $\epsilon > 0$ , algorithm  $FRLB(2 \lceil \ln(n/\epsilon) \rceil \cdot \lceil 4\Delta^{1/\bar{\tau}}/\log \Delta \rceil)$  solves local broadcast in  $O\left(\frac{\Delta^{1/\bar{\tau}} \cdot \bar{\tau}^2}{\log_{2e} \Delta} \cdot \log(n/\epsilon)\right)$  rounds, with probability at least  $1 - \epsilon$ .

Notice, for  $\tau \geq \log \Delta$  this bound simplifies to  $O(\log \Delta \log(n/\epsilon))$ , matching the performance of *decay* algorithm [1] and the lower bound in the standard reliable radio network model [2]. This performance, however, degrades toward the polynomial lower bounds from the existing dual graph literature [3] as  $\tau$  reduces from  $\log \Delta$  toward a minimum value of 1. We show this degradation to be near optimal by proving that *any* local broadcast algorithm that uses a fixed sequence of broadcast probabilities requires  $\Omega(\Delta^{1/\tau}/\log \Delta)$  rounds to solve the problem with probability  $1/2$  for a given  $\tau$ . For  $\tau \in O(\log \Delta / \log \log \Delta)$ , we refine this bound further to  $\Omega(\Delta^{1/\tau} \tau^2 / \log \Delta)$ , matching our upper bound within constant factors.

► **Theorem 2.** Fix a maximum degree  $\Delta \geq 10$ , stability factor  $\tau$  and uniform local broadcast algorithm  $\mathcal{A}$ . Assume that  $\mathcal{A}$  solves local broadcast in expected time  $f(\Delta, \tau)$  in all graphs with maximum degree  $\Delta$  and fading adversary with stability  $\tau$ . It follows that:

1. if  $\tau < \ln(\Delta - 1)/(12 \log \log(\Delta - 1))$  then  $f(\Delta, \tau) \in \Omega(\Delta^{1/\tau} \tau^2 / \log \Delta)$ ,
2. if  $\tau < \ln(\Delta - 1)/16$  then  $f(\Delta, \tau) \in \Omega(\Delta^{1/\tau} \tau / \log \Delta)$ .


## References

- 1 Reuven Bar-Yehuda, Oded Goldreich, and Alon Itai. On the time-complexity of broadcast in multi-hop radio networks: An exponential gap between determinism and randomization. *J. Comput. Syst. Sci.*, 45(1):104–126, 1992.
- 2 Mohsen Ghaffari, Bernhard Haeupler, Nancy A. Lynch, and Calvin C. Newport. Bounds on contention management in radio networks. In *Distributed Computing - 26th International Symposium, DISC 2012, Salvador, Brazil, October 16-18, 2012. Proceedings*, pages 223–237, 2012.
- 3 Mohsen Ghaffari, Nancy A. Lynch, and Calvin C. Newport. The cost of radio network broadcast for different models of unreliable links. In *ACM Symposium on Principles of Distributed Computing, PODC '13, Montreal, QC, Canada*, pages 345–354, 2013.




# Brief Announcement: Fast and Scalable Group Mutual Exclusion

**Shreyas Gokhale**

The University of Texas at Dallas  
Richardson, TX 75080, USA  
shreyas.gokhale@utdallas.edu  
 <https://orcid.org/0000-0002-7589-6927>

**Neeraj Mittal**

The University of Texas at Dallas  
Richardson, TX 75080, USA  
neerajm@utdallas.edu  
 <https://orcid.org/0000-0002-8734-1400>

---

## Abstract

---

The *group mutual exclusion (GME)* problem is a generalization of the classical mutual exclusion problem in which every critical section is associated with a *type* or *session*. Critical sections belonging to the same session can execute concurrently, whereas critical sections belonging to different sessions must be executed serially. The well-known read-write mutual exclusion problem is a special case of the group mutual exclusion problem.

In a shared memory system, locks based on traditional mutual exclusion or its variants are commonly used to manage content contention among processes. In concurrent algorithms based on *fine-grained* synchronization, a single lock is used to protect access to a *small* number of shared objects (e.g., a lock for every tree node) so as to minimize contention window. Evidently, a large number of shared objects in the system would translate into a large number of locks. Also, when fine-grained synchronization is used, most lock accesses are expected to be uncontended in practice.

Most existing algorithms for the solving the GME problem have high space-complexity per lock. Further, all algorithms except for one have high step-complexity in the uncontended case. This makes them unsuitable for use in concurrent algorithms based on fine-grained synchronization. In this work, we present a novel GME algorithm for an asynchronous shared-memory system that has  $O(1)$  space-complexity per GME lock when the system contains a large number of GME locks *as well as*  $O(1)$  step-complexity when the system contains no conflicting requests.

**2012 ACM Subject Classification** Theory of computation → Concurrent algorithms

**Keywords and phrases** Group Mutual Exclusion, Fine-Grained Synchronization, Space Complexity, Contention-Free Step Complexity

**Digital Object Identifier** 10.4230/LIPIcs.DISC.2018.49

**Funding** This work was supported, in part, by the National Science Foundation (NSF) under grants numbered CNS-1115733 and CNS-1619197.

## 1 Introduction

The *group mutual exclusion (GME)* problem is a generalization of the classical mutual exclusion (ME) problem in which every critical section is associated with a *type* or *session* [7]. Critical sections belonging to the same session can execute concurrently, whereas critical sections belonging to different sessions must be executed serially. The GME problem models



© Shreyas Gokhale and Neeraj Mittal;  
licensed under Creative Commons License CC-BY  
32nd International Symposium on Distributed Computing (DISC 2018).  
Editors: Ulrich Schmid and Josef Widder; Article No. 49; pp. 49:1–49:3  
Leibniz International Proceedings in Informatics



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

situations in which a resource may be accessed at the same time by processes of the same group, but not by processes of different groups. As an example, suppose data is stored on multiple discs in a shared CD-jukebox. When a disc is loaded into the player, users that need data on that disc can access the disc concurrently, whereas users that need data on a different disc have to wait until the current disc is unloaded [7]. The well-known readers/writers problem is a special case of the group mutual exclusion problem.

