

# 25th International Conference on Principles of Distributed Systems

OPODIS 2021, December 13–15, 2021, Strasbourg, France

Edited by

Quentin Bramas

Vincent Gramoli

Alessia Milani



*Editors*

**Quentin Bramas** 

University of Strasbourg, France  
bramas@unistra.fr

**Vincent Gramoli** 

University of Sydney, Australia and EPFL, Switzerland  
vincent.gramoli@sydney.edu.au

**Alessia Milani**

LIS UMR 7020 CNRS, Aix-Marseille University, France  
alessia.milani@univ-amu.fr

*ACM Classification 2012*

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

**ISBN 978-3-95977-219-8**

*Published online and open access by*

Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at <https://www.dagstuhl.de/dagpub/978-3-95977-219-8>.

*Publication date*

February, 2022

*Bibliographic information published by the Deutsche Nationalbibliothek*

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

*License*

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



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

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

The copyright is retained by the corresponding authors.

Digital Object Identifier: 10.4230/LIPICs.OPODIS.2021.0

ISBN 978-3-95977-219-8

ISSN 1868-8969

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

## LIPICs – Leibniz International Proceedings in Informatics

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

### *Editorial Board*

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

**ISSN 1868-8969**

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



## ■ Contents

Preface	
<i>Quentin Bramas, Vincent Gramoli, and Alessia Milani</i> .....	0:ix–0:x
Program Committee	
.....	0:xi
Steering Committee	
.....	0:xiii
External Reviewers	
.....	0:xv

### Invited Talks

Distributed Algorithms: A Challenging Playground for Model Checking	
<i>Nathalie Bertrand</i> .....	1:1–1:1
Accountable Distributed Computing	
<i>Petr Kuznetsov</i> .....	2:1–2:1
A Fresh Look at the Design and Implementation of Communication Paradigms	
<i>Robbert van Renesse</i> .....	3:1–3:1

### Regular Papers

Arbitrarily Accurate Aggregation Scheme for Byzantine SGD	
<i>Alexandre Maurer</i> .....	4:1–4:17
Good-Case and Bad-Case Latency of Unauthenticated Byzantine Broadcast: A Complete Categorization	
<i>Ittai Abraham, Ling Ren, and Zhuolun Xiang</i> .....	5:1–5:20
On Finality in Blockchains	
<i>Emmanuelle Anceaume, Antonella Del Pozzo, Thibault Rieutord, and Sara     Tucci-Piergiovanni</i> .....	6:1–6:19
Twins: BFT Systems Made Robust	
<i>Shehar Bano, Alberto Sonnino, Andrey Chursin, Dmitri Perelman, Zekun Li,     Avery Ching, and Dahlia Malkhi</i> .....	7:1–7:29
Near-Optimal Dispersion on Arbitrary Anonymous Graphs	
<i>Ajay D. Kshemkalyani and Gokarna Sharma</i> .....	8:1–8:19
Asynchronous Gathering in a Torus	
<i>Sayaka Kamei, Anissa Lamani, Fukuhito Ooshita, Sébastien Tixeuil, and     Koichi Wada</i> .....	9:1–9:17
Pattern Formation by Robots with Inaccurate Movements	
<i>Kaustav Bose, Archak Das, and Buddhadeb Sau</i> .....	10:1–10:20

25th International Conference on Principles of Distributed Systems (OPODIS 2021).

Editors: Quentin Bramas, Vincent Gramoli, and Alessia Milani



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

Near-Shortest Path Routing in Hybrid Communication Networks <i>Sam Coy, Artur Czumaj, Michael Feldmann, Kristian Hinnenthal, Fabian Kuhn, Christian Scheideler, Philipp Schneider, and Martijn Struijs</i> .....	11:1–11:23
Efficient Assignment of Identities in Anonymous Populations <i>Leszek Gąsieniec, Jesper Jansson, Christos Levcopoulos, and Andrzej Lingas</i> .....	12:1–12:21
Population Protocols for Graph Class Identification Problems <i>Hiroto Yasumi, Fukuhito Ooshita, and Michiko Inoue</i> .....	13:1–13:19
Fast Graphical Population Protocols <i>Dan Alistarh, Rati Gelashvili, and Joel Rybicki</i> .....	14:1–14:18
Beyond Distributed Subgraph Detection: Induced Subgraphs, Multicolored Problems and Graph Parameters <i>Amir Nikabadi and Janne H. Korhonen</i> .....	15:1–15:18
An Improved Random Shift Algorithm for Spanners and Low Diameter Decompositions <i>Sebastian Forster, Martin Grösbacher, and Tijn de Vos</i> .....	16:1–16:17
Distributed CONGEST Approximation of Weighted Vertex Covers and Matchings <i>Salwa Faour, Marc Fuchs, and Fabian Kuhn</i> .....	17:1–17:20
Improved Distributed Fractional Coloring Algorithms <i>Alkida Balliu, Fabian Kuhn, and Dennis Olivetti</i> .....	18:1–18:23
Distributed Recoloring of Interval and Chordal Graphs <i>Nicolas Bousquet, Laurent Feuilloley, Marc Heinrich, and Mikaël Rabie</i> .....	19:1–19:17
Non-Blocking Dynamic Unbounded Graphs with Worst-Case Amortized Bounds <i>Bapi Chatterjee, Sathya Peri, Muktikanta Sa, and Komma Manogna</i> .....	20:1–20:25
Explicit Space-Time Tradeoffs for Proof Labeling Schemes in Graphs with Small Separators <i>Orr Fischer, Rotem Oshman, and Dana Shamir</i> .....	21:1–21:22
Local Certification of Graph Decompositions and Applications to Minor-Free Classes <i>Nicolas Bousquet, Laurent Feuilloley, and Théo Pierron</i> .....	22:1–22:17
RandSolomon: Optimally Resilient Random Number Generator with Deterministic Termination <i>Luciano Freitas de Souza, Andrei Tonkikh, Sara Tucci-Piergiovanni, Renaud Sirdey, Oana Stan, Nicolas Quero, and Petr Kuznetsov</i> .....	23:1–23:16
Optimal Space Lower Bound for Deterministic Self-Stabilizing Leader Election Algorithms <i>Lélia Blin, Laurent Feuilloley, and Gabriel Le Bouder</i> .....	24:1–24:12
Accountability and Reconfiguration: Self-Healing Lattice Agreement <i>Luciano Freitas de Souza, Petr Kuznetsov, Thibault Rieutord, and Sara Tucci-Piergiovanni</i> .....	25:1–25:23
Design and Analysis of a Logless Dynamic Reconfiguration Protocol <i>William Schultz, Siyuan Zhou, Ian Dardik, and Stavros Tripakis</i> .....	26:1–26:16

Optimal Good-Case Latency for Rotating Leader Synchronous BFT <i>Ittai Abraham, Kartik Nayak, and Nibesh Shrestha</i> .....	27:1–27:19
Strongly Linearizable Linked List and Queue <i>Steven Munsu Hwang and Philipp Woelfel</i> .....	28:1–28:20
Recoverable and Detectable Fetch&Add <i>Liad Nahum, Hagit Attiya, Ohad Ben-Baruch, and Danny Hendler</i> .....	29:1–29:17
Using Nesting to Push the Limits of Transactional Data Structure Libraries <i>Gal Assa, Hagar Meir, Guy Golan-Gueta, Idit Keidar, and Alexander Spiegelman</i>	30:1–30:17
Asynchronous Rumor Spreading in Dynamic Graphs <i>Bernard Mans and Ali Pourmiri</i> .....	31:1–31:20





## ■ Preface

The papers in this volume were presented at the 25th International Conference on Principles of Distributed Systems (OPODIS 2021), held on December 13–15, 2021 in Strasbourg, France.

OPODIS is an open forum for the exchange of state-of-the-art knowledge about distributed computing. With strong roots in the theory of distributed systems, OPODIS has expanded its scope to cover the entire range between the theoretical aspects and practical implementations of distributed systems, as well as experimental and quantitative assessments. All aspects of distributed systems are within the scope of OPODIS: theory, specification, design, performance, and system building. Specifically, this year, the topics of interest at OPODIS included:

- Biological distributed algorithms
- Blockchain technology and theory
- Communication networks (protocols, architectures, services, applications)
- Cloud computing and data centers
- Dependable distributed algorithms and systems
- Design and analysis of concurrent and distributed data structures
- Design and analysis of distributed algorithms
- Randomization in distributed computing
- Social systems, peer-to-peer and overlay networks
- Distributed event processing
- Distributed operating systems, middleware, and distributed database systems
- Distributed storage and file systems, large-scale systems, and big data analytics
- Edge computing
- Embedded and energy-efficient distributed systems
- Game-theory and economical aspects of distributed computing
- Security and privacy, cryptographic protocols
- Synchronization, concurrent algorithms, shared and transactional memory
- Impossibility results for distributed computing
- High-performance, cluster, cloud and grid computing
- Internet of things and cyber-physical systems
- Mesh and ad-hoc networks (wireless, mobile, sensor), location and context-aware systems
- Mobile agents, robots, and rendezvous
- Programming languages, formal methods, specification and verification applied to distributed systems
- Self-stabilization, self-organization, autonomy
- Distributed deployments of machine learning

We received 70 submissions, each of which underwent a double-blind peer review process. Three submissions were rejected for being out of the scope of the conference or having the wrong format. Overall, the quality of the submissions was very high. From the 70 submissions, 28 papers were selected to be included in these proceedings.

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



The review process was done using HotCRP. The Best Paper Award was awarded to Ittai Abraham, Kartik Nayak and Nibesh Shrestha for their paper titled “Optimal Good-case Latency for Rotating Leader Synchronous BFT”. The Best Student Paper Award was given to Gabriel Le Boudier for his paper titled “Optimal Space Lower Bound for Deterministic Self-Stabilizing Leader Election Algorithms”, co-authored with Laurent Feuilloley and Lélia Blin.

This year OPODIS had three distinguished invited keynote speakers: Nathalie Bertrand (INRIA, Rennes), Petr Kuznetsov (INFRES, Telecom Paris, Institut Polytechnique de Paris) and Robbert van Renesse (Cornell University, Ithaca, NY, USA).

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

Organizing this event would not have been possible without the help of the Networks Team of the ICUBE Laboratory.

Finally, we thank the Steering Committee members for their valuable advice, as well as the sponsors and the University of Strasbourg for their support.

November 2021

Quentin Bramas (University of Strasbourg, ICUBE, France)  
Vincent Gramoli (University of Sydney and EPFL, Switzerland)  
Alessia Milani (LIS, Aix-Marseille Université, France)

## ■ Program Committee

### General Chair

Quentin Bramas, ICUBE, University of Strasbourg, France

### Program Chairs

Vincent Gramoli, University of Sydney and EPFL, Switzerland

Alessia Milani, LIS, Aix-Marseille Université, France

### Program Committee

Emmanuelle Anceaume, CNRS / IRISA

Hagit Attiya, Technion

Alkida Balliu, University of Freiburg

Alysson Bessani, LASIGE, FCUL, Universidade de Lisboa

Borzoo Bonakdarpour, Michigan State University

Janna Burman, Université Paris-Saclay, CNRS

Armando Castaneda, UNAM

Giuseppe Antonio Di Luna, Sapienza, Università di Roma

Alexey Gotsman, IMDEA Software Institute

Eschar Hillel, Yahoo

Colette Johnen, University of Bordeaux- LaBRI

Sayaka Kamei, Hiroshima University

Alex Kogan, Oracle Labs

Dariusz Kowalski, University of Liverpool

Kostas Magoutis, University of Crete and FORTH-ICS

Avery Miller, University of Manitoba

Marina Papatrifaftou, Chalmers University

Ami Paz, University of Vienna

Yvonne-Anne Pignolet, DFINITY Foundation

Etienne Rivière, Université catholique de Louvain

Luis Rodrigues, Universidade de Lisboa

Jared Saia, University of New Mexico

Stefan Schmid, University of Vienna

Gokarna Sharma, Kent State University

Michael Spear, Lehigh University

Jukka Suomela, Aalto University

Nitin Vaidya, Georgetown University

Gauthier Voron, EPFL





## ■ Steering Committee

Xavier Defago, Tokyo Institute of Technology, Japan

Panagiota Fatourou, University of Crete, Greece

Pascal Felber, University of Neuchâtel, Switzerland (**Chair**)

Paola Flocchini, University of Ottawa, Canada

Roy Friedman, Technion, Israel

Seth Gilbert, National University of Singapore

Vincent Gramoli, University of Sydney and EPFL, Switzerland

Alessia Milani, LIS, Aix-Marseille Université, France



## ■ External Reviewers

Evangelos Bampas, Université Paris-Saclay, LISN  
Andrea Clementi, Università degli Studi di Roma “TorVergata”  
Matthew Connor, University of Liverpool  
Stéphane Devismes, Université de Grenoble Alpes  
Laxman Dhulipala, MIT  
Laurent Feuilloley, University of Lyon 1  
Dianne Foreback, Kent State University  
Rati Gelashvili, Novi  
Yossi Gilad, Hebrew University of Jerusalem  
Chetan Gupta, Aalto University  
Juho Hirvonen, Aalto University  
David Ilcinkas, CNRS, Bordeaux, France  
Adnane Khattabi Riffi, Université de Bordeaux, CNRS, LaBRI  
Seri Khoury, UC Berkeley  
Yonghwan Kim, Nagoyoya Institute of Technology  
Christian Konrad, University of Bristol  
Janne H. Korhonen, IST Austria  
Anissa Lamani, University of Strasbourg, ICube  
Yannic Maus, TU Graz  
Uri Meir, Tel-Aviv University  
Darya Melnyk, Aalto University  
Junya Nakamura, Toyohashi University of Technology  
Yasamin Nazari, University of Salzburg  
Shreyas Pai, Aalto University  
Gopal Pandurangan, University of Houston  
Mor Perry, The Academic College of Tel Aviv-Yaffo  
Mikael Rabie, University de Paris, IRIF  
Ritam Ganguly, Michigan State University  
Matan Rusanovsky, Ben Gurion University  
Elad Michael Schiller, Chalmers University of Technology  
Noa Schiller, Technion  
Anastasios Sidiropoulos, University of Illinois at Chicago  
Jan Studený, Aalto University  
Yuichi Sudo, Hosei University  
Sara Tucci-Piergiovanni, CEA LIST, Université de Paris-Saclay  
Alex Weaver, Georgetown University  
Jennifer L. Welch, Texas A&M University  
Yukiko Yamauchi, Faculty of Information Science and Electrical Engineering, Kyushu University, Japan  
Max Young, Mississippi State







# Distributed Algorithms: A Challenging Playground for Model Checking

Nathalie Bertrand  

Univ Rennes, Inria, CNRS, IRISA, France

---

## Abstract

Distributed computing is increasingly spreading, in advanced technological applications as well as in our daily life. Failures in distributed algorithms can have important human and financial consequences, so that it is crucial to develop rigorous techniques to verify their correctness. Model checking is a model-based approach to formal verification, dating back the 80's. It has been successfully applied first to hardware, and later to software verification.

Distributed computing raises new challenges for the model checking community, and calls for the development of new verification techniques and tools. In particular, the parameterized verification paradigm is nowadays blooming to help proving automatically the correctness of distributed algorithms. In this invited talk, we present recent parameterized verification developments to automatically prove properties of some classical distributed algorithms.

**2012 ACM Subject Classification** Theory of computation → Verification by model checking; Theory of computation → Distributed algorithms

**Keywords and phrases** Verification, Distributed algorithms

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2021.1

**Category** Invited Talk



© Nathalie Bertrand;

licensed under Creative Commons License CC-BY 4.0

25th International Conference on Principles of Distributed Systems (OPODIS 2021).

Editors: Quentin Bramas, Vincent Gramoli, and Alessia Milani; Article No. 1; pp. 1:1–1:1

Leibniz International Proceedings in Informatics



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



# Accountable Distributed Computing

Petr Kuznetsov ✉

LTCI, Télécom Paris, Institut Polytechnique de Paris, France

---

## Abstract

There are two major ways to deal with failures in distributed computing: *fault-tolerance* and *accountability*. Fault-tolerance intends to anticipate failures by investing into replication and synchronization, so that the system's correctness is not affected by faulty components. In contrast, accountability enables detecting failures *a posteriori* and raising undeniable evidences against faulty components.

In this talk, we discuss how accountability can be achieved, both in generic and application-specific ways. We begin with an overview of fault detection mechanisms used in benign, *crash-prone* system, with a focus on the *weakest failure detector* question. We then consider the fault detection problem in systems with general, *Byzantine* failures and explore which classes of misbehavior can be detected and which – cannot. We then study the mechanism of *application-specific* accountability that, intuitively, only accounts for instances of misbehavior that affect particular correctness criteria. Finally, we discuss how fault detection can be combined with *reconfiguration*, opening an avenue of “self-healing” systems that seamlessly replace faulty system components with correct ones.

**2012 ACM Subject Classification** Theory of computation → Distributed algorithms

**Keywords and phrases** Fault-tolerance, fault detection, accountability, application-specific

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2021.2

**Category** Invited Talk



© Petr Kuznetsov;

licensed under Creative Commons License CC-BY 4.0

25th International Conference on Principles of Distributed Systems (OPODIS 2021).

Editors: Quentin Bramas, Vincent Gramoli, and Alessia Milani; Article No. 2; pp. 2:1–2:1



Leibniz International Proceedings in Informatics



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



# A Fresh Look at the Design and Implementation of Communication Paradigms

Robbert van Renesse  

Cornell University, Ithaca, NY, USA

---

## Abstract

Datacenter applications consist of many communicating components and evolve organically as requirements develop over time. In this talk I will present two projects that try to support such organic growth. The first project, Escher, recognizes that components of a distributed systems may themselves be distributed systems. Escher introduces a communication abstraction that hides the internals of a distributed component, and in particular how to communicate with it, from other components. Using Escher, a replicated server can invoke another replicated server without either server having to even know that the servers are replicated. The second project, Scalog, presents a datacenter scale totally ordered logging service. Logs are increasingly a central component in many datacenter applications, but log configurations can lead to significant hiccups in the performance of those applications. Scalog has seamless reconfiguration operations that allow it to scale up and down without any downtime.

**2012 ACM Subject Classification** Theory of computation → Distributed algorithms

**Keywords and phrases** Distributed systems

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2021.3

**Category** Invited Talk



© Robbert van Renesse;

licensed under Creative Commons License CC-BY 4.0

25th International Conference on Principles of Distributed Systems (OPODIS 2021).