In a shared memory system, locks based on traditional mutual exclusion or its variants are commonly used to manage contention among processes. A lock grants a process exclusive access to a shared object, preventing any other process from modifying the object. This makes it easier to design, analyze, implement and debug lock-based concurrent algorithms.

With the wide availability and use of multicore systems, developing concurrent data structures that scale well with the number of cores has gained increasing importance. Many of the best performing concurrent algorithms for data structures use fine-grained synchronization in which a single lock is used to protect access to a small number of shared objects so as to minimize contention window [5]. In order to perform an operation, a process typically needs to lock only a small number of objects, thereby allowing multiple processes whose operations do not conflict with each other to manipulate parts of the data structure at the same time. Typically, in such a system, most lock accesses are expected to be uncontended in practice [5]. Recently, GME-based locks have been used to improve the performance of a concurrent skip list using the notion of unrolling in which multiple key-value pairs are stored in a single node [8].

All existing algorithms for solving the GME problem have either high space-complexity of  $\Omega(n)$  per lock or high step-complexity of  $\Omega(n)$  in the uncontended case or both, where  $n$  denotes the number of processes in the system [7, 6, 2, 1, 4]. This makes them unsuitable for use in lock-based concurrent data structures that employ fine-grained synchronization to manage contention. In this work, we present a novel GME algorithm for an asynchronous shared-memory system that has  $O(1)$  space-complexity per GME lock when the system contains a large number of GME locks *as well as*  $O(1)$  step-complexity when the system contains no conflicting requests.

## 2 The Group Mutual Exclusion Algorithm

Our GME algorithm is inspired by Herlihy's universal construction for deriving a wait-free linearizable implementation of a concurrent object from its sequential specification using consensus objects [5]. Roughly speaking, the universal construction works as follows. The state of the concurrent object is represented using (i) its initial state and (ii) the sequence of operations that have applied to the object so far. The two are maintained using a singly linked list in which the first node represents the initial state and the remaining nodes represent the operations. To perform an operation, a process first creates a new node and initializes it with all the relevant details of the operation (type, input arguments, etc.). It then tries to append the node at the end of the list. To manage conflicts in case multiple processes are trying to append their own node to the list, a consensus object is used to determine which of the several nodes is chosen to be appended to the list. Specifically, every node stores a consensus object and the consensus object of the current last node is used to decide its successor (i.e., the next operation to be applied to the object). A process whose node is not selected simply tries again. A helping mechanism is used to guarantee that every process trying to perform an operation eventually succeeds in appending its node to the list.

We modify the aforementioned universal construction to derive a GME algorithm that satisfies several desirable properties. Intuitively, an operation in the universal construction corresponds to a critical section request in our GME algorithm. Appending a new node to the list thus corresponds to establishing a new session. However, unlike in the universal construction, a single session in our GME algorithm can be used to satisfy multiple critical section requests. This basically means that every critical section request does not cause a new node to be appended to the list. This requires some careful bookkeeping so that no “empty” sessions are established. Further, a simple consensus algorithm, implemented using LL/SC instructions, is used to determine the next session to be established. Helping is used to ensure that every request is eventually satisfied.

If the node for a critical section request is appended to the list, then the process that owns the node is said to enter the session as a leader; otherwise it is said to enter the session as a follower. A leader process, on leaving its critical section, relinquishes the ownership of its current node and claims the ownership of the node for the previous session instead. This enables us to bound the length of the linked list and make our algorithm space-efficient.

More details of our GME algorithm are available in [3], where we also describe a way to bound the values of all the variables used in our GME algorithm.

► **Theorem 1** (multi-lock space complexity). *The space complexity of our GME algorithm is  $O(m + n^2 + n\ell)$  space, where  $n$  denotes the number of processes,  $m$  denotes the number of GME objects and  $\ell$  denotes the maximum number of locks a process needs to hold at the same time.*

► **Theorem 2** (concurrent entering step complexity). *The maximum number of steps a process has to execute in its entry and exit sections provided no other process in the system has an outstanding conflicting request during that period is  $O(1)$ .*

Note that the above theorem also implies  $O(1)$  step complexity in contention-free case.

---

## References

- 1 V. Bhatt and C. C. Huang. Group Mutual Exclusion in  $O(\log n)$  RMR. In *Proceedings of the 29th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 45–54, JUL 2010.
- 2 R. Danek and V. Hadzilacos. Local-Spin Group Mutual Exclusion Algorithms. In *Proceedings of the 18th Symposium on Distributed Computing (DISC)*, pages 71–85, OCT 2004.
- 3 S. Gokhale and N. Mittal. Fast and Scalable Group Mutual Exclusion. Available at <http://arxiv.org/abs/1805.04819>.
- 4 Y. He, K. Gopalakrishnan, and E. Gafni. Group Mutual Exclusion in Linear Time and Space. In *Proceedings of the 17th International Conference on Distributed Computing And Networking (ICDCN)*, JAN 2016.
- 5 M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Morgan Kaufmann, 2012.
- 6 P. Jayanti, S. Petrovic, and K. Tan. Fair Group Mutual Exclusion. In *Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing (PODC)*, pages 275–284, JUL 2003.
- 7 Y.-J. Joung. Asynchronous Group Mutual Exclusion. *Distributed Computing (DC)*, 13(4):189–206, 2000.
- 8 K. Platz. *Saturation in Lock-Based Concurrent Data Structures*. PhD thesis, Department of Computer Science, The University of Texas at Dallas, 2017.





# Brief Announcement: On the Impossibility of Detecting Concurrency

**Éric Goubault**

École Polytechnique, Palaiseau, France  
eric.goubault@lix.polytechnique.fr

**Jérémy Ledent**

École Polytechnique, Palaiseau, France  
jeremy.ledent@lix.polytechnique.fr

**Samuel Mimram**

École Polytechnique, Palaiseau, France  
samuel.mimram@lix.polytechnique.fr

---

## Abstract

We identify a general principle of distributed computing: one cannot force two processes running in parallel to see each other. This principle is formally stated in the context of asynchronous processes communicating through shared objects, using trace-based semantics. We prove that it holds in a reasonable computational model, and then study the class of concurrent specifications which satisfy this property. This allows us to derive a Galois connection theorem for different variants of linearizability.

**2012 ACM Subject Classification** Theory of computation → Concurrency

**Keywords and phrases** concurrent specification, concurrent object, linearizability

**Digital Object Identifier** 10.4230/LIPIcs.DISC.2018.50

## 1 Introduction

A common setting to study distributed computing is the one of asynchronous processes communicating through shared objects. In this context, the question of how to formally specify the behavior of the shared objects arises: what we want is an abstract, high-level specification, that does not refer to a particular implementation of the object. This is easy to achieve when the objects that we consider are concurrent versions of sequential data structures, such as lists or queues. Namely, we can simply take the usual sequential specification of the object, and extend it to a concurrent setting using one of the many correctness criteria found in the literature: atomicity [8], sequential consistency [5], serializability [10], causal consistency [11], or linearizability [4]. However, we also want to be able to specify objects with an intrinsically concurrent nature, such as those found in the area of distributed computability [3]: consensus and set-agreement objects, immediate snapshot. Another example is Java's *Exchanger* object: two processes that call the *Exchanger* object concurrently can swap values. A process calling the *Exchanger* alone fails and receives an error value.

A very general way of specifying such objects was proposed by Lamport [6]. The specification of a concurrent object is simply the set of all the execution traces that we consider correct for this object. For example, a correct execution trace of the *Exchanger* object is depicted below:



© Éric Goubault and Jérémy Ledent and Samuel Mimram;  
licensed under Creative Commons License CC-BY

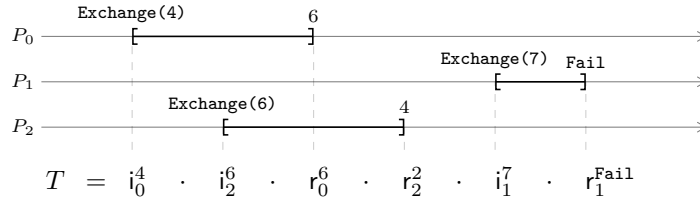
32nd International Symposium on Distributed Computing (DISC 2018).

Editors: Ulrich Schmid and Josef Widder; Article No. 50; pp. 50:1–50:4

Leibniz International Proceedings in Informatics



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



The trace  $T$  consists of *invocation* events  $i_i^x$  meaning that the object was called by process  $i$  with input  $x$ , and *response* events  $r_i^y$  meaning that process  $i$  returned with output value  $y$ . This trace can be seen as an abstraction of the real-time execution pictured above, where the horizontal axis represents global time. Formally, for a fixed set  $\mathcal{V}$  of values and  $n$  processes, the set of *actions* is:

$$\mathcal{A} = \{i_i^x \mid 0 \leq i < n \text{ and } x \in \mathcal{V}\} \cup \{r_i^y \mid 0 \leq i < n \text{ and } y \in \mathcal{V}\}$$

A *trace* is a word  $T \in \mathcal{A}^*$  such that for every process  $i$ , the projection of  $T$  on  $i$  starts with an invocation and alternates between invocations and responses. We write  $\mathcal{T}$  for the set of all traces. Then, a *concurrent specification* in the sense of [6] is simply a subset of  $\mathcal{T}$ .

This notion of concurrent specification is not convenient to use when reasoning about distributed systems. In fact, the correctness criteria such as linearizability can be regarded as more convenient ways of defining such concurrent specifications: starting from a sequential specification  $\sigma$ , we obtain  $\text{Lin}(\sigma) \subseteq \mathcal{T}$  which is the set of all the traces that are linearizable w.r.t.  $\sigma$ . The advantage of this is that sequential specifications are much easier to describe than general concurrent specifications. To specify objects with a more concurrent flavor, variants of linearizability have been described: set-linearizability [9] (a.k.a. concurrency-aware linearizability [2]) and interval-linearizability [1]. The last one is the most expressive: it captures all the distributed *tasks*, in the sense of [3].

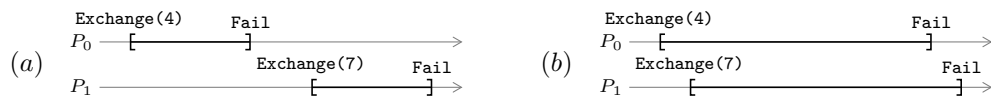
**Contribution.** In the following, we restrict to a class of concurrent specifications: those satisfying the *undetectability of concurrency* property. As it turns out, they correspond exactly to the concurrent specifications definable using interval-linearizability. We show that these are the only relevant concurrent specifications, and prove a theorem showing how the different notions of linearizability relate to this property.

## 2 Results

A concurrent specification  $\sigma \subseteq \mathcal{T}$  satisfies the *undetectability of concurrency* property if the following two conditions hold, where  $a$  is an action of some process  $j \neq i$ .