Editors: Quentin Bramas, Vincent Gramoli, and Alessia Milani; Article No. 3; pp. 3:1–3:1

Leibniz International Proceedings in Informatics



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



# Arbitrarily Accurate Aggregation Scheme for Byzantine SGD

Alexandre Maurer

School of Computer Science, UM6P, Ben Guerir, Morocco

---

## Abstract

---

A very common optimization technique in Machine Learning is Stochastic Gradient Descent (SGD). SGD can easily be distributed: several *workers* try to estimate the gradient of a loss function, and a central *parameter server* gathers these estimates. When all workers behave correctly, the more workers we have, the more accurate the gradient estimate is. We call this the Arbitrary Aggregation Accuracy (AAA) property.

However, in practice, some workers may be Byzantine (i.e., have an arbitrary behavior). Interestingly, when a fixed fraction of workers is assumed to be Byzantine (e.g. 20%), no existing aggregation scheme has the AAA property. In this paper, we propose the first aggregation scheme that has this property despite a fixed fraction of Byzantine workers (less than 50%). We theoretically prove this property, and then illustrate it with simulations.

**2012 ACM Subject Classification** Computing methodologies → Machine learning; Computing methodologies → Distributed algorithms

**Keywords and phrases** distributed machine learning, Byzantine failures, stochastic gradient descent

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2021.4

**Supplementary Material** *Software (Source Code)*: <https://tinyurl.com/sim-aaa-paper>

## 1 Introduction

Many machine learning models are trained using *stochastic gradient descent* (SGD) [17], an optimization technique that can easily be parallelized on multiple computers. As machine learning models become larger and larger, parallelizing their training becomes more and more important, if we want to train them in a reasonable amount of time.

If all computers are assumed to work correctly, parallelizing the training is relatively simple. The classical architecture is the following. A central *parameter server* is trying to minimize a *loss function*. To do so, it uses the gradient descent algorithm, which requires to compute (an approximation of) the gradient of the loss function, at several points of the loss function. As this is the most time-consuming task, the parameter server distributes this task among multiple *workers*. Each worker computes a vector which is an approximation of the desired gradient. The parameter server then collects and aggregates these vectors, to obtain a (reasonably good) approximation of the gradient. This process is repeated multiple times, until we reach a minimum of the loss function. We explain this in more details in Section 2.3.

However, when the number of workers becomes very large, one should assume that some workers will *not* behave correctly. Some workers may even be malicious agents trying to prevent a successful training of the model. This is especially true when workers are not identical computers stored in a data center, but personal computers or smartphones participating in the training in a collaborative way.

Therefore, a recent line of work is *robust distributed SGD*: the goal is to propose distributed architectures that manage to train the model despite the presence of malicious workers. In order to achieve very strong safety guarantees, we assume that these malicious workers are *Byzantine* [15], that is: they are omniscient, and can have any arbitrary behavior.



© Alexandre Maurer;

licensed under Creative Commons License CC-BY 4.0

25th International Conference on Principles of Distributed Systems (OPODIS 2021).

Editors: Quentin Bramas, Vincent Gramoli, and Alessia Milani; Article No. 4; pp. 4:1–4:17

Leibniz International Proceedings in Informatics



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

Let us give a quick overview of this literature.<sup>1</sup> In [18], a first solution was proposed for problems of dimension one. In [3], a score is associated to each proposed vector, defined as the sum of distances with some of its closest neighbors. The vector with the smallest score is then selected. In [9], several outputs of [3] are selected (without replacement), and then averaged. In [21], a scalar aggregation rule (SAR) is applied to each coordinate. This SAR can be, for instance, the median, or a trimmed mean (a mean after removing the  $x\%$  largest and  $x\%$  smallest values). In [19] (resp. [20]), the proposed values closest from the median (resp. trimmed mean) are selected, then averaged. In [12], a variant of the trimmed mean is applied to a selection of vectors obtained from [9]. In [7], several batches of vectors are averaged; then, their geometric median is computed. In [6], the vectors are aggregated using coding theory and a redundancy scheme. In [1] and [10], historical information is used to identify dishonest workers. The only solution tolerating asynchrony so far is [10].

Note that most of these works assume a centralized and reliable parameter server. However, as shown in [11], these schemes<sup>2</sup> can be transformed into fully decentralized schemes, where no entity is a single point of failure. In order to focus on the aggregation strategy, we also assume a centralized parameter server in this paper.

In the following, we focus on aggregation schemes where an approximation of the gradient is computed independently at each step, like in classical SGD. Aside from enabling a clearer mathematical analysis of the gradient, it also makes the system resilient to transient failures, that is: in addition of Byzantine workers, the system can recover from any temporary failures of correct workers.

### The Arbitrary Aggregation Accuracy (AAA) property

Now, let us come back to the case where all workers are correct, and consider a given step of the gradient descent algorithm. In this setting, the parameter server simply computes the mean of the received vectors. If each workers processes a given share of the dataset, and these shares are independent and identically distributed (which is usually assumed), then the more workers we have, the more *accurate* our approximation of the true gradient will be. Actually, for any arbitrary level of precision, there exists a number of workers that can achieve this level of precision. We call this the *Arbitrary Aggregation Accuracy* (AAA) property.

Formally, if  $G$  is the true gradient of the loss function (at a given step),  $n$  is the number of workers, and  $A_n$  is the vector aggregated by the parameter server (the approximation of the true gradient), then this property can be expressed as follows:<sup>3</sup>

$$\lim_{n \rightarrow +\infty} \mathbb{E} \|A_n - G\| = 0.$$

We now assume that there is a fixed fraction of Byzantine workers (for instance, 20% of workers, independently of the total number of workers). We may ask the following question: is it possible to have the AAA property in this setting?

---

<sup>1</sup> This problem shares some similarities with the more general problem of executing arbitrary tasks in a setting with a “master” and several (unreliable) “workers”. In [14], a bound is shown on the complexity of performing  $n$  tasks correctly with high probability. However, doing so does not give meaningful guarantees when the goal is to perform SGD.

<sup>2</sup> This applies to any scheme of the same (centralized) nature as those presented above.

<sup>3</sup> In this paper,  $\|\cdot\|$  refers to the L2 norm, and the expectation is both on the i.i.d. samples of the dataset and on the randomness involved in the algorithm.



Interestingly, no existing Byzantine-resilient version of SGD has this property (more details on this in Section 2.5): a fixed fraction of Byzantine workers results in a fixed error w.r.t. the true gradient, no matter how large the number of workers is.

### Our contribution

In this paper, we propose COMPASS, the first aggregation scheme that has the AAA property despite a fixed fraction  $f$  of Byzantine workers ( $f < \frac{1}{2}$ ). We describe this scheme, then prove that it has the AAA property. We then illustrate this property with simulations: we compare the accuracy of COMPASS (a modified version of COMPASS<sup>4</sup>) to an existing aggregation scheme.

The rest of the paper is organized as follows. In Section 2, we describe the general setting. In Section 3, we describe the COMPASS aggregation scheme. In Section 4, we prove that COMPASS has the AAA property. In Section 5, we illustrate the AAA property with simulations. We conclude in Section 6.

### Remarks and clarifications

Before going further, let us clarify several points about the contribution of this paper.

- This work is mostly a theoretical work.
- This work, as well as many previous works, proposes a scheme to approximate the gradient of the loss function. The precision of this approximation (of the gradient) should not be confused with the precision of the learned model. For a given gradient descent step, having an accurate gradient is always a desirable property, since the goal of a gradient descent step is to decrease the value of the loss function. Therefore, an accurate gradient approximation will not be the cause of problems like overfitting.
- Similarly, guarantees on the quality of the gradient approximation (as provided in this work and several previous works) should not be confused with guarantees on the accuracy of the trained model. Such guarantees can be found, for instance, in [1] and [5], under specific hypotheses (e.g., convex loss function, bounded gradient...).
- The gradient approximations proposed by correct workers may have a variance high enough to allow Byzantine workers to collude to move the mean without being detected (as described in [2]). However, they can only do so in aggregation schemes where the AAA property is not satisfied: this property ensures that the estimated gradient is arbitrarily close to the true gradient, independently of the behavior of Byzantine workers.
- In the SGD algorithm, there is always a probability of error w.r.t. the true gradient. It is also the case for distributed SGD (without failures), and for Byzantine-resilient solutions (including ours). Nevertheless, we ensure that this error shrinks to zero when the number of nodes increases.

## 2 Preliminaries

### 2.1 Setting

We want to train a machine learning model of  $d$  parameters, using the gradient descent algorithm.

---

<sup>4</sup> The reason for this change is motivated in Section 5.1.

## 4:4 Arbitrarily Accurate Aggregation Scheme for Byzantine SGD

The model can be represented by a function  $\mathcal{M}(P, X)$ , where  $P = (p_1, \dots, p_d)$  are the *parameters* of the model (for instance, the weights and biases of a neural network), and  $X$  is the current input of the model (usually a vector of real values).  $\mathcal{M}(P, X)$  returns a single real value  $y$ .

To train the model, we have a dataset consisting in two lists  $(X_1, \dots, X_m)$  and  $(y_1, \dots, y_m)$ , where  $X_i$  is an input of the model (*feature*), and  $y_i$  is the corresponding desired output (*label*). In the following, we denote  $y_i$  by  $y(X_i)$ . We define a *loss function*  $L(P)$ , measuring the “distance” between the current model and the desired outputs. For instance, a classic form of the loss function is:

$$L(P) = \frac{1}{m} \sum_{i=1}^m (\mathcal{M}(P, X_i) - y(X_i))^2.$$

We make no hypotheses on the shape of this function, except that it has, in each point, a gradient with finite coordinate values.<sup>5</sup>

Training the model consists in finding a set of parameters minimizing the loss function. The standard way to do this is to use the *gradient descent* algorithm, which consists of repeating the two following steps:

1. Compute the gradient  $\nabla L(P)$  of the loss function.
2. Update the vector of parameters  $P$  as follows:  $P \leftarrow P - \alpha \nabla L(P)$  (where  $\alpha$  is an arbitrarily small constant).

### 2.2 Stochastic gradient descent (SGD)

In practice, computing the exact gradient may be very long when the dataset contains a lot of elements (which is usually the case). An alternative is to use *stochastic gradient descent* (SGD), that is: at each step, we randomly select a set  $S$  of elements from the dataset, and use it to compute an approximation  $\nabla L^*(P, S)$  of  $\nabla L(P)$ . For instance, if  $L(P)$  has the aforementioned classical form, then:

$$L^*(P, S) = \frac{1}{|S|} \sum_{X \in S} (\mathcal{M}(P, X) - y(X))^2.$$

Over several steps, the errors due to randomness tend to cancel each other, and we generally achieve the same result with a much shorter computation time.

### 2.3 Distributed SGD

A convenient property of SGD is that it can easily be parallelized. The classical architecture is the following. We have a *parameter server*, that stores and updates the parameters of the model, and  $n$  *workers*  $(w_1, \dots, w_n)$ .

Let  $\alpha > 0$  be an arbitrarily small constant. At each step:

1. The parameter server sends the current vector of parameters  $P$  to each worker.
2. Each worker  $w_i$  selects a random subset  $S$  of the dataset, computes  $V_i = \nabla L^*(P, S)$ , and sends it to the parameter server.
3. The parameter server computes the mean  $A_n$  of the received vectors  $V_i$  ( $A_n = \frac{1}{n} \sum_{i=1}^n V_i$ ), and uses it to update the model ( $P \leftarrow P - \alpha A_n$ ).

---

<sup>5</sup> This function may have multiple local minima. However, for most machine learning models (e.g. neural networks), most local minima are sufficient to reach a satisfying accuracy [8].

Here, we assume that each worker possesses a copy of the whole dataset, and can randomly select elements from the dataset at each step. This is a reasonable hypothesis, given the current memory capacities of computers (or even smartphones), and the cheap cost of memory units (relatively to the cost of computing power). We make this hypothesis in the rest of the paper. Another justification might be that workers have remote access to the dataset through the internet (and may copy specific parts of it).

## 2.4 Failure model

In the aforementioned distributed setting, all workers are assumed to behave correctly. However, in practice, this may not always be the case.

Let  $f < 0.5$  be a fixed parameter of the problem. Let  $k$  be the largest integer such that  $k \leq fn$ . Among the  $n$  workers,  $k$  are assumed to be *Byzantine*, that is: their behavior is completely arbitrary. Here, as the workers send vectors to the parameter server, this means that up to  $k$  workers can send arbitrary vectors to the parameter server. The parameter server does not know which workers are Byzantine.

Note that, since the behavior of Byzantine workers is arbitrary, it does not matter whether or not they keep track of past events: we must always assume the worst-case scenario.

## 2.5 The Arbitrary Aggregation Accuracy (AAA) property

An *aggregation scheme* is a distributed system that, for a given step of the gradient descent algorithm, produces an approximation  $A$  of the true gradient  $G = \nabla L(P)$  of the loss function. The scheme presented in 2.3 is an example of aggregation scheme. If such a system allows an arbitrary number  $n$  of workers, we call the resulting aggregated vector  $A_n$ .

We say that an aggregation scheme has the Arbitrary Aggregation Accuracy (AAA) property if the expected value of the distance between  $A_n$  and  $G$  approaches 0 when  $n$  increases:

$$\lim_{n \rightarrow +\infty} \mathbb{E} \|A_n - G\| = 0.$$

A classical metric for Byzantine-resilient versions of SGD is the *angular error*, that is: the angle  $\theta_n$  between  $E[A_n]$  and  $G$ . An asymptotic comparison of the angular errors of existing aggregation schemes is provided in [4]. When the fraction of Byzantine workers is constant (independently of  $n$ ), these angular errors are at best  $\Theta(1)$  (i.e. constant). This contradicts the AAA property: when this property is satisfied,  $\lim_{n \rightarrow +\infty} \theta_n = 0$ .

To give some intuition on why these previous solutions do not satisfy the AAA property, let us consider, for instance, the coordinate-wise median: for each coordinate, we take the median of the values proposed by workers (see Section 5.1 for a more formal description). In this setting, the worst thing Byzantine workers can do is to propose extreme values (all positive or all negative): even if they are a minority, this will push the median towards these extreme values. Here, for a given distribution of values, a fixed fraction of Byzantine workers will have a fixed impact of the median value. This phenomena is illustrated experimentally in Section 5.

## 3 Our aggregation scheme

In this section, we present the COMPASS aggregation scheme. In 3.1, we give some preliminary definitions. In 3.2, we describe COMPASS. In 3.3, we explain its general idea, and comment it step by step.

### 3.1 Definitions

We consider  $n$  workers  $(w_1, \dots, w_n)$ . Let  $N$  be the largest integer such that  $N^2 \leq n$ . Let  $M$  be a fixed parameter, corresponding to the number of elements of the dataset that a worker uses to compute an approximation of the gradient at each step.

We now define several notions and functions used in our aggregation scheme:

- A *random split* consists in randomly selecting a set  $S$  of  $N^2$  workers among  $(w_1, \dots, w_n)$ , then randomly splitting the elements of  $S$  into  $N$  sets  $(W_1, \dots, W_N)$ , each one containing  $N$  elements.
- A *random pick* is a set of  $M$  randomly selected integers between 1 and  $m$  (as a reminder,  $m$  is the size of the dataset).
- For a set  $S$  of vectors of dimension  $d$ , we define the function  $Maj(S)$  as follows:
  - If there exists a vector  $V$  of  $S$  such that a strict majority of vectors of  $S$  are equal to  $V$ , then,  $Maj(S) = V$ .
  - Otherwise,  $Maj(S)$  returns a null vector  $(0, 0, \dots, 0)$  of dimension  $d$ .
- For two values  $p$  and  $x$  (with  $x \geq 0$ ), we define  $Cut_0(p, x)$  as follows:
  - If  $p > x$ ,  $Cut_0(p, x) = x$
  - If  $p < -x$ ,  $Cut_0(p, x) = -x$
  - Otherwise,  $Cut_0(p, x) = p$
- For a vector  $V = (v_1, v_2, \dots, v_d)$  and a value  $x$ , we define  $Cut(V, x)$  as follows:

$$Cut(V, x) = (Cut_0(v_1, x), Cut_0(v_2, x), \dots, Cut_0(v_d, x)).$$

### 3.2 Description of the aggregation scheme

We now describe the COMPASS aggregation scheme (similarly to the distributed SGD scheme described in 2.3).

Let  $\alpha > 0$  be an arbitrarily small constant. At each step:

1. The parameter server generates a random split  $(W_1, \dots, W_N)$  and  $N$  random picks  $(Z_1, \dots, Z_N)$ .
2.  $\forall i \in \{1, \dots, N\}$ , the parameter server sends  $Z_i$  and the current vector of parameters  $P$  to each worker of the set  $W_i$ .
3.  $\forall i \in \{1, \dots, N\}$ , let  $\Omega_i$  be the set containing the elements  $X_j$  of the dataset such that  $j \in Z_i$ . Each worker of the set  $W_i$  computes  $\nabla L^*(P, \Omega_i)$ , and sends it to the parameter server.
4.  $\forall i \in \{1, \dots, N\}$ , let  $S_i$  be the set of vectors sent by the workers of  $W_i$  (the parameter server only accepts one vector per worker<sup>6</sup>). The parameter server aggregates the received vectors as follows:

$$A_n = Cut \left( \frac{1}{N} \sum_{i=1}^N Maj(S_i), \sqrt{N} \right)$$

...and uses it to update the model ( $P \leftarrow P - \alpha A_n$ ).

---

<sup>6</sup> If a worker does not send any vector before the end of the round, we consider that it sent a null vector  $(0, 0, \dots, 0)$ . Therefore,  $|S_i| = |W_i|$ .

### 3.3 Detailed explanation

#### General idea

A common strategy to defeat Byzantine processes is replication: many processes perform the same computing task, and a majority vote selects the correct output. Here, however, if all correct workers compute the same vector, adding more workers will not improve the quality of the gradient approximation. To do so, we have to *aggregate* many (independent) approximations. This is done, for instance, in the simple scheme described in Section 2.3 (with a mean). However, this scheme is not robust to even one Byzantine worker (as shown in [3]).

Here, we propose a balanced mix of replication and aggregation: if we have  $N^2$  workers ( $N \approx \sqrt{n}$ ), then, we can choose  $N$  sets of  $N$  workers each. Each set computes the same vector, and a majority vote determines the output of this set. We then aggregate these outputs. As  $N$  grows with  $n$ , increasing the number of workers increases *both* the reliability of replication *and* the quality of aggregation.

#### Step-by-step description

Now, let us comment on our aggregation scheme step by step.