- (1) *invocations commute to the left:* if  $T \cdot a \cdot i_i^x \cdot T' \in \sigma$ , then  $T \cdot i_i^x \cdot a \cdot T' \in \sigma$ ,
- (2) *responses commute to the right:* if  $T \cdot r_i^y \cdot a \cdot T' \in \sigma$ , then  $T \cdot a \cdot r_i^y \cdot T' \in \sigma$ .

Such properties come up in Lipton’s reduction proof technique [7]: (1) and (2) assert that invocations and responses are left/right movers, respectively. Pictorially, these two properties mean that if we take a correct execution trace (a) and “expand” the intervals, then the resulting trace (b) must also be considered correct. Intuitively, in (b), the two processes failed to see each other and acted as in the sequential trace (a).



As a naive attempt at specifying the Exchanger object, we might have wanted to allow (a) and forbid (b). But implementing such a specification would have been hopeless, as we show in a reasonable trace-based computational model:

► **Theorem 1.** *The semantics  $\llbracket P \rrbracket$  of any program  $P$  satisfies properties (1) and (2).*

Intuitively, the reason why Theorem 1 holds is that calling an object or returning a value does not communicate any information to the other processes. If a process idles right after invoking, or just before returning, it is invisible to the other processes.

The undetectability of concurrency property is naturally enforced by the usual specification techniques such as linearizability, so by using these tools we do not have to worry about this property: we get it for free.

► **Proposition 2.** *Let  $\sigma$  be a sequential specification. Then  $\text{Lin}(\sigma)$ , the set of all linearizable traces, satisfies properties (1) and (2).*

We now write  $\text{SSpec}$  for the set of sequential specifications, and  $\text{CSpec}$  for the set of concurrent specifications which satisfy the undetectability of concurrency. Proposition 2 says that we can view  $\text{Lin}$  as a map from  $\text{SSpec}$  to  $\text{CSpec}$ . Conversely, there is also a map in the other direction  $\text{U} : \text{CSpec} \rightarrow \text{SSpec}$  which, given a concurrent specification, forgets about all the concurrent behaviors and keeps only the sequential ones.

► **Theorem 3.** *The functions  $\text{Lin}$  and  $\text{U}$  are monotonous w.r.t. inclusions, and form a Galois connection, i.e., for every  $\sigma \in \text{SSpec}$  and  $\tau \in \text{CSpec}$ ,  $\text{Lin}(\sigma) \subseteq \tau \iff \sigma \subseteq \text{U}(\tau)$ .*

The fact that we imposed the undetectability of concurrency property on  $\text{CSpec}$  is crucial in order to establish Theorem 3. This theorem can be understood as follows: given a sequential specification  $\sigma$ , we want to extend it to a concurrent one. Then any  $\tau \in \text{CSpec}$  that contains  $\sigma$  must also contain  $\text{Lin}(\sigma)$ , i.e.,  $\text{Lin}(\sigma)$  is the smallest extension of  $\sigma$  which is in  $\text{CSpec}$ . Thus,  $\text{Lin}(\sigma)$  can be described as follows: we start with the set of all sequential traces of  $\sigma$ , then close it under the two properties (1) and (2).

Finally, note that analogues of Proposition 2 and Theorem 3 still hold when we replace linearizability by set- or interval-linearizability. In particular, Theorem 3 for interval-linearizability gives us the following characterization of interval-linearizable objects:

► **Corollary 4.** *The concurrent specifications which are definable using interval-linearizability are exactly the ones satisfying the undetectability of concurrency.*

---

## References

- 1 Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. Specifying Concurrent Problems: Beyond Linearizability and up to Tasks. In *DISC 2015, Proceedings*, pages 420–435, 2015.
- 2 Nir Hemed, Noam Rinetzky, and Viktor Vafeiadis. Modular Verification of Concurrency-Aware Linearizability. In *DISC 2015, Proceedings*, pages 371–387, 2015.
- 3 Maurice Herlihy, Dmitry Kozlov, and Sergio Rajsbaum. *Distributed Computing Through Combinatorial Topology*. Morgan Kaufmann Publishers Inc., 2013.
- 4 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.
- 5 L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, 1979.

## 50:4 On the Impossibility of Detecting Concurrency

- 6 Leslie Lamport. On interprocess communication. *Distributed Computing*, 1(2):77–85, 1986.
- 7 Richard J Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, 1975.
- 8 J. Misra. Axioms for memory access in asynchronous hardware systems. In Stephen D. Brookes, Andrew William Roscoe, and Glynn Winskel, editors, *Seminar on Concurrency*, pages 96–110. Springer Berlin Heidelberg, 1985.
- 9 Gil Neiger. Set-Linearizability. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, page 396, 1994.
- 10 Christos H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, 1979.
- 11 M. Raynal, G. Thia-Kime, and M. Ahamad. From serializable to causal transactions for collaborative applications. In *Proceedings of the 23rd EUROMICRO*, pages 314–321, 1997.

# Brief Announcement: Effects of Topology Knowledge and Relay Depth on Asynchronous Consensus

**Dimitris Sakavalas**

Boston College, USA  
dimitris.sakavalas@bc.edu

**Lewis Tseng**

Boston College, USA  
lewis.tseng@bc.edu

**Nitin H. Vaidya**<sup>1</sup>

Georgetown University, USA  
nitin.vaidya@georgetown.edu

---

## Abstract

Consider an asynchronous incomplete directed network. We study the feasibility and efficiency of *approximate crash-tolerant* consensus under different restrictions on topology knowledge and relay depth, i.e., the maximum number of hops any message can be relayed.

**2012 ACM Subject Classification** Computer systems organization → Fault-tolerant network topologies

**Keywords and phrases** Asynchrony, crash fault, consensus, topology knowledge, relay

**Digital Object Identifier** 10.4230/LIPIcs.DISC.2018.51

**Related Version** A full version is available at [5], <https://arxiv.org/abs/1803.04513>.

## 1 Introduction

The fault-tolerant consensus problem introduced by Lamport et al. [4] and its variations have been studied extensively. The need to overcome the FLP impossibility result for consensus in asynchronous systems has led to the study of the *approximate consensus* problem [3], where nodes are required to output roughly the same value. We consider a directed network of  $n$  nodes, wherein at most  $f$  nodes are subject to crash failure. We explore the feasibility and efficiency of achieving approximate consensus in *asynchronous incomplete* networks under different restrictions on *topology knowledge* and *relay depth* (defined as the maximum number of hops that information can be propagated). These constraints are useful in large-scale networks to avoid memory overload and network congestion.

Our prior work [7] showed that exact crash-tolerant consensus is solvable in *synchronous* networks with only one-hop knowledge and relay depth 1, i.e., each node only needs to know its immediate neighbors, and no message needs to be relayed. Such a local algorithm is of practical interest due to low deployment cost and message complexity in each round. In *asynchronous* undirected networks, there exists a simple flooding-based algorithm adapted from [2] that achieves approximate consensus with up to  $f$  crash faults if the network satisfies  $(f + 1)$

---

<sup>1</sup> This research is supported in part by National Science Foundation awards 1421918. Any opinions, findings, and conclusions or recommendations expressed here are those of the authors and do not necessarily reflect the views of the funding agencies or the U.S. government.



node-connectivity and  $n > 2f$ , where  $n$  is the number of nodes. However, the sufficiency of the conditions is not guaranteed if we restrict topology knowledge and relay depth. Motivated by this observation, this work addresses the following question in *asynchronous* systems:

*What is a tight condition on the underlying communication graph to achieve approximate consensus if each node has only a  $k$ -hop topology knowledge and relay depth  $k'$ ?*

To the best of our knowledge, two prior papers [1, 6] examined a similar problem – *synchronous* Byzantine consensus. In [6], Su and Vaidya identified the condition under different relay depths. Alchieri et al. [1] studied the problem under unknown participants. The technique developed for asynchronous consensus in this work is significantly different. Please refer to our technical report [5] for more discussion on other related work.

**Model and Terminology.** The point-to-point message-passing network is represented by *directed* graph  $G(\mathcal{V}, \mathcal{E})$ , where  $\mathcal{V}$  is the set of  $n$  nodes, and  $\mathcal{E}$  is the set of directed edges. The communication links are assumed to be reliable. Node  $i$  can transmit messages to its outgoing neighbors and itself. Up to  $f$  nodes may suffer crash failures in an execution, in which case they stop taking steps. We consider *asynchronous* communication. i.e., a message may be delayed arbitrarily but will eventually be delivered. Let  $N_i^-, N_i^+$  denote the sets of incoming neighbors and outgoing neighbors of node  $i$  respectively. Also, for a node  $i$ , its  *$k$ -hop incoming neighbors*  $N_i^-(k)$ , are defined as the nodes  $j$  which can reach  $i$  using a directed path in  $G$  that has  $\leq k$  hops. The notion of  *$k$ -hop outgoing neighbors*  $N_i^+(k)$ , is defined similarly. For set  $B \subseteq \mathcal{V}$ , node  $i$  is said to be an incoming neighbor of set  $B$  if  $i \notin B$ , and there exists  $j \in B$  such that  $(i, j) \in \mathcal{E}$ . With  $N_{\bar{B}}^-$  we will denote the incoming neighbors of  $B$ .

**Approximate Consensus.** In the approximate consensus problem [3], each node  $i$  maintains a *state*  $v_i$  with  $v_i[p]$  denoting the state of node  $i$  at the end of phase (or iteration)  $p$ . The initial state of node  $i$ ,  $v_i[0]$ , is equal to the initial input provided to node  $i$ . At the start of *asynchronous* phase  $p$  ( $p > 0$ ), the state of node  $i$  is  $v_i[p-1]$ . Let  $U[p]$  and  $\mu[p]$  be the maximum and the minimum state at nodes that have not crashed by the end of phase  $p$ . Then, a *correct* approximate consensus algorithm needs to satisfy the following two conditions for any  $\epsilon > 0$ :

- *Validity*:  $\forall p > 0, U[p] \leq U[0]$  and  $\mu[p] \geq \mu[0]$ ; and
- $\epsilon$ -*Convergence*:  $\exists p, \forall r \geq p, U[r] - \mu[r] < \epsilon$ .

## 2 Limited Topology Knowledge and Relay Depth

Prior works (e.g., [7]) assumed that each node has  $n$ -hop topology knowledge and relay depth  $n$ , which is not realistic in large-scale networks. Hence, we are interested in the family of algorithms (*iterative  $k$ -hop algorithms*) in which nodes only know their  $k$ -hop neighborhoods, and propagate state values to nodes that are at most  $k$ -hops away for  $1 \leq k \leq n$ . Note that no exchange of topology information takes place.

**Iterative  $k$ -hop algorithms.** Each node  $i$  performs the following three steps in phase  $p$ :

1. *Transmit*: Transmit messages of the form  $(v_i[p-1], \cdot)$  to nodes that are reachable from node  $i$  via at most  $k$  hops away, through intermediate relays.
2. *Receive*: Receive messages from all  $k$ -hop incoming neighbors. Denote by  $R_i[p]$  the set of messages that node  $i$  received at phase  $p$ .
3. *Update*: Update state using a transition function  $Z_i$ , where  $Z_i$  is a part of the specification of the algorithm, and takes as input the set  $R_i[t]$ . i.e.,  $v_i[t] := Z_i(R_i[t], v_i[t-1])$  at node  $i$ .