In Step 1, the parameter server generates the  $N$  aforementioned sets ( $W_1, \dots, W_N$ ). To prevent any strategic placement of Byzantine workers, these sets are chosen randomly at each step. The parameter server also generates  $N$  random picks ( $Z_1, \dots, Z_N$ ). Each  $Z_i$  is a set of identifiers of elements of the dataset. These elements will be processed by the corresponding group of workers  $W_i$ .

In Step 2, the parameter server sends  $P$  (the current vector of parameters) and  $Z_i$  (the aforementioned set of identifiers) to each worker of the set  $W_i$ , for each  $i \in \{1, \dots, N\}$ .

In Step 3, each worker of the set  $W_i$  (for each  $i \in \{1, \dots, N\}$ ) computes the approximation  $\nabla L^*(P, \Omega_i)$  of the gradient  $\nabla L(P)$ , where  $\Omega_i$  is the set of elements of the dataset corresponding to  $Z_i$ . Then, it sends it back to the parameter server.

In Step 4, the parameter server aggregates the received vectors. First, it uses a majority vote ( $Maj(S_i)$ ) to determine the output of the set  $W_i$  (for each  $i \in \{1, \dots, N\}$ ). Then, it computes the mean of these outputs. Finally, it applies the *Cut* function to ensure that the coordinates of the aggregated vector remain smaller than  $\sqrt{N}$ . Doing so is important, because there is always a probability  $\mu > 0$  that a set  $W_i$  contains a majority of Byzantine workers. If so, the output of this set (after the majority vote) will be determined by Byzantine workers, that could (for instance) propose a vector with coordinate values inversely proportional to  $\mu$ , to ensure that the expected mean of outputs remains far away from the true gradient. Applying the *Cut* function enables to prove the result of Lemma 3 (see Section 4), and then the AAA property.

## 4 Analysis

In this section, we prove that COMPASS has the AAA property (see Theorem 4).

In the following proofs, we introduce several variables in order to constrain some values to be integers. For instance: “Let  $k$  be the largest integer such that  $k < n^2$ ”. Some readers may consider that it would be simpler to just write “ $\sqrt{n}$ ” here; however, some other readers may object that doing so would make the proofs less rigorous, and make their validity unclear. For this reason, we choose to use the first notation.

## 4:8 Arbitrarily Accurate Aggregation Scheme for Byzantine SGD

► **Lemma 1.** *Let  $p < \frac{1}{2}$  be a probability. Consider  $N$  sets of  $N$  workers, where each worker has an independent probability  $p$  to be Byzantine. Let  $E_N$  be the following event: “All  $N$  sets contain a strict minority of Byzantine workers”. Then, there exists  $N_0$  such that,  $\forall N \geq N_0$ ,  $P(E_N) \geq 1 - \frac{2}{N}$ .*

**Proof.** As  $\lim_{x \rightarrow +\infty} \frac{\ln x}{x} = 0$ , let  $N_0$  be the smallest integer such that,  $\forall N \geq N_0$ :

$$\frac{\ln N_0}{N_0} < \left( \frac{1 - 2p}{4} \right)^2.$$

Consider a set of workers. Let  $k$  be the number of Byzantine workers in this set. According to Hoeffding’s inequality:

$$P(|k - Np| \geq \sqrt{N \ln N}) \leq \frac{2}{N^2}.$$

Therefore:

$$P\left(\left|\frac{k}{N} - p\right| < \sqrt{\frac{\ln N}{N}}\right) \geq 1 - \frac{2}{N^2}$$

... and, for  $N \geq N_0$ :

$$P\left(\left|\frac{k}{N} - p\right| < \frac{1 - 2p}{4}\right) \geq 1 - \frac{2}{N^2}.$$

Now, we can remark that:

$$P\left(\left|\frac{k}{N} - p\right| < \frac{1 - 2p}{4}\right) \leq P\left(\frac{k}{N} - p < \frac{1 - 2p}{4}\right) = P\left(\frac{k}{N} < \frac{1 + 2p}{4}\right).$$

As  $p < \frac{1}{2}$ , we have  $1 + 2p < 2$ , and:

$$P\left(\frac{k}{N} < \frac{1 + 2p}{4}\right) \leq P\left(\frac{k}{N} < \frac{2}{4}\right) = P\left(k < \frac{N}{2}\right).$$

Thus,  $\forall N \geq N_0$ :

$$P\left(k < \frac{N}{2}\right) \geq 1 - \frac{2}{N^2}$$

and

$$P(E_N) \geq \left(1 - \frac{2}{N^2}\right)^N \geq 1 - \frac{2N}{N^2} = 1 - \frac{2}{N}. \quad \blacktriangleleft$$

► **Lemma 2.** *Let  $N \geq 1$  be an integer, and let  $f < \frac{1}{2}$  be a positive value. Consider  $N^2$  workers, among which  $k$  are Byzantine, with  $k \leq fN^2$ . Assume that these  $N^2$  workers are randomly assigned to  $N$  sets. Let  $E'_N$  be the following event: “All  $N$  sets contain a strict minority of Byzantine workers”. Then, there exists  $N_1$  such that,  $\forall N \geq N_1$ ,  $P(E'_N) \geq 1 - \frac{3}{N}$ .*

**Proof.** The proof of this lemma can be found in the appendix. ◀

► **Lemma 3.** *Let  $G$  be a vector of dimension  $d$ , and let  $p < \frac{3}{N}$  be a probability. Let  $(V_1, V_2, V_3, \dots)$  be a sequence of vectors of dimension  $d$ , such that  $\lim_{N \rightarrow +\infty} \frac{1}{N} \sum_{i=1}^N V_i = G$ .<sup>7</sup> Let  $(B_1, B_2, B_3, \dots)$  be an arbitrary sequence of vectors of dimension  $d$ . Let  $R_N$  be a random vector defined as follows: with probability  $p$ ,  $R_N = \text{Cut}(B_N, \sqrt{N})$ ; otherwise,  $R_N = \text{Cut}\left(\frac{1}{N} \sum_{i=1}^N V_i, \sqrt{N}\right)$ . Then,  $\lim_{N \rightarrow +\infty} \mathbb{E}\|R_N - G\| = 0$ .*

**Proof.** Before going further, let us clarify one possible misunderstanding. Some readers may confuse  $N$  with the number of parameters of the model (which is not the case). According to the lemma's statement,  $N$  is related to the numbers of vectors used to approximate the true gradient. The number of parameters of the model is not used in the proofs of this paper.

$$\mathbb{E}\|R_N - G\| = p \underbrace{\left\| \text{Cut}(B_N, \sqrt{N}) - G \right\|}_{X_N} + (1-p) \underbrace{\left\| \text{Cut}\left(\frac{1}{N} \sum_{i=1}^N V_i, \sqrt{N}\right) - G \right\|}_{Y_N}$$

By definition of the *Cut* function,  $\forall N \geq 1$ ,  $\left\| \text{Cut}(B_N, \sqrt{N}) \right\| \leq \sqrt{d}\sqrt{N}$ .

As  $p < \frac{3}{N}$ :

$$p \left\| \text{Cut}(B_N, \sqrt{N}) \right\| < \frac{3}{N} \sqrt{d}\sqrt{N} = \frac{3\sqrt{d}}{\sqrt{N}}.$$

Therefore:

$$X_N \leq p \left\| \text{Cut}(B_N, \sqrt{N}) \right\| + p\|G\| < \frac{3\sqrt{d}}{\sqrt{N}} + \frac{3}{N}\|G\|.$$

As a result,  $\lim_{N \rightarrow +\infty} X_N = 0$ . Now, let us determine  $\lim_{N \rightarrow +\infty} Y_N$ .

Let  $\delta > 0$ . Let  $j \in \{1, \dots, d\}$ , and let  $v(i, j)$  (resp.  $g(j)$ ) be the  $j^{\text{th}}$  coordinate of  $V_i$  (resp.  $G$ ). As  $\lim_{N \rightarrow +\infty} \frac{1}{N} \sum_{i=1}^N V_i = G$ , in particular:

$$\lim_{N \rightarrow +\infty} \frac{1}{N} \sum_{i=1}^N v(i, j) = g(j).$$

Thus, there exists  $n_j$  such that,  $\forall N \geq n_j$ :

$$\left| g(j) - \frac{1}{N} \sum_{i=1}^N v(i, j) \right| \leq \delta.$$

Thus,  $\forall N \geq n_j$ :

$$\left| \frac{1}{N} \sum_{i=1}^N v(i, j) \right| \leq \delta + |g(j)|.$$

Let  $N_0$  be the smallest integer such that  $N_0 \geq \max(n_1, \dots, n_d)$  and  $\sqrt{N_0} \geq \delta + \max_{j \in \{1, \dots, d\}} |g(j)|$ .

Then,  $\forall N \geq N_0$  and  $\forall j \in \{1, \dots, d\}$ :

$$\left| \frac{1}{N} \sum_{i=1}^N v(i, j) \right| \leq \sqrt{N}$$

<sup>7</sup> This sequence represents the vectors proposed by Byzantine workers. The reason why we write them as a sequence is that we further write " $\lim_{N \rightarrow +\infty} f_N$ ", where  $f_N$  is a function of  $B_N$ .

## 4:10 Arbitrarily Accurate Aggregation Scheme for Byzantine SGD

and

$$Cut_0 \left( \frac{1}{N} \sum_{i=1}^N v(i, j), \sqrt{N} \right) = \frac{1}{N} \sum_{i=1}^N v(i, j).$$

Thus, for all  $N \geq N_0$ :

$$Cut \left( \frac{1}{N} \sum_{i=1}^N V_i, \sqrt{N} \right) = \frac{1}{N} \sum_{i=1}^N V_i.$$

As a result:

$$\lim_{N \rightarrow +\infty} Cut \left( \frac{1}{N} \sum_{i=1}^N V_i, \sqrt{N} \right) = \lim_{N \rightarrow +\infty} \frac{1}{N} \sum_{i=1}^N V_i = G$$

and

$$\lim_{N \rightarrow +\infty} \left\| Cut \left( \frac{1}{N} \sum_{i=1}^N V_i, \sqrt{N} \right) - G \right\| = 0.$$

As  $p < \frac{3}{N}$ ,  $\lim_{N \rightarrow +\infty} (1 - p) = 1$ , and  $\lim_{N \rightarrow +\infty} Y_N = 0$ . Therefore:

$$\lim_{N \rightarrow +\infty} \mathbb{E} \|R_N - G\| = \lim_{N \rightarrow +\infty} X_N + \lim_{N \rightarrow +\infty} Y_N = 0. \quad \blacktriangleleft$$

► **Theorem 4.** *COMPASS has the AAA property.*

**Proof.** As a reminder,  $N$  is the largest integer such that  $N^2 \leq n$ .

Let  $(W_1, \dots, W_N)$  be the random split generated in Step 1 of COMPASS. According to Lemma 2, with a probability at least  $1 - \frac{3}{N}$ , each set  $W_i$  contains a strict minority of Byzantine workers. In other words, the probability  $p$  that these sets do *not* all contain a strict minority of Byzantine workers is such that  $p < \frac{3}{N}$ .

Besides, when all sets  $W_i$  contain a strict minority of Byzantine workers,  $Maj(S_i)$  corresponds to the vector sent by the correct workers of  $W_i$  (as a reminder,  $S_i$  is the set of vectors sent by the workers of  $W_i$ ). As these vectors  $Maj(S_i)$  are all based on random samples of the dataset,  $\mathbb{E}[Maj(S_i)] = G$ . In other words:

$$\lim_{N \rightarrow +\infty} \frac{1}{N} \sum_{i=1}^N Maj(S_i) = G.$$

Therefore, the output  $A_n$  of COMPASS can be represented by the random vector  $R_N$  of Lemma 3 (where the arbitrary vectors  $(B_1, B_2, B_3, \dots)$  correspond to the cases where not *all* sets  $W_i$  contain a strict minority of Byzantine workers).

When  $n \rightarrow +\infty$ ,  $N \rightarrow +\infty$ . Therefore, according to Lemma 3,  $\lim_{n \rightarrow +\infty} \mathbb{E} \|A_n - G\| = 0$ , and COMPASS has the AAA property. ◀

## 5 Simulations

In this section, we illustrate the AAA property with simulations. We compare COMPASS (a modified version of COMPASS) with an existing aggregation scheme. In 5.1, we describe these two aggregation schemes. In 5.2, we describe the simulation setting. In 5.3, we show how to make simulations both simpler and more general. The simulation results are presented in 5.4.



## 5.1 Aggregation schemes

Let us describe the aggregation schemes CWMED and COMPMED.

- CWMED (*Coordinate-Wise Median*) is an aggregation scheme introduced in [21]. It consists in taking the median value for each coordinate, in order to exclude extreme values proposed by Byzantine workers. Its angular error is constant. As a reminder (see 2.5), among existing aggregation schemes, the angular error is at best constant.
- COMPMED is a modified version of COMPASS. The principle is the same, except that the final aggregation formula is now similar to CWMED. The reason for this change is that COMPASS is designed to prove a very general result (for any distribution of values), but may be slow to converge in practice. For these simulations, we assume that the coordinates values proposed by correct workers follow a normal distribution (see 5.2). In this setting, COMPMED converges much more quickly.<sup>8</sup>

### Description of CWMed

We first define the function *Med*.

Let  $L$  be a list of  $n$  values. Let  $(x_1, \dots, x_n)$  be a list containing the same values as  $L$ , but sorted in increasing order. We define the function  $med(L)$  as follows:

- If  $n$  is even,  $med(L) = \frac{x_{\frac{n}{2}} + x_{\frac{n}{2}+1}}{2}$ .
- If  $n$  is odd,  $med(L) = x_{\frac{n+1}{2}}$ .

Let  $(V_1, \dots, V_n)$  be  $n$  vectors. Let  $v(i, j)$  be the  $j^{th}$  coordinate of  $V_i$ . Let  $C_j = (v(1, j), v(2, j), \dots, v(n, j))$ .

We define  $Med(V_1, V_2, \dots, V_n)$  as follows:

$$Med(V_1, V_2, \dots, V_n) = (med(C_1), med(C_2), \dots, med(C_d)).$$

We now describe the CWMED aggregation scheme.

Let  $\alpha > 0$  be an arbitrarily small constant. At each step:

1. The parameter server generates  $n$  random picks  $(Z_1, \dots, Z_N)$ .
2.  $\forall i \in \{1, \dots, n\}$ , the parameter server sends  $Z_i$  and the current vector of parameters  $P$  to worker  $w_i$ .
3.  $\forall i \in \{1, \dots, n\}$ , let  $\Omega_i$  be the set containing the elements  $X_j$  of the dataset such that  $j \in Z_i$ . Each worker  $w_i$  computes  $\nabla L^*(P, \Omega_i)$ , and sends it to the parameter server.
4.  $\forall i \in \{1, \dots, n\}$ , let  $V_i$  be the vector sent by worker  $w_i$ .<sup>9</sup> The parameter server aggregates the received vectors as follows:

<sup>8</sup> This is due to the fact that COMPASS computes a mean of several vectors, some of which being potentially Byzantine. Therefore, the size  $N$  of the groups of workers must be large enough to ensure that all these vectors are correct with a very high probability (the *Cut* function takes care of the extremely unlikely bad cases).

Here, we assume that the coordinate values proposed by correct workers follow a normal distribution, which means that their expected median value is equal to their expected mean value. Therefore, we can use CWMED, which also excludes extreme values. However, in the general case, the expected median value of a distribution is not always equal to its expected mean value. This is why we used COMPASS to prove the main theoretical result.

Note that this problem (of the expected median value now always being equal to the expected mean value) is a theoretical limitation of both CWMED and COMPMED. Therefore, the comparison we make here is fair with regards to this particular aspect.

<sup>9</sup> If a worker does not send any vector before the end of the round, we consider that it sent a null vector  $(0, 0, \dots, 0)$ .

## 4:12 Arbitrarily Accurate Aggregation Scheme for Byzantine SGD

$$A_n = \text{Med}(V_1, V_2, \dots, V_n)$$

...and uses it to update the model ( $P \leftarrow P - \alpha A_n$ ).

### Description of CompMed

The COMPMED aggregation scheme is defined similarly to the COMPASS aggregation scheme, except that the aggregated vector is now defined as follows:

$$A_n = \text{Med}(\text{Maj}(S_1), \text{Maj}(S_2), \dots, \text{Maj}(S_n)).$$

## 5.2 Simulation setting

Let  $\sigma > 0$  be a positive constants. Let  $G = (g_1, \dots, g_d) = \nabla L(P)$  be the gradient of the loss function.

Let  $\Omega^*$  be a set of  $M$  random elements of the dataset. We assume that  $L^*(P, \Omega^*) = (g_1^*, g_2^*, \dots, g_d^*)$  (i.e., the approximation of the gradient that each correct worker computes) follows a normal distribution centered on the true gradient, that is:  $\forall j \in \{1, \dots, d\}$ ,  $g_j$  follows the normal distribution  $\mathcal{N}(g_j, \sigma^2)$ . This assumption is backed by recent results in machine learning [13]: many normally distributed datasets result in normally distributed gradients.

### Aggregation error

To measure the quality of an aggregation scheme  $A_n$  (for a given number of workers  $n$ ), we define the *aggregation error*  $\lambda_n$  as follows:

$$\lambda_n = \frac{\mathbb{E}[\|A_n - G\|^2]}{d}.$$

This quantity measures the average distance between  $A_n$  and  $G$ , with regards to the randomness of our model. Dividing by the dimension  $d$  (which is a constant of the problem) enables to significantly simplify the simulations, as shown in Section 5.3.

### Attack model

Let  $f < \frac{1}{2}$  be the fraction of Byzantine workers. We assume that all Byzantine workers send the vector  $V_B = (\omega, \omega, \dots, \omega)$  to the parameter server, where  $\omega$  is an arbitrarily large positive constant.

For CWMED, this attack has a maximal impact: it “pushes” the median values of coordinates as far a possible from the value they would have had otherwise. The same is true for COMPMED: if some groups of workers contain a majority of Byzantine workers, their output will be  $V_B$ .

## 5.3 Making the simulations simpler and more general

Let us show that, for CWMED and COMPMED, the aggregation error  $\lambda_n$  can actually be computed without choosing specific values for  $d$  and  $(g_1, \dots, g_d)$ . Besides simplifying the simulations, this makes the simulation results more general (i.e., not dependent on  $d$  and  $(g_1, \dots, g_d)$ ). Therefore, the only parameters of the simulations (defined above) are:  $\sigma$ ,  $f$  and  $\omega$ .

In the following, we explain how to compute two metrics  $\beta_n$  and  $\gamma_n$ , that do not depend on  $d$  or  $(g_1, \dots, g_d)$ . Then, in Theorem 5 and 6, we show that  $\lambda_n = \beta_n$  (for CWMED) and  $\lambda_n = \gamma_n$  (for COMPMED).