**Main Results.** Below, we present two definitions to facilitate the discussion.

► **Definition 1** ( $A \rightarrow_k B$ ). Given disjoint non-empty subsets of nodes  $A$  and  $B$ , we will say that  $A \rightarrow_k B$  holds if there exists a node  $i$  in  $B$  for which there exist at least  $f+1$  node-disjoint paths of length at most  $k$  from distinct nodes in  $A$  to  $i$ . More formally, if  $\mathcal{P}_i^A(k)$  is the family of all sets of  $k$ -length node-disjoint paths (with  $i$  being their only common node) initiating in  $A$  and ending in node  $i$ ,  $A \rightarrow_k B$  means that  $\exists i \in B, \max_{P \in \mathcal{P}_i^A(k)} |P| \geq f+1$ .

► **Definition 2** (Condition  $k$ -CCA). For any partition  $L, C, R$  of  $\mathcal{V}$ , where  $L$  and  $R$  are both non-empty, either  $L \cup C \rightarrow_k R$  or  $R \cup C \rightarrow_k L$ .

► **Theorem 3.** *Approximate crash-tolerant consensus in an asynchronous system using iterative  $k$ -hop algorithms is feasible iff  $G$  satisfies Condition  $k$ -CCA.*

The complete proof is presented in [5]. We only sketch the proof here. The necessity is proved using an indistinguishable argument inspired by [3, 7]. For sufficiency, we present Algorithm  $k$ -LocWA. Our key contribution is to identify what are the set of messages that each node needs to receive before updating its state value in Step 3 of the iterative  $k$ -hop algorithms. Algorithm  $k$ -LocWA relies on *Condition  $k$ -WAIT*: For  $F_i \subseteq N_i^-(k)$ , we denote with  $reach_i^k(F_i)$  the set of nodes that have paths of length  $l \leq k$  to node  $i$  in  $G_{V-F_i}$ . That is, the set of  $k$ -hop incoming neighbors of  $i$  that remain connected with  $i$  even when all nodes in set  $F_i$  crash. The condition is satisfied at node  $i$ , in phase  $p$  if there exists  $F_i \subseteq N_i^-(k)$  with  $|F_i[p]| \leq f$  such that  $reach_i^k(F_i[p]) \subseteq heard_i[p]$ . Finally, we show that if  $G$  satisfies Condition  $k$ -CCA, then Algorithm  $k$ -LocWA correctly solves approximate consensus.

We derive an upper bound on the number of *asynchronous* phases needed for  $\epsilon$ -convergence of Algorithm  $k$ -LocWA in [5]. This upper bound is naturally a function of values  $\epsilon, k, f, n$  and  $\delta = U[0] - \mu[0]$ . As a function of  $k$ , the bound implies that for  $k' \geq k$ , Algorithm  $k'$ -LocWA  $\epsilon$ -converges faster than  $k$ -LocWA. We also prove that for values  $k, k' \in \mathbb{N}$  with  $k \leq k'$ , Condition  $k$ -CCA implies Condition  $k'$ -CCA and that  $n$ -CCA is equivalent to CCA from [7].

**Topology Discovery and Unlimited Relay Depth.** Even if the topology knowledge of the nodes is restricted to their 1-hop neighborhood, we show that allowing topology information exchange and relay depth  $n$ , one can achieve approximate consensus whenever condition CCA [7] holds. This can be achieved through Algorithm LWA, presented in the full version [5], which introduces a topology discovery mechanism to learn the crucial topology information that is necessary to achieve consensus. This result implies that knowledge of topology does not affect the feasibility of the problem if topology knowledge can be relayed.

---

## References

- 1 Eduardo A. P. Alchieri, Alysson Neves Bessani, Joni da Silva Fraga, and Fabíola Greve. Byzantine consensus with unknown participants. In *OPODIS 2008*, volume 5401 of *LNCS*, pages 22–40. Springer, 2008. doi:10.1007/978-3-540-92221-6\_4.
- 2 Danny Dolev. The Byzantine generals strike again. *Journal of Algorithms*, 3(1), 1982.
- 3 Danny Dolev, Nancy A. Lynch, Shlomit S. Pinter, Eugene W. Stark, and William E. Weihl. Reaching approximate agreement in the presence of faults. *J. ACM*, 33(3):499–516, 1986. doi:10.1145/5925.5931.
- 4 M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980. doi:10.1145/322186.322188.
- 5 Dimitris Sakavalas, Lewis Tseng, and Nitin H. Vaidya. Effects of topology knowledge and relay depth on asynchronous consensus. *CoRR*, abs/1803.04513, 2018. arXiv:1803.04513.



## 51:4 Topology Knowledge, Relay Depth, and Asynchronous Consensus

- 6 Lili Su and Nitin Vaidya. Reaching approximate Byzantine consensus with multi-hop communication. In *SSS 2015*, volume 9212 of *LNCS*, pages 21–35. Springer, 2015. doi:10.1007/978-3-319-21741-3\_2.
- 7 Lewis Tseng and Nitin H. Vaidya. Fault-tolerant consensus in directed graphs. In *PODC '15*, pages 451–460. ACM, 2015. doi:10.1145/2767386.2767399.

# Brief Announcement: Loosely-stabilizing Leader Election with Polylogarithmic Convergence Time

**Yuichi Sudo**

Graduate School of Information Science and Technology, Osaka University, Japan  
y-sudou@ist.osaka-u.ac.jp

**Fukuhito Ooshita**

Graduate School of Science and Technology, Nara Institute of Science and Technology, Japan  
f-ooshita@is.naist.jp

**Hirotsugu Kakugawa**

Graduate School of Information Science and Technology, Osaka University, Japan  
kakugawa@ist.osaka-u.ac.jp

**Toshimitsu Masuzawa**

Graduate School of Information Science and Technology, Osaka University, Japan  
masuzawa@ist.osaka-u.ac.jp

---

## Abstract

We present a fast loosely-stabilizing leader election protocol in the population protocol model. It elects a unique leader in a poly-logarithmic time and holds the leader for a polynomial time with arbitrarily large degree in terms of parallel time, i.e., the number of steps per the population size.