### Definition of $\beta_n$

Let  $k$  be the largest integer such that  $k \leq fn$ . Let  $L = (y_1, \dots, y_n)$  be a list of  $n$  values, such that:

- $\forall i \in \{1, \dots, n - k\}$ ,  $y_i$  is a random value following the normal distribution  $\mathcal{N}(0, \sigma^2)$ ;
- $\forall i \in \{n - k + 1, \dots, n\}$ ,  $y_i = \omega$ .

We define  $\beta_n$  as follows:  $\beta_n = \mathbb{E}[\text{med}(L)^2]$ .

### Definition of $\gamma_n$

For a given step of COMPMED, among the  $N$  sets of workers  $(W_1, \dots, W_N)$ , let  $K$  be the number of sets that do *not* contain a strict majority of correct workers.

Let  $L' = (y_1, \dots, y_N)$  be a list of  $N$  values, such that:

- $\forall i \in \{1, \dots, N - K\}$ ,  $y_i$  is a random value following the normal distribution  $\mathcal{N}(0, \sigma^2)$ ;
- $\forall i \in \{N - K + 1, \dots, N\}$ ,  $y_i = \omega$ .

We define  $\gamma_n$  as follows:  $\gamma_n = \mathbb{E}[\text{med}(L')^2]$ .<sup>10</sup>

► **Theorem 5.** For CWMED,  $\lambda_n = \beta_n$ .

► **Theorem 6.** For COMPMED,  $\lambda_n = \gamma_n$ .

The proofs of Theorem 5 and Theorem 6 can be found in the appendix.

## 5.4 Simulation results

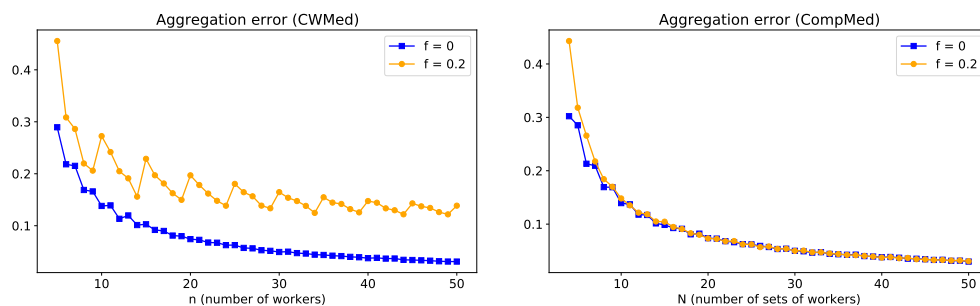
The parameters of the simulations are  $\sigma = 1$  and  $\omega = 10^5$ . The code used for simulations can be found in [16].

We simulated the evolution of the aggregation error  $\lambda_n$  as a function of the number of workers, for both CWMED and COMPMED. The results are presented in Figure 1.

For  $f = 0$ , the aggregation error converges to 0 for both aggregation schemes. We now consider the case  $f = 0.2$  (i.e., 20% of Byzantine workers). For CWMED, the aggregation error converges to a value close to 0.12 (the irregularities of the plot are due to the fact than one new Byzantine worker is added for every 5 new workers). For COMPMED, the aggregation error quickly becomes indistinguishable from the case  $f = 0$  (i.e., it converges to 0).

This illustrates the AAA property of our aggregation scheme: the aggregation error converges to 0 when the number of workers increases, despite a constant fraction of Byzantine workers (which is not the case for existing aggregation schemes, e.g. CWMED).

<sup>10</sup>Note that here, the randomness comes from the values  $y_i$ , but also from  $K$ .



■ **Figure 1** Evolution of the aggregation error for CWMed (left side) and COMPMed (right side), as a function of  $n$  (number of workers) and  $N$  (number of sets of workers) respectively, for  $f = 0$  and  $f = 0.2$ . As a reminder,  $N$  is the largest integer such that  $N \leq n^2$ , where  $n$  is the number of workers.

## 6 Conclusion

In this paper, we presented the first aggregation scheme with the AAA property, and proved its correctness. We illustrated this property with simulations, and compared it to an existing scheme.

The goal of this work was to show that it was possible to have an aggregation error converging to 0 (when  $n$  increases) in the presence of Byzantine workers. For future works, an interesting question would be: how *fast* can it converge to zero? The challenge would be to design an aggregation scheme ensuring a faster convergence, both in theory and in simulations.

---

## References

- 1 Dan Alistarh, Zeyuan Allen-Zhu, and Jerry Li. Byzantine stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 4613–4623, 2018.
- 2 Gilad Baruch, Moran Baruch, and Yoav Goldberg. A little is enough: Circumventing defenses for distributed learning. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d’Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *NeurIPS 2019*, pages 8632–8642, 2019. URL: <https://proceedings.neurips.cc/paper/2019/hash/ec1c59141046cd1866bbbcdfb6ae31d4-Abstract.html>.
- 3 Peva Blanchard, El Mahdi El Mhamdi, Rachid Guerraoui, and Julien Stainer. Machine learning with adversaries: Byzantine tolerant gradient descent. In *Advances in Neural Information Processing Systems 30*, pages 119–129. Curran Associates, Inc., 2017.
- 4 Amine Boussetta, El-Mahdi El-Mhamdi, Rachid Guerraoui, Alexandre Maurer, and Sébastien Rouault. AKSEL: Fast Byzantine SGD. In *24th International Conference on Principles of Distributed Systems (OPODIS 2020)*, 2021.
- 5 Saikiran Bulusu, Prashant Khanduri, Pranay Sharma, and Pramod K. Varshney. On distributed stochastic gradient descent for nonconvex functions in the presence of byzantines. In *2020 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2020, Barcelona, Spain, May 4-8, 2020*, pages 3137–3141. IEEE, 2020. doi:10.1109/ICASSP40776.2020.9052956.
- 6 Lingjiao Chen, H. Wang, Zachary B. Charles, and Dimitris Papailiopoulos. Draco: Byzantine-resilient distributed training via redundant gradients. In *ICML*, 2018.
- 7 Yudong Chen, Lili Su, and Jiaming Xu. Distributed statistical machine learning in adversarial settings: Byzantine gradient descent. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 1(2):44, 2017.

- 8 Anna Choromanska, Mikael Henaff, Michaël Mathieu, Gérard Ben Arous, and Yann LeCun. The loss surfaces of multilayer networks. In *Proceedings of the Eighteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2015, San Diego, California, USA, May 9-12, 2015*, 2015. URL: <http://proceedings.mlr.press/v38/choromanska15.html>.
- 9 Georgios Damaskinos, El Mahdi El Mhamdi, Rachid Guerraoui, Arsany Guirguis, and Sébastien Rouault. Aggregathor: Byzantine machine learning via robust gradient aggregation. In *SysML*, 2019.
- 10 Georgios Damaskinos, El Mahdi El Mhamdi, Rachid Guerraoui, Rhicheek Patra, Mahsa Taziki, et al. Asynchronous byzantine machine learning (the case of sgd). In *ICML*, pages 1153–1162, 2018.
- 11 El-Mahdi El-Mhamdi, Rachid Guerraoui, Arsany Guirguis, and Lê Nguyễn Hoàng. Geniunely distributed byzantine machine learning. In *PODC*, 2020.
- 12 El Mahdi El Mhamdi, Rachid Guerraoui, and Sébastien Rouault. The hidden vulnerability of distributed learning in Byzantium. In *Proceedings of the 35th International Conference on Machine Learning*, pages 3521–3530. PMLR, 2018.
- 13 Arthur Jacot, Clément Hongler, and Franck Gabriel. Neural tangent kernel: Convergence and generalization in neural networks. In *NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 8580–8589, 2018. URL: <https://proceedings.neurips.cc/paper/2018/hash/5a4be1fa34e62bb8a6ec6b91d2462f5a-Abstract.html>.
- 14 Kishori M. Konwar, Sanguthevar Rajasekaran, and Alexander A. Shvartsman. Robust network supercomputing with malicious processes. In Shlomi Dolev, editor, *Distributed Computing, 20th International Symposium, DISC 2006, Stockholm, Sweden, September 18-20, 2006, Proceedings*, volume 4167 of *Lecture Notes in Computer Science*, pages 474–488. Springer, 2006. doi:10.1007/11864219\_33.
- 15 Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.
- 16 Alexandre Maurer. Source code for simulations related to this paper. URL: <https://tinyurl.com/sim-aaa-paper>.
- 17 David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.
- 18 Lili Su and Nitin H. Vaidya. Fault-tolerant multi-agent optimization: Optimal iterative distributed algorithms. In George Giakkoupis, editor, *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, July 25-28, 2016*, pages 425–434. ACM, 2016. doi:10.1145/2933057.2933105.
- 19 Cong Xie, Oluwasanmi Koyejo, and Indranil Gupta. Generalized byzantine-tolerant sgd, 2018.
- 20 Cong Xie, Oluwasanmi Koyejo, and Indranil Gupta. Phocas: dimensional byzantine-resilient stochastic gradient descent, 2018.
- 21 Dong Yin, Yudong Chen, Ramchandran Kannan, and Peter Bartlett. Byzantine-robust distributed learning: Towards optimal statistical rates. In *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 5650–5659. PMLR, 2018.

## A Appendix

**Proof of Lemma 2.** Let  $p = \frac{2f+1}{4}$ . Let us describe 4 ways to select some Byzantine workers among  $N^2$  workers, that we call “games”.

- **Game A:**  $k$  workers are selected randomly, and then turned Byzantine.
- **Game B:** Each worker is turned Byzantine with probability  $p$ .
- **Game C:** Game B is executed. Then, if the number of Byzantine workers is  $k$  or less: all workers are turned Byzantine.

## 4:16 Arbitrarily Accurate Aggregation Scheme for Byzantine SGD

- **Game D:** Game C is executed. Then, we randomly pick one Byzantine worker, make it correct again, and repeat the process until we have exactly  $k$  Byzantine workers.

Let  $\Phi_X$  be the event: “After Game X, all  $N$  sets contain a strict minority of Byzantine workers.” As Game D consists in executing Game C, then only removing Byzantine workers, we have:  $P(\Phi_D) \geq P(\Phi_C)$ .

Then, we can notice that Game D is equivalent to Game A (since each worker is equally likely to end up Byzantine). Therefore,  $P(\Phi_A) = P(\Phi_D) \geq P(\Phi_C)$ . Now, let us give a lower bound of  $P(\Phi_C)$ .

Let  $\Psi_B$  be the following event: “After Game B, there are strictly more than  $k$  Byzantine workers”. Then, we can notice that, for  $\Phi_C$  to be true, it is necessary that both  $\Phi_B$  and  $\Psi_B$  are true. Indeed, if  $\Psi_B$  is false,  $\Phi_C$  cannot be true, because all workers would then be turned Byzantine in Game C (just after executing Game B). And, if  $\Psi_B$  is true but  $\Phi_B$  is false,  $\Phi_C$  cannot be true, because we would not have a strict minority of Byzantine workers in all  $N$  sets. Therefore,  $P(\Phi_C) \geq P(\Phi_B \wedge \Psi_B)$ .

Now, notice that  $P(\Psi_B) = P(\Phi_B \wedge \Psi_B) + P(\neg\Phi_B \wedge \Psi_B)$ . Since  $P(\neg\Phi_B \wedge \Psi_B) \leq P(\neg\Phi_B) = 1 - P(\Phi_B)$ , we have:  $P(\Psi_B) \leq P(\Phi_B \wedge \Psi_B) + 1 - P(\Phi_B)$ , and  $P(\Phi_B \wedge \Psi_B) \geq P(\Phi_B) + P(\Psi_B) - 1$ .

Before going further, let us give a lower bound of  $P(\Psi_B)$ . Let  $N'_0$  be the smallest integer such that,  $\forall N \geq N'_0$ :

$$\frac{\ln N^2}{N^2} < (p - f)^2$$

Let  $k'$  be the number of Byzantine workers after Game B. According to Hoeffding’s inequality, applied to the  $N^2$  workers:

$$P\left(\left|\frac{k'}{N^2} - p\right| < \sqrt{\frac{\ln N^2}{N^2}}\right) \geq 1 - \frac{2}{N^4}$$

Thus,  $\forall N \geq N'_0$ :

$$P\left(\left|\frac{k'}{N^2} - p\right| < p - f\right) \geq 1 - \frac{2}{N^4}$$

Now, we can remark that:

$$P\left(\left|\frac{k'}{N^2} - p\right| < p - f\right) \leq P\left(-\frac{k'}{N^2} + p < p - f\right) = P(k' > fN^2)$$

Thus,  $\forall N \geq N'_0$ :

$$P(\Psi_B) = P(k' > k) \geq P(k' > fN^2) \geq 1 - \frac{2}{N^4}$$

Thus, according to Lemma 1,  $\forall N \geq \max(N_0, N'_0)$ :

$$P(\Phi_B \wedge \Psi_B) \geq \left(1 - \frac{2}{N}\right) + \left(1 - \frac{2}{N^4}\right) - 1 = 1 - \frac{2}{N} - \frac{2}{N^4}$$

Therefore, we have:

$$P(E'_N) = P(\Phi_A) = P(\Phi_D) \geq P(\Phi_C) \geq P(\Phi_B \wedge \Psi_B) \geq 1 - \frac{2}{N} - \frac{2}{N^4}$$

Let  $N_1$  be such that  $N_1 \geq \max(N_0, N'_0)$  and,  $\forall N \geq N_1$ ,  $\frac{2}{N^4} \leq \frac{1}{N}$ . Then, we have:

$$P(E'_N) \geq 1 - \frac{3}{N} \quad \blacktriangleleft$$

**Proof of Theorem 5.** Let  $j \in \{1, \dots, d\}$ , and let  $L^*$  be a list defined similarly to  $L$ , except that we replace  $\mathcal{N}(0, \sigma^2)$  by  $\mathcal{N}(g_j, \sigma^2)$ . Note that this is equivalent to adding  $g_j$  to each value of  $L$ .

Let us call  $a_j$  the  $j^{\text{th}}$  coordinate of the aggregated vector  $A_n$ . Then:

$$\mathbb{E}[(a_j - g_j)^2] = \mathbb{E}[(\text{med}(L^*) - g_j)^2] = \mathbb{E}[\text{med}(L)^2] = \beta_n$$

Therefore,  $\forall j \in \{1, \dots, d\}$ ,  $\mathbb{E}[(a_j - g_j)^2] = \beta_n$ , and:

$$\lambda_n = \frac{\mathbb{E}[\|A_n - G\|^2]}{d} = \frac{\sum_{j=1}^d \mathbb{E}[(a_j - g_j)^2]}{d} = \frac{d\beta_n}{d} = \beta_n \quad \blacktriangleleft$$

**Proof of Theorem 6.** Let  $(W_1, \dots, W_N)$  be the  $N$  sets of workers chosen at each step of COMP MED. Let  $S_i$  be the set of vectors sent by the workers of  $W_i$ . Let  $\text{Maj}(S_i) = (h_1, h_2, \dots, h_d)$ .

If  $W_i$  contains a strict majority of correct workers, then,  $\forall j \in \{1, \dots, d\}$ ,  $h_j$  follows the normal distribution  $\mathcal{N}(g_j, \sigma^2)$ . Otherwise,  $\forall j \in \{1, \dots, d\}$ ,  $h_j = \omega$ .

Let  $K$  be the number of sets of workers  $W_i$  that do *not* contain a strict majority of correct workers. Then, the rest of the proof is identical to the proof of Theorem 5, if we replace  $L$  by  $L'$  (that is, replacing  $n$  by  $N$  and  $k$  by  $K$ ). Thus, the result.  $\blacktriangleleft$





# Good-Case and Bad-Case Latency of Unauthenticated Byzantine Broadcast: A Complete Categorization

Ittai Abraham ✉

VMware Research, Herzliya, Israel

Ling Ren ✉

University of Illinois at Urbana-Champaign, IL, USA

Zhuolun Xiang ✉

University of Illinois at Urbana-Champaign, IL, USA

---

## Abstract

This paper studies the *good-case latency* of *unauthenticated* Byzantine fault-tolerant broadcast, which measures the time it takes for all non-faulty parties to commit given a non-faulty broadcaster. For both asynchrony and synchrony, we show that  $n \geq 4f$  is the tight resilience threshold that separates good-case 2 rounds and 3 rounds. For asynchronous Byzantine reliable broadcast (BRB), we also investigate the *bad-case latency* for all non-faulty parties to commit when the broadcaster is faulty but some non-faulty party commits. We provide matching upper and lower bounds on the resilience threshold of bad-case latency for BRB protocols with optimal good-case latency of 2 rounds. In particular, we show 2 impossibility results and propose 4 asynchronous BRB protocols.

**2012 ACM Subject Classification** Theory of computation → Distributed algorithms; Security and privacy → Distributed systems security

**Keywords and phrases** Byzantine broadcast, asynchrony, synchrony, latency, good-case, optimal

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2021.5

**Acknowledgements** The authors would like to thank Kartik Nayak for helpful discussions related to the paper.

## 1 Introduction

Byzantine fault-tolerant broadcast is a fundamental primitive in distributed systems, where a designated broadcaster sends its value to all parties, such that all non-faulty parties commit on the same value despite arbitrary deviation from Byzantine parties. Moreover, if the broadcaster is non-faulty, then all honest parties are required to commit the same value as the broadcaster's input. Byzantine broadcast (BB) requires all non-faulty parties to eventually commit, while Byzantine reliable broadcast (BRB) relaxes the condition to only require termination when the broadcaster is honest or if some non-faulty party terminates. When the network is asynchronous, meaning the message delays are unbounded, it is well-known that BB is unsolvable with even a single fault. On the other hand, BRB is solvable under asynchrony as long as there are  $n \geq 3f + 1$  parties.

Recent work of Abraham et al. [4] investigates the notion of good-case latency of Byzantine fault-tolerant broadcast, which is the time for all honest parties to commit given that the broadcaster is honest. Theoretically, the good-case latency is a natural and interesting metric that has not been formally studied by the literature until recently; Practically, for applications like leader-based Byzantine fault-tolerant state machine replication (BFT SMR), the good-case latency study answers the fundamental question of how fast can leader-based



© Ittai Abraham, Ling Ren, and Zhuolun Xiang;  
licensed under Creative Commons License CC-BY 4.0

25th International Conference on Principles of Distributed Systems (OPODIS 2021).

Editors: Quentin Bramas, Vincent Gramoli, and Alessia Milani; Article No. 5; pp. 5:1–5:20

Leibniz International Proceedings in Informatics



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

■ **Table 1** Upper and lower bounds for good-case latency of *unauthenticated* Byzantine fault-tolerant broadcast.

Problem	Timing Model	Resilience	Lower Bound	Upper Bound
BRB	Asynchrony	$n \geq 4f$	2 rounds [4]	<b>2 rounds</b> (Thm 9)
		$3f + 1 \leq n \leq 4f - 1$	<b>3 rounds</b> (Thm 10)	3 rounds [5]
BB	Synchrony	$n \geq 4f$	$2\delta$ [4, 9]	<b><math>2\delta</math></b> (Thm 18)
		$3f + 1 \leq n \leq 4f - 1$	<b><math>3\delta</math></b> (Thm 17)	$3\delta$ (Thm 19) [5]

■ **Table 2** Comparison of our results and previous results of asynchronous *unauthenticated* Byzantine reliable broadcast.

Result	Resilience	Good-case	Bad-case	Comm. cost	Reference
Bracha	$n \geq 3f + 1$	3 rounds	4 rounds	$O(n^2)$	[5]
Imbs and Raynal	$n \geq 5f + 1$	2 rounds	3 rounds	$O(n^2)$	[12]
Impossibility of (2, 2)	$f \geq 2$	2 rounds	2 rounds	–	Thm 11
F1-BRB	$n \geq 4f, f = 1$	2 rounds	2 rounds	$O(n^2)$	Thm 12
Impossibility of (2, 3)	$n \leq 5f - 2, f \geq 3$	2 rounds	3 rounds	–	Thm 13
F2-BRB	$n \geq 4f, f = 2$	2 rounds	3 rounds	$O(n^3)$	Thm 15
(2, 4)-BRB	$n \geq 4f$	2 rounds	4 rounds	$O(n^2)$	Thm 9
(2, 3)-BRB	$n \geq 5f - 1$	2 rounds	3 rounds	$O(n^2)$	Thm 16

BFT SMR commit decisions during the steady state when the leader is non-faulty. Moreover, for asynchronous Byzantine reliable broadcast, good-case latency is particularly important since BRB may not terminate under a Byzantine leader.

The work of Abraham et al. [4] reveals a surprisingly rich structure in the good-case latency tight bounds for *authenticated* Byzantine broadcast, where digital signatures are used and the adversary is assumed to be computationally bounded. In this work, we study the good-case latency and bad-case latency of *unauthenticated* Byzantine fault-tolerant broadcast. Our results are summarized in Table 1 and 2.

**Complete categorization for good-case latency under asynchrony and synchrony.** Under asynchrony when the message delays are unbounded, we show that  $n \geq 4f$  is the tight resilience threshold that separates good-case latency of 2 rounds and 3 rounds. For  $n \geq 4f$ , [4] shows a 2-round lower bound, and we present a protocol with good-case latency of 2 rounds. For  $3f + 1 \leq n \leq 4f - 1$ , Bracha’s reliable broadcast [5] has good-case latency of 3 rounds, and we prove a matching 3-round lower bound.

► **Theorem 1** (Informal; tight bounds on good-case latency in asynchrony). *For unauthenticated Byzantine reliable broadcast with  $f$  Byzantine parties under asynchrony, in the good-case:*

1. 2 rounds are necessary and sufficient if  $n \geq 4f$  (Section 3.1), and
2. 3 rounds are necessary and sufficient if  $3f + 1 \leq n < 4f$  (Section 3.2).

The above asynchronous good-case latency bounds also imply similar results for good-case latency of BB and BRB under synchrony as well. Let  $\delta$  denote the actual message delay bound during the execution (see Section 5 for details). For  $n \geq 3f + 1$ , [4] shows a  $2\delta$  lower bound (also implied by the early-stopping results [9]), and we present a synchronous BB protocol with good-case latency of  $2\delta$  under  $n \geq 4f$ , inspired by our 2-round asynchronous

BRB protocol. For  $3f + 1 \leq n \leq 4f - 1$ , we show a synchronous BB protocol that has good-case latency of  $3\delta$  inspired by Bracha's reliable broadcast [5], and the aforementioned  $3\delta$  lower bound also applies to synchrony.

► **Theorem 2** (Informal; tight bounds on good-case latency in synchrony). *For unauthenticated Byzantine broadcast and Byzantine reliable broadcast with  $f$  Byzantine parties under synchrony (message delay bounded by  $\delta$ ), in the good-case:*

1.  $2\delta$  are necessary and sufficient if  $n \geq 4f$  (Section 5), and
2.  $3\delta$  are necessary and sufficient if  $3f + 1 \leq n < 4f$  (Section 5).

### Complete categorization for bad-case latency of asynchronous Byzantine reliable broadcast.

In addition to the good-case commit path, asynchronous BRB protocols usually have a second commit path to ensure all honest parties eventually commit, when the Byzantine broadcaster and Byzantine parties deliberately make only a few honest parties commit in the good-case commit path. We use *bad-case latency* to denote the latency of such second commit path, and say a BRB protocol is  $(R_g, R_b)$ -round if it has good-case latency of  $R_g$  rounds and bad-case latency of  $R_b$  rounds. For instance, Bracha's reliable broadcast [5] is  $(3, 4)$ -round.

We provide a complete categorization of the threshold resilience for BRB with good-case latency of 2. We show two lower bound results on the resilience threshold: for  $(2, 2)$  and for  $(2, 3)$ . We also show 4 protocols with matching resilience bounds: these protocols have the optimal good-case latency of 2 rounds, but with different trade-offs in resilience and bad-case latency, matching the lower bound results. As summarized in Table 2, prior upper bound results include Bracha's  $(3, 4)$ -round BRB for  $n \geq 3f + 1$ , and the  $(2, 3)$ -round BRB for  $n \geq 5f + 1$  by Imbs and Raynal [12].

- First, we show it is impossible to achieve  $(2, 2)$ -round BRB, except for the special case of  $f = 1$  where we propose a protocol F1-BRB that has  $(2, 2)$ -round and optimal resilience  $n \geq 4f$ .
- Next, we show another impossibility result stating that no BRB protocol can achieve  $(2, 3)$ -round under  $n \leq 5f - 2$  for  $f \geq 3$ . That is, for  $f \geq 3$ , no BRB protocol can have optimality in all three metrics: good-case latency, bad-case latency and resilience. For the special case of  $f = 2$ , we propose a protocol F2-BRB that has  $(2, 3)$ -round and optimal resilience  $n \geq 4f$ . For the general case of  $f \geq 3$ , we have two protocols – a protocol named  $(2, 4)$ -BRB under  $n \geq 4f$  that has  $(2, 4)$ -round, and a protocol named  $(2, 3)$ -BRB which improves the resilience of Imbs and Raynal [12] from  $n \geq 5f + 1$  to  $n \geq 5f - 1$  while keeping the protocol  $(2, 3)$ -round. Both  $(2, 4)$ -BRB and  $(2, 3)$ -BRB have tight resilience and latencies due to the impossibility result.

## 2 Preliminaries

**Model of execution.** We define a protocol for a set of  $n$  parties, among which at most  $f$  are Byzantine faulty and can behave arbitrarily and has unbounded computational power. If a party remains non-faulty for the entire protocol execution, we call the party honest. During an execution  $E$  of a protocol, parties perform sequences of events, including *send*, *receive/deliver*, *local computation*.

In this paper, we investigate results for deterministic unauthenticated protocols. If the protocol is deterministic, for any two executions, if an honest party has the same initial state and receives the same set of messages at the same corresponding time points (by its local clock), the honest party cannot distinguish two executions. We will use the standard indistinguishability argument to prove lower bounds.

## 5:4 Good-Case and Bad-Case Latency of Unauthenticated Byzantine Broadcast

We consider both synchronous and asynchronous network models. Under synchrony, any message between two honest parties will be delivered within  $\delta$  time during the execution. More details about the synchrony model assumption is deferred to Section 5. Under asynchrony, the adversary can control the message delay of any message to be an arbitrary non-negative value. We assume all-to-all, reliable and authenticated communication channels, such that the adversary cannot fake, modify or drop the messages sent by honest parties.

**Byzantine broadcast variants.** We investigate two standard variants of Byzantine broadcast problem for synchrony and asynchrony.

► **Definition 3** (Byzantine Broadcast (BB)). *A Byzantine broadcast protocol must satisfy the following properties.*

- *Agreement.* If two honest parties commit values  $v$  and  $v'$  respectively, then  $v = v'$ .
- *Validity.* If the designated broadcaster is honest, then all honest parties commit the broadcaster's value and terminate.
- *Termination.* All honest parties commit and terminate.

► **Definition 4** (Byzantine Reliable Broadcast (BRB)). *A Byzantine reliable broadcast protocol must satisfy the following properties.*

- *Agreement.* Same as above.
- *Validity.* Same as above.
- *Termination.* If an honest party commits a value and terminates, then all honest parties commit a value and terminate.

We will also use *Byzantine agreement* as a primitive to simplify the construction of our BB protocols under synchrony in Section 5. The Byzantine agreement gives each party an input, and its validity requires that if all honest parties have the same input value, then all honest parties commit that value.

**Good-case latency of broadcast.** Depending on the network model, the measurement of latency is different. Under synchrony, we can measure the latency using the physical clock time.

► **Definition 5** (Good-case Latency under Synchrony [4]). *A Byzantine broadcast (or Byzantine reliable broadcast) protocol has good-case latency of  $T$  under synchrony, if all honest parties commit within time  $T$  since the broadcaster starts the protocol (over all executions and adversarial strategies), given the designated broadcaster is honest.*

Under asynchrony, the network delay is unbounded. To measure the latency of asynchronous protocols, we use the natural notion of *asynchronous rounds* from the literature [6], where a protocol runs in  $R$  asynchronous rounds if its running time is at most  $R$  times the maximum message delay between honest parties during the execution.

► **Definition 6** (Good-case Latency under Asynchrony [4]). *A Byzantine reliable broadcast protocol has good-case latency of  $R$  rounds under asynchrony, if all honest parties commit within asynchronous round  $R$  (over all executions and adversarial strategies), given the designated broadcaster is honest.*

When the broadcaster is dishonest, Byzantine broadcast will have worst-case latency of  $f + 1$  rounds [11], and for Byzantine reliable broadcast by definition it does not guarantee termination (the broadcaster can just remain silent). Therefore, the notion of good-case

latency is the natural metric to measure the latency performance of reliable broadcast. Another important latency metric for reliable broadcast is to measure how fast can all honest parties commit, once an honest party commit. We formally define it as the bad-case latency as below.

► **Definition 7** (Bad-case Latency under Asynchrony). *A Byzantine reliable broadcast protocol has bad-case latency of  $R' = R + R_{ex}$  rounds under asynchrony, if all honest parties commit within  $R_{ex}$  asynchronous round after an honest party commits (over all executions and adversarial strategies), and the good-case latency of the protocol is  $R$ .*

We will use the notation  $(R_g, R_b)$ -round BRB to denote an authenticated Byzantine reliable broadcast protocol that has good-case latency of  $R_g$  rounds and bad-case latency of  $R_b$  rounds. For instance, the classic Bracha reliable broadcast [5] has a good-case latency of 3 rounds and a bad-case latency of 4 rounds ( $R_{ex} = 1$ ), under  $n \geq 3f + 1$  parties; it is thus a  $(3, 4)$ -round BRB.

### 3 Good-case Latency of Asynchronous Byzantine Reliable Broadcast

Under asynchrony, Byzantine reliable broadcast is solvable if and only if  $n \geq 3f + 1$ . We show the *tight* lower and upper bound on the good-case latency of asynchronous unauthenticated BRB is 2 rounds when  $n \geq 4f$ , and 3 rounds when  $3f + 1 \leq n \leq 4f - 1$ .

#### 3.1 2-round Unauthenticated BRB under $n \geq 4f$

We show the tightness of the bound by presenting a 2-round unauthenticated BRB protocol, which has good-case latency of 2 rounds and bad-case latency of 4 rounds with  $n \geq 4f$  parties, as presented in Figure 1.

In the protocol, in the first round the broadcaster sends its proposal to all parties. Then in the second round, all parties send an `ack` for the first proposal received. Parties commit in 2 rounds when receiving  $n - f - 1$  `ack` for the same value from distinct parties other than the broadcaster, which will happen when the broadcaster is honest. To ensure termination, the protocol has another 4-round commit path, to guarantee that all honest parties will commit even if the Byzantine parties deliberately make only a few honest parties commit in round 2. The 4-round commit path consists of a Bracha-style reliable broadcast, where the parties send `vote-1` and `vote-2` messages upon receiving enough messages as specified in Step 4. Finally, when receiving enough `vote-2` messages, party can also commit in round 4.

► **Lemma 8.** *If an honest party commits  $v$  at Step 3, then no honest party will send `vote-1` or `vote-2` for any other value  $v' \neq v$ .*

**Proof.** Since the honest party commit  $v$  at Step 3, it receives  $n - f - 1$  `ack` messages for  $v$  from distinct non-broadcaster parties. If the broadcaster is honest, then no honest party will send `vote-1` or `vote-2` message for  $v'$  since there are at most  $f$  Byzantine parties. If the broadcaster is Byzantine, and suppose there are  $t$  Byzantine parties, then there are at most  $t - 1$  Byzantine parties among all non-broadcaster parties, and there must be at least  $(n - f - 1) - (t - 1) = n - f - t$  honest parties sending `ack` for  $v$ . Suppose an honest party receives  $n - 2f$  `ack` messages for  $v'$  from distinct non-broadcaster parties, then there must be at least  $(n - 2f) - (t - 1) = n - 2f - t + 1$  honest parties sending `ack` for  $v'$ . Since there are only  $n - t$  honest parties, there must be at least  $(n - f - t) + (n - 2f - t + 1) - (n - t) = n - 3f - t + 1 \geq n - 4f + 1 \geq 1$  honest party that sends `ack` for both  $v$  and  $v'$ , contradiction. Hence no honest party can receive  $n - 2f$

1. **Propose.** The designated broadcaster  $L$  with input  $v$  sends  $\langle \text{propose}, v \rangle$  to all parties.
2. **Ack.** When receiving the first proposal  $\langle \text{propose}, v \rangle$  from the broadcaster, a party sends an  $\langle \text{ack}, v \rangle$  message to all parties.
3. **2-round Commit.** When receiving  $\langle \text{ack}, v \rangle$  from  $n - f - 1$  distinct non-broadcaster parties, a party commits  $v$ , sends  $\langle \text{vote-1}, v \rangle$  and  $\langle \text{vote-2}, v \rangle$  to all parties, and terminates.
4. **Vote.**
  - When receiving  $\langle \text{ack}, v \rangle$  from  $n - 2f$  distinct non-broadcaster parties, a party sends a  $\langle \text{vote-1}, v \rangle$  message to all parties, if it has not already sent `vote-1` for any value.
  - When receiving  $\langle \text{vote-1}, v \rangle$  from  $n - f - 1$  distinct non-broadcaster parties, a party sends a  $\langle \text{vote-2}, v \rangle$  message to all parties, if it has not already sent `vote-2` for any value.
  - When receiving  $\langle \text{vote-2}, v \rangle$  from  $f + 1$  distinct non-broadcaster parties, a party sends a  $\langle \text{vote-2}, v \rangle$  message to all parties, if it has not already sent `vote-2` for any value.
5. **4-round Commit.** When receiving  $\langle \text{vote-2}, v \rangle$  from  $n - f - 1$  distinct non-broadcaster parties, a party commits  $v$  and terminates.

■ **Figure 1** (2, 4)-round BRB protocol under  $n \geq 4f$ .

`ack` messages for  $v'$ . Moreover, since the thresholds in Step 4 are larger than the number of Byzantine parties, i.e.,  $n - f - 1 \geq 3f - 1 > f$  and  $f + 1 > f$ , no honest party will send `vote-1` or `vote-2` for  $v' \neq v$ . ◀

► **Theorem 9.** *The protocol in Figure 1 solves Byzantine reliable broadcast under asynchrony with optimal resilience  $n \geq 4f$  and optimal good-case latency of 2 rounds, and has bad-case latency of 4 rounds.*

**Proof.**

**Validity and Good-case Latency.** If the broadcaster is honest, it sends the same proposal of value  $v$  to all parties. Then all  $n - f - 1$  non-broadcaster honest parties will multicast the `ack` message for  $v$ . The Byzantine parties cannot make any honest party to send `vote-1`, `vote-2`, for any other value  $v' \neq v$  since  $f$  is below any threshold specified in the protocol. All honest parties will eventually commit  $v$  after receiving  $n - f - 1$  `ack` messages at Step 3 and terminate. The good-case latency is 2 rounds, including broadcaster sending the proposal and all parties sending `ack` message.

**Agreement.** If the broadcaster is honest, by validity all honest parties will commit the same value. Now consider when the broadcaster is Byzantine, there are at most  $f - 1$  Byzantine parties among non-broadcasters.

If any two honest parties commit different values at Step 3, then there must be at least  $n - f - 1 - (f - 1) = n - 2f \geq 2f$  honest parties sending `ack` for each of these different values. It is impossible by quorum intersection since there are only  $3f$  honest parties. Similarly, no two honest parties can commit different values at Step 5.

Now we show that if an honest party  $h1$  commits  $v$  at Step 3 and another honest party  $h2$  commits  $v'$  at Step 5, then it must be  $v = v'$ . Suppose  $h1$  commits  $v$  at Step 3, then by Lemma 8, no honest party will send `vote-1` or `vote-2` for  $v' \neq v$ , and thus not enough `vote-2` for any  $v' \neq v$  to be committed at Step 5. Suppose  $h2$  commit  $v'$  at Step 5, then  $h2$  receives at least  $n - f - 1 - (f - 1) = n - 2f \geq 2f$  `vote-2` messages for  $v'$  from honest parties. By the contrapositive of Lemma 8, no honest party commits  $v \neq v'$  at Step 3.

**Termination and Bad-case Latency.** If the broadcaster is honest, by validity all honest parties will commit the same value. Now consider when the broadcaster is Byzantine, there are at most  $f - 1$  Byzantine parties among non-broadcasters.

Suppose that an honest party commits  $v$  at Step 3, by Lemma 8, no honest party will send `vote-1` or `vote-2` for any  $v' \neq v$ . Since there are at least  $n - f - 1 - (f - 1) = n - 2f$  non-broadcaster honest parties sending `ack` for  $v$ , all honest parties will receive at least  $n - 2f$  `ack` for  $v$  from non-broadcasters, and thus send `vote-1` for  $v$ . Since there are  $n - f$  honest non-broadcasters, all honest parties will send `vote-2` for  $v$ , and then commit after receiving  $n - f - 1$  `vote-2` messages.