**2012 ACM Subject Classification** Theory of computation → Self-organization

**Keywords and phrases** Self-stabilization, Loose-stabilization, Population protocols

**Digital Object Identifier** 10.4230/LIPIcs.DISC.2018.52

**Acknowledgements** This work was supported by JSPS KAKENHI Grant Numbers 17K19977, 16K00018, 18K11167, and 18K18000 and Japan Science and Technology Agency(JST) SICORP.

## 1 Introduction

We consider the *population protocol* (PP) model [1] in this paper. A network called *population* consists of a large number of finite-state automata, called *agents*. Agents often make *interactions*, each between a pair of agents to communicate with, by which agents update their states. As with the majority of studies on population protocols, we consider only the population of complete graphs and the uniformly-random scheduler, which selects an interacting pair of agents at each step uniformly at random.

We focus on Self-Stabilizing Leader Election (SS-LE) problem, which is one of the most important and well-studied problems in the PP model. This problem requires that starting from any configuration, a population reaches a safe configuration in which exactly one leader exists; and after that, the population keeps that leader forever. Unfortunately, it is well known that no protocol solves SS-LE unless every agent knows the exact size  $n$  of the population (i.e., the number of agents). To circumvent this impossibility, our previous work [3] introduced the concept of *loose-stabilization*, a relaxed variant of self-stabilization: Starting from any initial configuration, the population must reach a safe configuration within a short time; after that, the specification of the problem must be sustained for a sufficiently long time, though not necessarily forever. This previous work gave a loosely-stabilizing leader



© Yuichi Sudo, Fukuhito Ooshita, Hirotsugu Kakugawa, and Toshimitsu Masuzawa;  
licensed under Creative Commons License CC-BY

32nd International Symposium on Distributed Computing (DISC 2018).

Editors: Ulrich Schmid and Josef Widder; Article No. 52; pp. 52:1–52:3

Leibniz International Proceedings in Informatics



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

■ **Table 1** LS-LE protocols in the PP model. Time complexities are presented in parallel time.

	Convergence Time	Holding Time	Agent Memory
Sudo et al. [3]	$O(N \log N)$	$\Omega(e^N)$	$O(\log N)$
Izumi [2]	$O(N)$	$\Omega(e^N)$	$O(\log N)$
Proposed Protocol ( $P_{PL}$ )	$O(c \log^3 N)$	$\Omega(cn^{10c})$	$O(\log \log N)$

**Algorithm 1**  $P_{PL}$ : specifying a state-transition at interaction between agents  $a_0$  and  $a_1$ .

---

```

1:  $a_0.\text{timer}_P \leftarrow a_1.\text{timer}_P \leftarrow \max(a_0.\text{timer}_P - 1, a_1.\text{timer}_P - 1, 0)$ 
2: for  $i \in \{0, 1\}$  such that  $a_i.\text{timer}_P = 0$  do  $a_i.\text{leader} \leftarrow \top$  endfor
3: if  $\exists i \in \{0, 1\} : a_i.\text{leader} = \top$  then  $a_0.\text{timer}_P \leftarrow a_1.\text{timer}_P \leftarrow t_{\max}$  endif
4:  $a_0.\text{virus} \leftarrow a_1.\text{virus} \leftarrow \max(a_0.\text{virus} - 1, a_1.\text{virus} - 1, 0)$ 
5: for  $i \in \{0, 1\}$  such that  $\neg a_i.\text{shield} \wedge (a_i.\text{virus} > 0)$  do  $a_i.\text{leader} \leftarrow \perp$  endfor
6: for  $i \in \{0, 1\}$  do  $a_i.\text{timer}_L \leftarrow \max(a_i.\text{timer}_L - 1, 0)$  endfor
7: if  $a_0.\text{timer}_L = 0 \wedge a_0.\text{leader} = \top$  then
8:    $a_0.\text{virus} \leftarrow t_{\text{virus}}$ 
9:    $a_0.\text{shield} \leftarrow \top$ 
10: end if
11: if  $a_1.\text{timer}_L = 0 \wedge a_1.\text{leader} = \top$  then  $a_1.\text{shield} \leftarrow \perp$  endif
12: for  $i \in \{0, 1\}$  such that  $a_i.\text{timer}_L = 0$  do  $a_i.\text{timer}_L \leftarrow t_{\text{emit}}$  endfor

```

---

election (LS-LE) protocol assuming that every agent knows only a common upper bound  $N$  of  $n$ . This protocol is practically equivalent to an SS-LE protocol since it maintains the unique leader for exponential time in  $n$  after reaching a safe configuration within  $O(N \log N)$  parallel time, i.e., the number of steps (interactions) per the population size  $n$ . Recently, Izumi [2] presented a method to improve the convergence time of this protocol to  $O(N)$  parallel time. He also proved the optimality of its convergence time by showing that any LS-LE protocol whose holding time is exponential requires  $\Omega(N)$  parallel time for convergence.

In this paper, we break through the barrier of this linear lower bound of convergence time and achieve poly-logarithmic parallel convergence time. Given a parameter  $c \geq 1$  and an upper bound  $N$  of  $n$ , our protocol converges to a safe configuration in  $O(c \log^3 N)$  time, and preserves the unique leader for  $\Omega(cn^{10c})$  time thereafter (Table 1). Owing to the above impossibility result by [2], the holding time of our protocol is no longer exponential but polynomial in  $n$ . However, we can arbitrarily increase the degree of the polynomial using parameter  $c$ . For example, the holding time is  $\Omega(n^{100})$  if we assign  $c = 10$ , which is expected to be large enough in all practical situations.