Suppose that an honest party commits  $v$  at Step 5, then at least  $n - f - 1 - (f - 1) = n - 2f \geq f + 1$  honest non-broadcasters send `vote-2` for  $v$ . We only need to show that no honest party send `vote-2` for  $v' \neq v$ , then all honest parties will send `vote-2` for  $v$  and thus commit  $v$ . Suppose there is an honest party that send `vote-2` for  $v' \neq v$ , then there exists two sets of `vote-1` messages from  $n - f - 1$  distinct non-broadcasters for  $v$  and  $v'$  respectively. Suppose there are  $t > 0$  Byzantine parties, then at least  $n - f - 1 - (t - 1) \geq n - t - f$  honest parties send `vote-1` for  $v$  and  $v'$  respectively, which is impossible as there are  $n - t < 2(n - t - f)$  honest parties. Therefore no honest party sends `vote-2` for  $v' \neq v$ , and all honest parties commits  $v$ .

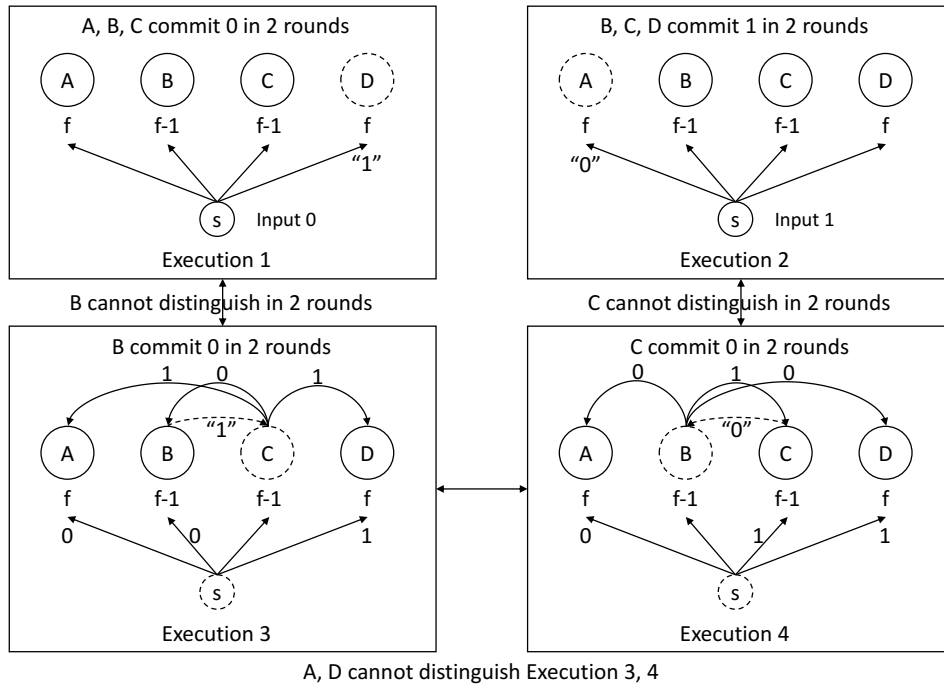
It is clear from the protocol that after at most 2 rounds (`vote-1` and `vote-2`) since any honest party commits, all honest parties also commit. Hence the bad-case latency is 4 rounds.  $\blacktriangleleft$

### 3.2 3-round Lower Bound for Unauthenticated BRB under $n \leq 4f - 1$

► **Theorem 10.** *Any unauthenticated Byzantine reliable broadcast protocol under  $3f + 1 \leq n \leq 4f - 1$  must have a good-case latency of at least 3 rounds even under synchrony.*

**Proof of Theorem 10.** The proof is illustrated in Figure 2. We assume all parties start their protocol at the same time, which strengthens the lower bound result. Under synchrony, any message between all honest parties will be delivered within  $\delta$  time, and hence each round of the protocol is of  $\delta$  time. Without loss of generality, we prove the lower bound for  $n = 4f - 1$ . Suppose there exists a BRB protocol  $\Pi$  that has a good-case latency of 2 round, which means the honest parties can always commit after receiving two rounds of messages but before receiving any message from the third round, if the designated broadcaster is honest. Let party  $s$  be the broadcaster, and divide the remaining  $n - 1 = 4f - 2$  parties into 4 groups  $A, B, C, D$  where  $|A| = |D| = f$  and  $|B| = |C| = f - 1$ . For brevity, we often use  $A (B, C, D)$  to refer all the parties in  $A (B, C, D)$ . Consider the following three executions of  $\Pi$ .

- Execution 1. The broadcaster  $s$  is honest and has input 0. Parties in  $D$  are Byzantine, they behave honestly according to the protocol except that they pretend to receive from a broadcaster whose input is 1. Since the broadcaster is honest, by validity and good-case latency, parties in  $A, B, C$  will commit 0 after receiving two rounds of messages but before receiving any message from the third round.



■ **Figure 2** Unauthenticated BRB Good-case Latency Lower Bound: 3 rounds under  $n = 4f - 1$ . Dotted circles denote Byzantine parties.

- Execution 2. This execution is a symmetric case of Execution 1. The broadcaster  $s$  is honest and has input 1. Parties in  $A$  are Byzantine, they behave honestly according to the protocol except that they pretend to receive from a broadcaster whose input is 0. Since the broadcaster is honest, by validity and good-case latency, parties in  $B, C, D$  will commit 1 after receiving two rounds of messages but before receiving any message from the third round.
- Execution 3. The broadcaster  $s$  and the parties in  $C$  are Byzantine.  $s$  behaves to  $A, B$  identically as in Execution 1 and to  $D$  identically as in Execution 2. Parties in  $C$  behave to  $B$  honestly according to the protocol except that they pretend to receive the same messages from the broadcaster as in Execution 1, and only send messages to  $B$  in the first two rounds. Parties in  $C$  behave to  $A, D$  honestly except that they pretend to receive the same messages from the broadcaster as in Execution 2, and pretend to receive messages from  $B$  as in Execution 2 only in the first two rounds.
- Execution 4. This execution is a symmetric case of Execution 3. The broadcaster  $s$  and the parties in  $B$  are Byzantine.  $s$  behaves to  $A$  identically as in Execution 1 and to  $C, D$  identically as in Execution 2. Parties in  $B$  behave to  $C$  honestly according to the protocol except that they pretend to receive the same messages from the broadcaster as in Execution 2, and only send messages to  $C$  in the first two rounds. Parties in  $B$  behave to  $A, D$  honestly except that they pretend to receive the same messages from the broadcaster as in Execution 1, and pretend to receive messages from  $C$  as in Execution 1 only in the first two rounds.

We show the following indistinguishability and contradiction.



- $B$  cannot distinguish Execution 1 and 3 in the first two rounds, and thus will commit 0 in the end of round 2 in Execution 3. The broadcaster  $s$  behaves to  $B$  identically in both executions. The messages sent to  $B$  in the first round by any non-broadcaster party are identical in Execution 1 and 3, since the first round message only depends on the initial state and all Byzantine parties behave honestly in the first round. For the second round, in Execution 3, since parties in  $D$  pretends to  $B$  that it receives messages from the broadcaster with input 1, and parties in  $C$  pretends to  $B$  that it receives the same messages from the broadcaster as in Execution 1, the parties in  $B$  observe the same messages in the first two rounds of both executions. Hence,  $B$  cannot distinguish Execution 1 and 3 in the first two rounds. Since  $B$  commit 0 in the end of round 2 in Execution 1 due to validity and good-case latency,  $B$  also commit 0 in the end of round 2 in Execution 3.
- Similarly,  $C$  cannot distinguish Execution 2 and 4 in the first two rounds, and thus will commit 1 in the end of round 2 in Execution 4.
- $A, D$  cannot distinguish Execution 3 and 4. Similarly, the messages sent to  $A, D$  in the first round are identical in both executions. The broadcaster  $s$  behaves to  $A, B$  identically in Execution 3 and 4 as in Execution 1, and to  $C, D$  identically in Execution 3 and 4 as in Execution 2. In Execution 3, parties in  $B$  only receive messages from  $C$  in the first two rounds, and Byzantine parties in  $C$  pretend to receive messages from a broadcaster whose input is 1. In Execution 4, Byzantine parties in  $B$  pretends only receiving two rounds of messages from  $C$ . Since the first two rounds of messages only depend on the initial state and the message received from the broadcaster in the first round, parties in  $B$  receives the same messages from  $C$ . Therefore,  $A, D$  receive the same messages from  $B$  in both Execution 3 and 4. Similarly,  $A, D$  receive the same messages from  $C$  in both Execution 3 and 4, and thus cannot distinguish these two executions.

**Contradiction.** Since parties in  $B$  commit 0 in Execution 3, parties in  $C$  commit 1 in Execution 4, and parties in  $A, D$  cannot distinguish Execution 3 and 4, either agreement or termination of BRB will be violated. Therefore no such protocol  $\Pi$  exists. ◀

#### 4 Bad-case Latency of Asynchronous Byzantine Reliable Broadcast

In this section, we present 2 impossibility results and 4 asynchronous BRB protocols with tight trade-offs between resilience, good-case latency and bad-case latency.

Recall that the classic Bracha reliable broadcast [5] has optimal resilience of  $n \geq 3f + 1$ , non-optimal good-case latency of 3 rounds and bad-case latency of 4 rounds (1 extra round). The 2-round BRB protocol by Imbs and Raynal [12] has non-optimal resilience of  $n \geq 5f + 1$ , optimal good-case latency of 2 rounds and bad-case latency of 3 rounds (1 extra round). Meanwhile, our 2-round BRB protocol from Section 3 has optimal resilience  $n \geq 4f$ , optimal good-case latency of 2 rounds and bad-case latency of 4 round (2 extra rounds). All protocols above have optimal communication complexity of  $O(n^2)$ , matching the lower bound [8].

On the other hand, we can show that for any  $f > 1$ , no asynchronous BRB protocol can achieve both good-case latency of 2 rounds and bad-case latency of 2 rounds (Theorem 11 in Section 4.1). For the special case of  $f = 1$ , we show it is possible to have a (2, 2)-round BRB (Theorem 12).

Therefore, it is interesting to ask:

*Under what conditions can BRB achieve optimality in all three metrics – optimal resilience of  $n \geq 4f$ , optimal good-case latency of 2 rounds and optimal bad-case latency of 3 rounds (1 extra round)?*

1. **Propose.** The designated broadcaster  $L$  with input  $v$  sends  $\langle \text{propose}, v \rangle$  to all parties.
2. **Ack.** When receiving the first proposal  $\langle \text{propose}, v \rangle$  from the broadcaster, a party sends an **ack** message for  $v$  to all parties in the form of  $\langle \text{ack}, v \rangle$ .
3. **2-round Commit.** When receiving  $\langle \text{ack}, v \rangle$  from  $n - 2$  distinct non-broadcaster parties, a party commits  $v$  and terminates.

■ **Figure 3** (2, 2)-round BRB Protocol under  $n \geq 4f, f = 1$ .

We show it is impossible for the general case of  $f \geq 3$ , by proving that no BRB protocol under  $n \leq 5f - 2, f \geq 3$  can achieve (2, 3)-round (Theorem 13). For  $f \geq 3$ , our BRB (Figure 1) in earlier Section 3.1 has optimal good-case latency of 2 rounds and optimal resilience  $n \geq 4f$ , but with bad-case latency of 4 rounds. On the other hand, we give a (2, 3)-round BRB protocol (Figure 5) with tight resilience  $n \geq 5f - 1$ , improving the  $n \geq 5f + 1$  resilience of Imbs and Raynal [12]. For the special case of  $f = 2$ , we show it is possible to construct a (2, 3)-round BRB (Figure 4) with optimal resilience  $n \geq 4f$ .

#### 4.1 Impossibility of (2, 2)-round BRB

For the general case of  $f \geq 2$ , we show any asynchronous BRB protocol cannot achieve (2, 2)-round. The proof of Theorem 11 is deferred to Appendix A due to space limit.

► **Theorem 11.** *Any asynchronous unauthenticated Byzantine reliable broadcast protocol under  $f \geq 2$  and has a good-case latency of 2 rounds must have a bad-case latency of at least 3 rounds.*

#### 4.2 (2, 2)-round BRB Protocol under $n \geq 4f, f = 1$

For the special case of  $f = 1$ , we can show a simple BRB protocol (Figure 3) that has optimal good-case latency and bad-case latency of 2 rounds, while having optimal resilience  $n \geq 4$ .

► **Theorem 12.** *The protocol in Figure 3 solves Byzantine reliable broadcast under asynchrony with optimal resilience  $n \geq 4, f = 1$ , optimal good-case latency and bad-case latency of 2 rounds.*

**Proof.**

**Validity and Good-case Latency.** If the broadcaster is honest, it sends the same proposal of value  $v$  to all parties, and all  $n - 2 \geq 2$  non-broadcaster honest parties will multicast the **ack** message for  $v$ . Since there is just one Byzantine party, its **ack** is below the  $n - 2$  threshold. Then all honest parties will commit  $v$  after receiving  $n - 2$  **ack** messages at Step 3 and terminate. The good-case latency is 2 rounds, including broadcaster sending the proposal and all parties sending **ack** message.

**Agreement, Termination and Bad-case Latency.** If the broadcaster is honest, by validity all honest parties will commit the same value. If the broadcaster is Byzantine, then all  $n - 1$  non-broadcaster parties are honest. If an honest party commits  $v$  at Step 3, then it receives  $n - 2$  **ack** messages of  $v$  from distinct non-broadcaster parties, and thus all honest parties will also receive these **ack** messages and commit  $v$ . Since all honest parties commit in the same asynchronous round, the bad-case latency is also 2 rounds. ◀

1. **Propose.** The designated broadcaster  $L$  with input  $v$  sends  $\langle \text{propose}, v \rangle$  to all parties.
2. **Ack.** When receiving the first proposal  $\langle \text{propose}, v \rangle$  from the broadcaster, a party sends a **ack** message for  $v$  to all parties in the form of  $\langle \text{ack}, v \rangle$ .
3. **2-round Commit.** When receiving  $\langle \text{ack}, v \rangle$  from  $n - f - 1$  distinct non-broadcaster parties, a party commits  $v$  and terminates.
4. **Vote and Lock.**
  - When receiving  $\langle \text{ack}, v \rangle$  from a non-broadcaster party  $j$ , a party sends  $\langle \text{vote}, j, v \rangle$  to all parties if not yet sent  $\langle \text{vote}, j, v \rangle$ .
  - When receiving  $\langle \text{vote}, j, v \rangle$  from  $n - f - 2$  distinct *non-broadcaster parties other than  $j$* , a party locks on  $v$  for party  $j$ .
5. **3-round Commit.** When locking on the same  $v$  for  $n - 2f$  distinct non-broadcaster parties, a party commits  $v$  and terminates.

■ **Figure 4** (2, 3)-round BRB under  $n \geq 4f, f = 2$ .

### 4.3 Impossibility of (2, 3)-round BRB

► **Theorem 13.** *Any asynchronous unauthenticated Byzantine reliable broadcast protocol under  $n \leq 5f - 2, f \geq 3$  and has a good-case latency of 2 rounds must have a bad-case latency of at least 4 rounds.*

The proof of Theorem 13 is deferred to Appendix B due to the space limit.

### 4.4 (2, 3)-round BRB Protocol under $n \geq 4f, f = 2$

For the special case of  $f = 2$ , we propose a (2, 3)-round BRB protocol (Figure 4) that has optimal resilience  $n \geq 4f$ . The main idea is that all parties send **ack** for broadcaster's proposal, and also send **vote** for other parties' **ack**. When receiving enough **vote** messages of  $v$  for the same party, a party locks on  $v$ . The protocol guarantees that all honest parties lock on the same value for each party when  $f = 2$ . Then, the 3-round commit step let a party commits if the party locks on the same value for a majority of the parties. Since all parties send a **vote** for all other parties, the message and communication complexity are both  $O(n^3)$ .

► **Lemma 14.** *If the broadcaster is Byzantine and an honest party locks on  $v$  for party  $j$ , then all honest parties also lock on  $v$  for party  $j$ .*

**Proof.** Since an honest party locks on  $v$  for party  $j$ , it receives  $n - f - 2$  **vote** messages from non-broadcaster parties other than  $j$ . If  $j$  is honest, then it sends the same **ack** to all parties, and thus all honest parties receive  $n - f - 2$  **vote** for party  $j$  from non-broadcaster honest parties other than  $j$ . If  $j$  is Byzantine, then the parties other than  $j$  and the broadcaster are all honest. Since an honest party receives  $n - f - 2$  **vote** messages from these honest parties, all honest parties will also receive the messages. Therefore, all honest parties also lock on  $v$  for party  $j$ . ◀

► **Theorem 15.** *The protocol in Figure 4 solves Byzantine reliable broadcast under asynchrony with optimal resilience  $n \geq 4f, f = 2$  and optimal good-case latency of 2 rounds, and has bad-case latency of 3 rounds.*

**Proof.**

**Validity and Good-case Latency.** If the broadcaster is honest, it sends the same proposal of value  $v$  to all parties, and all  $n - f - 1$  honest non-broadcaster parties will multicast the **ack** message for  $v$ . Since there are only  $f$  Byzantine party, their **ack** messages is below the  $n - f - 1$  threshold. Then all honest parties will commit  $v$  after receiving  $n - f - 1$  **ack** messages at Step 3 and terminate. The good-case latency is 2 rounds, including broadcaster sending the proposal and all parties sending **ack** message.

**Agreement.** If the broadcaster is honest, by validity all honest parties will commit the same value. Now consider when the broadcaster is Byzantine, and suppose there are  $t > 0$  Byzantine parties there are at most  $t - 1$  Byzantine parties among non-broadcasters.

If any two honest parties commit different values at Step 3, then there must be at least  $n - f - 1 - (t - 1) = n - f - t$  honest parties sending **ack** for each of these different values. It is impossible by quorum intersection since there are only  $n - t$  honest parties.

Suppose any two honest parties commit different values at Step 5. Then, there must exists at least  $2(n - 2f) - (n - 1) \geq 1$  party for which the two committed honest parties lock different values. However, this contradicts Lemma 14, which states honest parties lock on the same value for any party when the broadcaster is Byzantine. Hence, no two honest parties can commit different values at Step 5.

Now we show that if an honest party  $h1$  commits  $v$  at Step 3 and another honest party  $h2$  commits  $v'$  at Step 5, then it must be  $v = v'$ . Suppose  $h1$  commits  $v$  at Step 3, then at least  $n - f - 1 - (f - 1) = n - 2f$  honest non-broadcaster parties send **ack** for  $v$ . All honest parties will lock on  $v$  for these  $n - 2f$  non-broadcaster parties, which is a majority of the  $n - 1$  non-broadcaster parties. Therefore any honest party that commits  $v'$  at Step 5 must have  $v' = v$ .

**Termination and Bad-case Latency.** If the broadcaster is honest, by validity all honest parties will commit the same value. If the broadcaster is Byzantine, once an honest party commits  $v$  at Step 3, there are  $n - 2f$  non-broadcaster honest parties that send **ack** for  $v$ , and all honest parties will eventually lock on  $v$  for these parties after receiving the **vote** messages. Therefore all honest parties will commit  $v$  at Step 5 after 1 extra round. ◀

## 4.5 (2, 3)-round BRB under $n \geq 5f - 1$

In this section, we improve the resilience of 2-round BRB protocol in the previous work [12] from  $5f + 1$  to  $5f - 1$ , while keeping the bad-case latency 3 rounds. The protocol is presented in Figure 5, and the main difference compared to Imbs and Raynal [12] is that in Step 2, parties send **ack** for  $v$  if receiving  $n - 2f$  **ack** from *non-broadcaster parties*, instead of from any parties as in [12]. The intuition is that when the broadcaster is Byzantine, the above set of non-broadcaster parties only contains  $f - 1$  Byzantine parties, and thus we can reduce the total number of parties but still ensure quorum intersection.

► **Theorem 16.** *The protocol in Figure 5 solves Byzantine reliable broadcast under asynchrony with resilience  $n \geq 5f - 1$  and optimal good-case latency of 2 rounds, and has bad-case latency of 3 rounds.*

1. **Propose.** The designated broadcaster  $L$  with input  $v$  sends  $\langle \text{propose}, v \rangle$  to all parties.
2. **Ack.**
  - When receiving the first proposal  $\langle \text{propose}, v \rangle$  from the broadcaster, a party sends a **ack** message for  $v$  to all parties in the form of  $\langle \text{ack}, v \rangle$ .
  - When receiving  $\langle \text{ack}, v \rangle$  from  $n - 2f$  distinct *non-broadcaster parties*, a party sends  $\langle \text{ack}, v \rangle$  to all parties if not yet sent  $\langle \text{ack}, v \rangle$ .
3. **Commit.** When receiving  $\langle \text{ack}, v \rangle$  from  $n - f - 1$  distinct non-broadcaster parties, a party commits  $v$  and terminates.

■ **Figure 5** (2, 3)-round BRB under  $n \geq 5f - 1$ .

**Proof.**

**Validity and Good-case Latency.** If the broadcaster is honest, it sends the same proposal of value  $v$  to all parties, and all  $n - f - 1$  honest non-broadcaster parties will multicast the **ack** message for  $v$ . Since there are only  $f$  Byzantine party, their **ack** messages is below the  $n - f - 1$  threshold. Then all honest parties will commit  $v$  after receiving  $n - f - 1$  **ack** messages at Step 3 and terminate. The good-case latency is 2 rounds, including broadcaster sending the proposal and all parties sending **ack** message.

**Agreement.** If the broadcaster is honest, by validity all honest parties will commit the same value. If the broadcaster is Byzantine, and suppose there are  $t > 0$  Byzantine parties, then there are  $t - 1$  Byzantine parties among all non-broadcaster parties. Suppose that two honest parties commit different values  $v \neq v'$ , then by Step 3 there are at least  $n - f - 1 - (t - 1) = n - f - t$  honest parties  $A$  that send **ack** for  $v$  and at least  $n - f - 1 - (t - 1) = n - f - t$  honest parties  $B$  that send **ack** for  $v'$ . Since there are  $n - t$  honest parties in total,  $|A \cap B| \geq 2(n - f - t) - (n - t) = n - 2f - t \geq 3f - t - 1 > 0$ , there must exist some honest party that sends **ack** due to the second condition of Step 2. If the above only happens to  $v$ , then there are at least  $n - 2f - (t - 1) = n - 2f - t + 1$  honest parties that send **ack** for  $v$  due to receiving the **propose** from the broadcaster. This contradicts the fact that at least  $n - f - t$  honest parties send **ack** for  $v'$  due to receiving **propose**, since  $(n - 2f - t + 1) + (n - f - t) > n - t$ . If the above happens to both  $v, v'$ , then there are at least  $n - 2f - (t - 1) = n - 2f - t + 1$  honest parties that send **ack** for  $v$  (and for  $v'$ , respectively) due to receiving the **propose** from the broadcaster. This is also impossible since  $2(n - 2f - t + 1) \geq n + f - 2t + 1 > n - t$ . Therefore, all honest parties commit the same value.

**Termination and Bad-case Latency.** If the broadcaster is honest, by validity all honest parties will commit the same value. If the broadcaster is Byzantine, once an honest party commits  $v$ , there are  $n - 2f$  non-broadcaster honest parties that send **ack** for  $v$ . Therefore all honest parties will send **ack** for  $v$  and hence commit  $v$  after 1 extra round. ◀

## 5 Extension to Unauthenticated Byzantine Broadcast under Synchrony

In this section, we extend the previous results to show the good-case latency results for unauthenticated Byzantine broadcast under synchrony. It is well-known that unauthenticated Byzantine broadcast or Byzantine reliable broadcast is solvable if and only if  $n \geq 3f + 1$ .

## 5:14 Good-Case and Bad-Case Latency of Unauthenticated Byzantine Broadcast

We adopt the synchrony model assumptions from [4], including distinguishing the latency bounds  $\delta$  and  $\Delta$ , and the clock assumption, briefly as follows. More details about the model assumptions can be found in [4].

**Network delays.** We separate the *actual bound*  $\delta$ , and the *conservative bound*  $\Delta$  on the network delay:

- For one execution,  $\delta$  is the upper bound for message delays between any pair of honest parties, but the value of  $\delta$  is *unknown* to the protocol designer or any party. Different executions may have different  $\delta$  values.
- For all executions,  $\Delta$  is the upper bound for message delays between any pair of honest parties, and the value of  $\Delta$  is *known* to the protocol designer and all parties.

**Clock synchronization.** Each party is equipped with a local clock that starts counting at the beginning of the protocol execution. We assume the *clock skew* is at most  $\sigma$ , i.e., they start the protocol at most  $\sigma$  apart from each other. We assume parties have *no clock drift* for convenience. There exist clock synchronization protocols [7, 1] that guarantee a bounded clock skew of  $\sigma \leq \delta$ . Since the value of  $\delta$  is unknown to the protocol designer or any party, our protocol will use  $\Delta$  as the parameter for clock skew in the protocol. Note that the actual clock skew is still  $\sigma \leq \delta$ , guaranteed by the clock synchronization protocols [7, 1]. In addition, due to clock skew, the BA primitive used in our BB protocol (Figure 6) needs to tolerate up to  $\sigma$  clock skew. For instance, any synchronous lock-step BA can do so by using a clock synchronization algorithm [7, 1] to ensure at most  $\Delta$  clock skew, and setting each round duration to be  $2\Delta$  to enforce the abstraction of lock-step rounds.

► **Theorem 17.** *Any unauthenticated Byzantine reliable broadcast protocol under  $3f + 1 \leq n \leq 4f - 1$  must have a good-case latency of at least  $3\delta$  under synchrony.*

The proof of Theorem 17 is analogous to that of Theorem 10, and is omitted here for brevity. Next, we show a synchronous BB protocol in Figure 6 that has good-case latency of  $2\delta$  under  $n \geq 4f$ .

**Protocol description.** The protocol is presented in Figure 6, and is inspired by our (2, 4)-round asynchronous BRB protocol (Figure 1) from Section 3.1. The main idea is to add a Byzantine agreement at the end of the protocol to ensure termination, since BRB does not require termination when the broadcaster is Byzantine. The input of the BA is called `lock`, which is set to be some default value  $\perp$  initially, and will be set when commit in Step 3 or receiving enough `vote` in Step 4. One guarantee implied by the (2, 4)-round BRB protocol is that, when any honest party commit  $v$  in Step 3, all honest parties will lock on  $v$ , and therefore the BA will only output  $v$ .

► **Theorem 18.** *The protocol in Figure 6 solves Byzantine broadcast under synchrony with optimal resilience  $n \geq 4f$  and optimal good-case latency of  $2\delta$ .*

**Proof.**

**Validity and Good-case Latency.** If the broadcaster is honest, it proposes the same value  $v$  to all parties, and all honest parties will send `ack` for  $v$ . Then at Step 3, all honest parties receive  $n - f - 1$  `ack` messages of  $v$  after  $2\delta$  time (which is before local time  $2\Delta + \sigma$ ), and commits  $v$ .

Initially, every party  $i$  starts the protocol at most  $\delta$  time apart with a local clock and sets  $\text{lock} = \perp$ ,  $\sigma = \Delta$ .

1. **Propose.** The designated broadcaster  $L$  with input  $v$  sends  $\langle \text{propose}, v \rangle$  to all parties.
2. **Ack.** When receiving the first proposal  $\langle \text{propose}, v \rangle$  from the broadcaster, a party sends an **ack** message for  $v$  to all parties in the form of  $\langle \text{ack}, v \rangle$ .
3. **Commit.** When receiving  $\langle \text{ack}, v \rangle$  from  $n - f - 1$  distinct non-broadcaster parties at time  $t$ , a party sets  $\text{lock} = v$ . If  $t \leq 2\Delta + \sigma$ , the party commits  $v$ .
4. **Vote.**
  - When receiving  $\langle \text{ack}, v \rangle$  from  $n - 2f$  distinct non-broadcaster parties, a party sends a **vote** message for  $v$  to all parties in the form of  $\langle \text{vote}, v \rangle$  if not yet sent **vote** for any value.
  - When receiving  $\langle \text{vote}, v \rangle$  from  $n - f - 1$  distinct non-broadcaster parties, a party sets  $\text{lock} = v$ .
5. **Byzantine agreement.** At local time  $3\Delta + 2\sigma$ , a party invokes an instance of Byzantine agreement with  $\text{lock}$  as the input. If not committed, the party commits on the output of the Byzantine agreement. Terminate.

■ **Figure 6**  $2\delta$  unauthenticated BB protocol under  $n \geq 4f$ .

**Agreement.** If all honest parties commit at Step 5, all honest parties commit on the same value due to the agreement property of the BA. Otherwise, there must be some honest party that commits at Step 3. First, no two honest parties can commit different values at Step 3 due to quorum intersection. Now suppose any honest party  $h$  that commits  $v$  at Step 3. If the broadcaster is honest, by validity, all honest parties commits  $v$ . If the broadcaster is Byzantine, then there are  $f - 1$  Byzantine parties among non-broadcasters. Since  $h$  receives  $n - f - 1$  **ack** messages from non-broadcasters, at least  $n - f - 1 - (f - 1) = n - 2f$  of them are from honest parties. Then, all honest parties receive these  $n - 2f$  **ack** messages and set  $\text{lock} = v$  at their local time  $\leq (2\Delta + \sigma) + \Delta + \sigma = 3\Delta + 2\sigma$ , before invoking the Byzantine agreement primitive at Step 5, since the clock skew is  $\sigma$  and message delay is bounded by  $\Delta$ . Also by quorum intersection, there cannot be  $n - 2f$  **ack** messages for  $v' \neq v$ , since the set of  $(n - 2f) - (f - 1) = n - 3f + 1$  honest parties who voted for  $v'$  and the set of  $n - 2f$  honest parties who voted for  $v$  intersect at  $\geq (n - 3f + 1) + (n - 2f) - (n - f) \geq 1$  honest parties. Therefore, at Step 5, all honest parties have the same input  $\text{lock} = v$  to the BA. Then by the validity condition of the BA primitive, the output of the agreement is also  $v$ . Any honest party that does not commit at Step 3 will commit  $v$  at Step 5.

**Termination.** According to the protocol, honest parties terminate at Step 5, and they commit a value before termination. ◀

## 6 Related Work

Byzantine fault-tolerant broadcast, first proposed by Lamport et al. [13], have received a significant amount of attention for several decades. Under synchrony, the deterministic Dolev-Strong protocol [10] solves Byzantine broadcast in worst-case  $f + 1$  rounds, matching a lower bound [11]. Under asynchrony, Byzantine broadcast is unsolvable even with a single

failure. Byzantine reliable broadcast relaxes the termination property of Byzantine broadcast, and the classic Byzantine reliable broadcast by Bracha [5] has a good-case latency of 3 rounds and bad-case latency of 4 rounds with optimal resilience  $n \geq 3f + 1$ . Later works improve the good-case latency of reliable broadcast to 2 rounds by trading off resilience [12] or using authentication (signatures) [4]. A recent line of work studies the good-case latency of authenticated BFT protocols, including [2, 3, 4].

## 7 Conclusion

In this paper, we investigate the good-case latency of unauthenticated Byzantine fault-tolerant broadcast, which is time for all honest parties to commit given that the broadcaster is honest. We show the tight results are 2 rounds under  $n \geq 4f$  and 3 rounds under  $3f + 1 \leq n \leq 4f - 1$  for asynchronous Byzantine reliable broadcast, which can be extended for synchronous Byzantine broadcast as well. In addition, we also study the bad-case latency for asynchronous BRB which measures how fast can all honest parties commit when the broadcaster is dishonest and some honest party commits. We show 2 impossibility results and 4 matching asynchronous BRB protocols, including (2, 4)-BRB under  $n \geq 4f$ , F2-BRB of (2, 3)-round under  $n \geq 4f, f = 2$ , F1-BRB of (2, 2)-round under  $n \geq 4f, f = 1$ , and (2, 3)-BRB under  $n \geq 5f - 1$ .

---

## References

- 1 Ittai Abraham, Srinivas Devadas, Danny Dolev, Kartik Nayak, and Ling Ren. Synchronous byzantine agreement with expected  $O(1)$  rounds, expected  $O(n^2)$  communication, and optimal resilience. In *International Conference on Financial Cryptography and Data Security (FC)*, pages 320–334. Springer, 2019.
- 2 Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Maofan Yin. Sync hotstuff: Simple and practical synchronous state machine replication. *IEEE Symposium on Security and Privacy (SP)*, 2020.
- 3 Ittai Abraham, Kartik Nayak, Ling Ren, and Zhuolun Xiang. Brief announcement: Byzantine agreement, broadcast and state machine replication with optimal good-case latency. In *34th International Symposium on Distributed Computing (DISC)*, 2020.
- 4 Ittai Abraham, Kartik Nayak, Ling Ren, and Zhuolun Xiang. Good-case latency of byzantine broadcast: A complete categorization. In *Proceedings of the third annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 331–341, 2021.
- 5 Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.
- 6 Ran Canetti and Tal Rabin. Fast asynchronous byzantine agreement with optimal resilience. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing (STOC)*, pages 42–51, 1993.
- 7 Danny Dolev, Joseph Y Halpern, Barbara Simons, and Ray Strong. Dynamic fault-tolerant clock synchronization. *Journal of the ACM (JACM)*, 42(1):143–185, 1995.
- 8 Danny Dolev and Rüdiger Reischuk. Bounds on information exchange for byzantine agreement. *Journal of the ACM (JACM)*, 32(1):191–204, 1985.
- 9 Danny Dolev, Ruediger Reischuk, and H Raymond Strong. Early stopping in byzantine agreement. *Journal of the ACM (JACM)*, 37(4):720–741, 1990.
- 10 Danny Dolev and H. Raymond Strong. Authenticated algorithms for byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983.
- 11 Michael J Fischer and Nancy A Lynch. A lower bound for the time to assure interactive consistency. *Information Processing Letters*, 14(4):183–186, 1982.



- 12 Damien Imbs and Michel Raynal. Trading off  $t$ -resilience for efficiency in asynchronous byzantine reliable broadcast. *Parallel Processing Letters*, 26(04):1650017, 2016.
- 13 Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

## A Proof of Theorem 11

**Proof.** Suppose on the contrary there exists an asynchronous BRB protocol  $\Pi$  that tolerates  $f = 2$  and has  $(2, 2)$ -round. We assume  $n \geq 4f = 8$ , otherwise no protocol can solve BRB with good-case latency of 2 rounds by Theorem 10. Denote the broadcaster as party 0 always, and remaining parties as party  $1, \dots, n - 1$ . We construct the following executions.

- Execution 1. The broadcaster is honest, and has input 0. Party  $n - 1$  is Byzantine and remain silent. Then all honest parties commit 0 in 2 rounds by assumption.
- Execution 2. The broadcaster is Byzantine, and behaves honestly to parties  $1, \dots, n - 3$  with input 0, and remains silent to other parties. Party  $n - 2$  is Byzantine, and behaves identically to party  $n - 3$  as in Execution 1, but remains silent to rest of the parties. Any messages from party  $n - 1$  are delayed and not delivered in 2 rounds. It is easy to see that party  $n - 3$  cannot distinguish Execution 2 and 1 in 2 rounds, therefore it commits 0 in 2 rounds in Execution 2 as well. By assumption, parties  $1, \dots, n - 4$  also commit 0 in 2 rounds in Execution 2.
- Execution  $x$  for  $x = 3, \dots, n - 3$ . The broadcaster is Byzantine, and behaves honestly to parties  $1, \dots, n - x - 1$  with input 0, and remains silent to other parties. Party  $n - x$  is Byzantine, and behaves identically to party  $n - x - 1$  as in Execution  $x - 1$ , but remains silent to rest of the parties. Any messages from party  $n - x + 1$  are delayed and not delivered in 2 rounds. It is easy to see that party  $n - x - 1$  cannot distinguish Execution  $x$  and  $x - 1$  in 2 rounds, therefore it commits 0 in 2 rounds in Execution  $x$  as well. By assumption, parties  $1, \dots, n - x - 2$  also commit 0 in 2 rounds in Execution 2.
- Execution  $n - 2$ . The broadcaster is Byzantine, and behaves honestly to party 1 with input 0, and remains silent to other parties. Party 2 is Byzantine, and behaves identically to party 1 as in Execution  $n - 3$ , but remains silent to rest of the parties. Any messages from party 3 are delayed and not delivered in 2 rounds. It is easy to see that party 1 cannot distinguish Execution  $n - 2$  and  $n - 3$  in 2 rounds, therefore it commits 0 in 2 rounds in Execution  $n - 2$  as well.

Similarly, we can construct  $n - 2$  symmetric executions, where the broadcaster has input 1, and in the last execution the broadcaster only behaves honestly to party  $n - 1$  with input 1, and party  $n - 1$  commits 1 in 2 rounds.