## 2 Proposed Protocol

The pseudo code of  $P_{PL}$  is shown in Algorithm 1. Each agent has five variables **leader**  $\in \{\top, \perp\}$ , **shield**  $\in \{\top, \perp\}$ , **virus**  $\in [0, t_{\text{virus}}]$ , **timer<sub>P</sub>**  $\in [0, t_{\max}]$ , and **timer<sub>L</sub>**  $\in [0, t_{\text{emit}}]$ . The first two variables **leader** and **shield** are Boolean variables:  $v.\text{leader} = \top$  means that agent  $v$  is a leader, and  $v.\text{shield}$  will be explained later. The variables **virus**, **timer<sub>P</sub>**, and **timer<sub>L</sub>** are count-down timers where their maximum values are  $t_{\text{virus}} = 60 \lceil \log N \rceil$ ,  $t_{\max} = 12c \cdot t_{\text{virus}} \lceil \log N \rceil$ , and  $t_{\text{emit}} = 12c \cdot t_{\text{virus}} \lceil \log N \rceil$ , respectively ( $t_{\max} = t_{\text{emit}}$ ).

Protocol  $P_{PL}$  consists of a *timeout mechanism* (Lines 1-3) and a *virus-war mechanism* (Lines 4-12). The timeout mechanism creates a leader when no leader exists in the population while the virus-war mechanism reduces the number of leaders when two or more leaders exist.

The timeout mechanism of  $P_{PL}$  (Lines 1-3) is almost the same as that of the protocol given in [3]. This mechanism uses a propagating timer  $\mathbf{timer}_P$ , which indicates the possibility of existence of a leader. A leader agent always keeps the maximum value of the timer, i.e.,  $\mathbf{timer}_P = t_{\max}$ , and resets the timer of the other agent to  $t_{\max}$  every time it interacts with a non-leader agent (Line 3). When two non-leaders interact, the higher value of the two timers is propagated, but is decremented by one (Line 1). When the timer of a non-leader decreases to zero, it suspects that no leader exists in the population, and it becomes a new leader (Line 2). The loosely-stabilizing property of this mechanism holds because (i) when no leader exists, some agent detects the timeout of its timer within a short time ( $O(t_{\max} \log n)$  parallel time) and it becomes a leader, and (ii) when at least one leader exists, timeout rarely happens thanks to the timer reset by the leader(s) and the larger-value propagation.

We use the virus war mechanism presented in [4], but implements it in a considerably different way to achieve a poly-logarithmic convergence time. Every leader tries to kill other leaders by using viruses and become the unique leader. We say that agent  $v$  has a virus if  $v.\mathbf{virus} > 0$ , and that  $v$  is wearing a shield if  $v.\mathbf{shield} = \top$ . Every agent has a local timer  $\mathbf{timer}_L$  to create a new virus periodically. This timer is decreased by one every time the agent interacts (Line 6). When the local timer of a leader reaches zero at an interaction, the leader meets a different fate according to its role, initiator or responder, in the interaction. If the leader is an initiator, it succeeds in creating a new virus and wears a shield, that is, it substitutes  $\mathbf{virus} \leftarrow t_{\mathbf{virus}}$  and  $\mathbf{shield} \leftarrow \top$  (Lines 8-9). If it is a responder, it fails to create a new virus and its shield gets broken if it wears (Line 11). Note that the uniformly-random scheduler gives each side of the coin-toss (initiator or responder) the same probability, i.e.,  $1/2$ . Thereafter, the local timer is reset to the maximum value  $t_{\mathbf{emit}}$  (Line 12). A virus spreads by interactions (Line 4). A leader is killed and becomes a non-leader if it catches a virus when it does not wear a shield (Line 5). The value of  $\mathbf{virus}$  corresponds to the TTL (time to live) of a virus and decreases in the large-value propagation fashion. The loosely-stabilizing property of this mechanism holds because (i) as long as multiple leaders exist, the number of leaders decreases by half in every  $O(t_{\mathbf{emit}})$  parallel time thanks to the fair coin-toss of the uniformly random scheduler and (ii) viruses rarely remove all leaders from the population thanks to  $t_{\mathbf{virus}} \ll t_{\mathbf{emit}}$ . ( $t_{\mathbf{virus}} \ll t_{\mathbf{emit}}$  guarantees that at least one leader wears a shield with high probability when viruses exist in the population.)

The above intuitive explanation holds if  $t_{\mathbf{virus}}$  is sufficiently large. However, we assign logarithmic value to  $t_{\mathbf{virus}}$  to get poly-logarithmic convergence time. A critical question arises here: Are created viruses propagated to the whole population with high probability? A similar question arises for the propagating timers. Careful and non-trivial analysis, omitted from this paper, affirms these questions and proves the performance of  $P_{PL}$  in Table 1.

---

## References

- 1 D. Angluin, J. Aspnes, Z. Diamadi, M.J. Fischer, and R. Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed Computing*, 18(4):235–253, 2006.
- 2 T. Izumi. On space and time complexity of loosely-stabilizing leader election. In *SIROCCO*, pages 299–312, 2015.
- 3 Y. Sudo, J. Nakamura, Y. Yamauchi, F. Ooshita, et al. Loosely-stabilizing leader election in a population protocol model. *Theoretical Computer Science*, 444:100–112, 2012.
- 4 Y. Sudo, F. Ooshita, H. Kakugawa, and T. Masuzawa. Loosely-stabilizing leader election on arbitrary graphs in population protocols. In *OPODIS*, pages 339–354, 2014.