**Contradiction.** Now we consider another execution, where the broadcaster is Byzantine, it behaves to party 1 honestly with input 0, and to party  $n - 1$  honestly with input 1, and remain silent to other parties. Party 2 is Byzantine, it behaves to party 1 identically as in Execution  $n - 3$ , and to party  $n - 1$  identically as the party  $n - 2$  to party  $n - 1$  in the last execution of the constructed symmetric executions (due to symmetric of the non-broadcaster parties, the index does not matter). Any messages between parties  $1, n - 1$  are delayed and not delivered in 2 rounds. Then, party 1 commits 0 in 2 rounds while party  $n - 1$  commit 1 in 2 rounds, breaking agreement of the BRB. Therefore, such protocol  $\Pi$  does not exist. ◀

**B Proof of Theorem 13**

**Proof.** Suppose on the contrary that there exists an asynchronous BRB protocol  $\Pi$  under  $n = 5f - 2$ ,  $f \geq 3$  that has  $(2, 3)$ -round. Denote the broadcaster as party 0 always, denote 2 non-broadcaster parties as  $p, q$ , and divide the remaining  $5f - 5$  parties into 5 groups  $G_1, \dots, G_5$  each of size  $f - 1$  (recall  $f - 1 \geq 2$ ). Denote  $G_L = \{p\} \cup G_1 \cup G_2$  and  $G_R = G_4 \cup G_5 \cup \{q\}$ . We use  $S[i]$  to denote the  $i$ -th party in set  $S$ , where  $S$  can be any set defined above (such as  $G_j$  for  $j = 1, \dots, 5$  and  $G_L, G_R$ ). We construct the following executions. In all constructed executions, all messages are delivered by the recipient after  $\Delta$  time by default, and we will explicitly specify the messages that are delayed by the adversary due to asynchrony.

- $E_1^0$ . The broadcaster is honest and has input 0. Parties in  $G_5 \cup \{q\}$  are Byzantine, and they behave honestly except that they pretend to receive from a broadcaster whose input is 1. Since the broadcaster is honest, by validity and good-case latency, all honest parties commit 0 after receiving two rounds of messages.
- $E_1^1$ . This execution is a symmetric case of  $E_1^0$ . The broadcaster is honest and has input 1. Parties in  $G_1 \cup \{p\}$  are Byzantine, and they behave honestly except that they pretend to receive from a broadcaster whose input is 0. Since the broadcaster is honest, by validity and good-case latency, all honest parties commit 1 after receiving two rounds of messages.
- $E_2^0$ . The broadcaster is Byzantine, it behaves to  $G_L \cup G_3$  identically as in  $E_1^0$ , and to  $G_5 \cup \{q\}$  identically as in  $E_1^1$ . Parties in  $G_4$  are Byzantine, they behave to the party  $G_3[f - 1]$  honestly (recall that  $f - 1 \geq 2$  so  $G_3[f - 1] \neq G_3[1]$ ) but pretending to receive from the broadcaster in  $E_1^0$ , and to other parties honestly but pretending to receive from the broadcaster in  $E_1^1$ .

▷ **Claim.** The honest party  $G_3[f - 1]$  cannot distinguish  $E_2^0$  and  $E_1^0$  in 2 rounds, and thus will commit 0 in round 2. Then, by assumption, all honest parties also commit 0 in round 3 in  $E_2^0$ . The broadcaster behaves to  $G_3[f - 1]$  identically in both executions. The messages sent to  $G_3[f - 1]$  in the first round by any non-broadcaster party are identical in  $E_2^0$  and  $E_1^0$ , since the first round message only depends on the initial state and all Byzantine parties behave honestly in the first round. For the second round, since in  $E_1^0$  the Byzantine parties in  $G_5 \cup \{q\}$  pretend to receive from a broadcaster with input 1, they send the same round-2 messages as in  $E_2^0$ . For the Byzantine parties in  $G_4$ , they behave identically to  $G_3[f - 1]$  by construction. All honest parties in  $G_L$  also behave identically to  $G_3[f - 1]$  in round 2 since they receive the same round-1 messages. Therefore party  $G_3[f - 1]$  cannot distinguish  $E_2^0$  and  $E_1^0$  in 2 rounds, and thus will commit 0 in round 2.

- $E_3^0$ . The broadcaster is Byzantine, it behaves to  $G_L$  identically as in  $E_1^0$ , and to  $G_R$  identically as in  $E_1^1$ . Parties in  $G_3$  are Byzantine, they behave to other parties identically as in  $E_2^0$ .

▷ **Claim.** The honest parties in  $G_L \cup G_R$  cannot distinguish  $E_3^0$  and  $E_2^0$  in 3 rounds, and thus will commit 0 in round 3. For the round-1 message, honest parties receive the same messages in both executions since Byzantine parties including the broadcaster send the same messages. For the round-2 message, the Byzantine parties of  $G_4$  in  $E_2^0$  behave to  $G_L \cup G_R$  as if they receive from a broadcaster with input 1, which would be identically to  $E_3^0$ . The Byzantine parties of  $G_3$  in  $E_3^0$  behave to other parties identically as in  $E_2^0$  by construction. Similarly,  $G_1 \cup G_2$  receive the same round-3 messages in both executions, and thus cannot distinguish  $E_3^0$  and  $E_2^0$  in 3 rounds, and will commit 0 in round 3 in  $E_3^0$  as well.

- $E_{2j+2}^0$  for  $j = 1, 2, \dots, |G_R| = 2f - 1$ . The broadcaster is Byzantine, it behaves to  $G_L \cup \{G_3[1]\}$  identically as in  $E_1^0$ , and to  $G_R$  identically as in  $E_1^1$ . Parties in  $G_3 \setminus \{G_3[1]\}$  are Byzantine (recall that  $|G_3| = f - 1 \geq 2$ ), and they behave to all honest parties identically as in  $E_{2j+1}^0$ . Party  $G_R[j]$  is Byzantine, and it behaves to all honest parties except  $p$  identically as in  $E_{2j+1}^0$ , and to party  $p$  honestly except that it pretends receiving no message from  $G_3$  sent after round 1. Any round-2 or round-3 message from  $G_3[1]$  to parties in  $G_R[i], i = 1, \dots, j - 1$  are delayed and received only after round 3.
- $E_{2j+3}^0$  for  $j = 1, 2, \dots, |G_R| = 2f - 1$ . The broadcaster is Byzantine, it behaves to  $G_L$  identically as in  $E_1^0$ , and to  $G_R$  identically as in  $E_1^1$ . Parties in  $G_3$  are Byzantine, they behave to other parties identically as in  $E_{2j+2}^0$ , but they send no message to  $G_R[j]$  after round 1.

▷ Claim. Any honest party in  $G_L \setminus \{p\}$  cannot distinguish  $E_{2j+2}^0$  and  $E_{2j+1}^0$  in 3 rounds, and it will commit 0 in round 3 in  $E_{2j+2}^0$ . Then, by assumption, party  $p$  will also commit 0 in round 3 in  $E_{2j+2}^0$ . Similar to the previous claim, honest parties receive the same round-1 messages. For round 2, the Byzantine parties in  $E_{2j+2}^0$  behave identically to all honest parties, including party  $p$  since the difference from  $G_R[j]$  to  $p$  is reflected only after round 2. Hence, honest parties in  $G_L \setminus \{p\}$  will also receive the same messages in round 3, thus cannot distinguish  $E_{2j+2}^0$  and  $E_{2j+1}^0$  in 3 rounds.

▷ Claim. Party  $p$  cannot distinguish  $E_{2j+3}^0$  and  $E_{2j+2}^0$  in 3 rounds, and thus will commit 0 in round 3 in  $E_{2j+3}^0$ . Then, by assumption, all honest parties in  $G_L \cup G_R$  also commit 0 in round 3. Similar to previous claim, honest parties receive the same round-1 and round-2 messages. For round 3, since Byzantine parties in  $G_3$  send no message to  $G_R[j]$  after round 1 in  $E_{2j+3}^0$ , the honest party  $G_R[j]$  in  $E_{2j+3}^0$  will behave the same to  $p$  as the Byzantine party  $G_R[j]$  which pretends to  $p$  that it receives no message from  $G_3$  in  $E_{2j+2}^0$ . Hence,  $p$  cannot distinguish  $E_{2j+3}^0$  and  $E_{2j+2}^0$  in 3 rounds.

By the above constructions, we finally have an execution  $E_{2j+3, j=2f-1}^0 = E_{4f+1}^0$  where the Byzantine broadcaster behaves to  $G_L$  with input 0, and to  $G_R$  with input 1, and the Byzantine parties in  $G_3$  send no message to  $G_R$ , but party  $p$  has to commit 0 in 3 rounds. Similarly, we can construct a series of symmetric executions of the above executions including  $E_1^1$ , i.e.,  $E_1^1, E_2^1, \dots, E_{4f+1}^1$ , and have the execution  $E_{4f+1}^1$  where the Byzantine broadcaster also behaves to  $G_L$  with input 0, and to  $G_R$  with input 1, and the Byzantine parties in  $G_3$  send no message to  $G_L$ , but party  $q$  has to commit 1 in 3 rounds.

**Contradiction.** Now we construct another middle execution  $E_m$ , where the Byzantine broadcaster behaves to  $G_L$  with input 0, and to  $G_R$  with input 1, and Byzantine parties in  $G_3$  behave to  $G_L$  identically as in  $E_{4f+1}^0$  and to  $G_R$  identically as in  $E_{4f+1}^1$ . It is easy to see that party  $p$  cannot distinguish  $E_m$  and  $E_{4f+1}^0$  in 3 rounds, and thus will commit 0 in round 3, while party  $q$  cannot distinguish  $E_m$  and  $E_{4f+1}^1$  in 3 rounds, and thus will commit 1 in round 3. This violates the agreement property of BRB, and hence such BRB protocol  $\Pi$  does not exist. ◀

## C 3δ Unauthenticated Byzantine Broadcast under Synchrony

For completeness, we show an unauthenticated BB protocol in Figure 7 with good-case latency of  $3\delta$  under synchrony and  $n \geq 3f + 1$ , inspired by Bracha's reliable broadcast [5].

► **Theorem 19.** *The protocol in Figure 7 solves Byzantine broadcast under synchrony with resilience  $n \geq 3f + 1$  and good-case latency of  $3\delta$ .*

## 5:20 Good-Case and Bad-Case Latency of Unauthenticated Byzantine Broadcast

The correctness proof is similar to that of Theorem 18, and we omit it here for brevity.

Initially, every party  $i$  starts the protocol at most  $\delta$  time apart with a local clock and sets  $\text{lock} = \perp$ ,  $\sigma = \Delta$ .

1. **Propose.** The designated broadcaster  $L$  with input  $v$  sends  $\langle \text{propose}, v \rangle$  to all parties.
2. **Echo.** When receiving the first proposal  $\langle \text{propose}, v \rangle$  from the broadcaster, a party sends an **echo** message for  $v$  to all parties in the form of  $\langle \text{echo}, v \rangle$ .
3. **Vote.**
  - When receiving  $\langle \text{echo}, v \rangle$  from  $n - f$  distinct parties, a party sends a **vote** message for  $v$  to all parties in the form of  $\langle \text{vote}, v \rangle$  and sets  $\text{lock} = v$  if not yet sent **vote** for any value.
  - When receiving  $\langle \text{vote}, v \rangle$  from  $f + 1$  distinct parties, a party sends a **vote** message for  $v$  to all parties in the form of  $\langle \text{vote}, v \rangle$  and sets  $\text{lock} = v$  if not yet sent **vote** for any value.
4. **Commit.** When receiving  $\langle \text{vote}, v \rangle$  from  $n - f$  distinct parties at time  $t$ , a party sets  $\text{lock} = v$ . If  $t \leq 3\Delta + \sigma$ , the party commits  $v$ .
5. **Byzantine agreement.** At local time  $4\Delta + 2\sigma$ , a party invokes an instance of Byzantine agreement with  $\text{lock}$  as the input. If not committed, the party commits on the output of the Byzantine agreement. Terminate.

■ **Figure 7**  $3\delta$  unauthenticated BB protocol under synchrony and  $n \geq 3f + 1$ .

# On Finality in Blockchains

**Emmanuelle Anceaume** ✉ 

CNRS, Univ Rennes, Inria, IRISA, Rennes, France

**Antonella Del Pozzo** ✉ 

CEA-List, Université Paris-Saclay, Palaiseau, France

**Thibault Rieutord** ✉

CEA-List, Université Paris-Saclay, Palaiseau, France

**Sara Tucci-Piergiovanni** ✉ 

CEA-List, Université Paris-Saclay, Palaiseau, France

---

## Abstract

This paper focuses on blockchain finality, which refers to the time when it becomes impossible to remove a block that has previously been appended to the blockchain. Blockchain finality can be deterministic or probabilistic, immediate or eventual. To favor availability against consistency in the face of partitions, most blockchains only offer probabilistic eventual finality: blocks may be revoked after being appended to the blockchain, yet with decreasing probability as they sink deeper into the chain. Other blockchains favor consistency by leveraging the immediate finality of Consensus – a block appended is never revoked – at the cost of additional synchronization.

The quest for “good” deterministic finality properties for blockchains is still in its infancy, though. Our motivation is to provide a thorough study of several possible deterministic finality properties and explore their solvability. This is achieved by introducing the notion of bounded revocation, which informally says that the number of blocks that can be revoked from the current blockchain is bounded. Based on the requirements we impose on this revocation number, we provide reductions between different forms of eventual finality, Consensus and Eventual Consensus. From these reductions, we show some related impossibility results in presence of Byzantine processes, and provide non-trivial results. In particular, we provide an algorithm that solves a weak form of eventual finality in an asynchronous system in presence of an unbounded number of Byzantine processes. We also provide an algorithm that solves eventual finality with a bounded revocation number in an eventually synchronous environment in presence of less than half of Byzantine processes. The simplicity of the arguments should better guide blockchain designs and link them to clear formal properties of finality.

**2012 ACM Subject Classification** Theory of computation

**Keywords and phrases** Blockchain, consistency properties, Byzantine tolerant implementations

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2021.6

**Funding** This work was partially supported by the French ANR project ByBloS (ANR-20-CE25-0002) devoted to the modular design of building blocks for large-scale fault-tolerant multi-users applications.

## 1 Introduction

This paper focuses on blockchain finality, which refers to the time when it becomes impossible to remove a block previously appended to the blockchain. Blockchain finality can be deterministic or probabilistic, immediate or eventual.

Informally, immediate finality guarantees, as its name suggests, that when a block is appended to a local copy, it is immediately finalized and thus will never be revoked in the future. Designing blockchains with immediate finality favors consistency against availability in presence of transient partitions of the system. It leverages the properties of Consensus (i.e. a decision value is unique and agreed by everyone), at the cost of synchronization constraints.



© Emmanuelle Anceaume, Antonella Del Pozzo, Thibault Rieutord, and Sara Tucci-Piergiovanni; licensed under Creative Commons License CC-BY 4.0

25th International Conference on Principles of Distributed Systems (OPODIS 2021).

Editors: Quentin Bramas, Vincent Gramoli, and Alessia Milani; Article No. 6; pp. 6:1–6:19



Leibniz International Proceedings in Informatics

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

## 6:2 On Finality in Blockchains

Assuming partially synchronous environments, most of the permissioned blockchains satisfy the deterministic form of immediate consistency, as for example Red Belly blockchain [8] and Hyperledger Fabric blockchain [2]. The probabilistic form of immediate finality is typically achieved by permissionless pure proof-of-stake blockchains such as Algorand [7].

Unlike immediate finality, eventual finality only ensures that eventually all local copies of the blockchain share a common increasing prefix, and thus finality of their blocks increases as more blocks are appended to the blockchain. The majority of permissionless cryptoassets blockchains, with Bitcoin [20] and Ethereum [25] as celebrated examples, guarantee eventual finality with some probability: blocks may be revoked after being appended to the blockchain, yet with decreasing probability as they sink deeper into the chain. In an effort to replace the energy-wasting proof-of-work (PoW) method of Bitcoin and Ethereum, recent proof-of-stake blockchains such as e.g. [16, 12, 15] emerged. These blockchains offer as well a form of eventual finality. More broadly, all these permissionless solutions favor availability (or progress) relying on a Nakamoto-style consensus: a broadcast primitive to diffuse blocks and a local reconciliation mechanism to select a unique chain. It is indeed admitted that a blockchain may lose consistency by incurring a fork, which is the presence of multiple chains at different processes. The reconciliation mechanism, available to recover from a fork, consists in a local deterministic rule selecting a chain among the different possible alternatives. In Bitcoin for instance any participant reconciles the state following the “longest” chain rule (the term “longest” chain rule is commonly employed, but this is actually the one that required the most work to be built). Once a winner chain is chosen, the other alternatives are revoked, as such all the blocks belonging to them. In designs using Nakamoto-style consensus, however, network effects make the moment at which all honest processes observe the same set of candidate chains unknown. Reconciliation and finalisation guarantees are then uncertain, or simply extremely inefficient, for example by considering a block as finalised after one or more days. To solve this problem a number of projects are investigating how to add “finality gadgets” (e.g., [5, 24]) to Nakamoto-style blockchains, which means seeking additional mechanisms or protocols to reach “better” finality properties in network adversarial settings. The hope is to find ways to get deterministic finality by periodically running finality gadgets on top of Nakamoto-style consensus. For the time being, the only way that has been concretely pursued is to resort to Byzantine Consensus – e.g. Tenderbake [4] adds Byzantine Consensus to the existing proof-of-stake method assuring deterministic finality to each block followed by other two blocks. How to add mechanisms that do not resort to Consensus, however, is an intriguing and open question, related to the finality properties one would like to guarantee.

The quest for “good” deterministic finality properties for blockchains is still in its infancy, though. Our motivation is to provide a protocol-independent abstraction of several possible finality properties to study their solvability. To this aim we formalise, for the first time, the notion of finality in a protocol-agnostic way. At the heart of the proposed formalisation lies the notion of *revocation number*. Informally, given a system run and a blockchain  $bc$  read by a user at some time  $t$ , we call the revocation number the natural number  $n$  such that by pruning the last  $n$  blocks from  $bc$ , we obtain a prefix of any blockchain  $bc'$  read after  $t$ .

By leaving the revocation number unbounded in all the runs of the system, we formalise our weakest form of finality, the eventual finality consistency criterion  $\mathcal{F}$ : In each run, the revocation number can be infinite when the run goes to infinity, still each block will be eventually finalised.

By introducing restrictions on the revocation number, we then introduce stronger criteria. The strongest criterion, called  $\mathcal{F}^c$ , is obtained by restricting the revocation number to be a constant  $c$  in all the runs of the system. Informally,  $\mathcal{F}^c$  guarantees that finality of each block is deferred by at most  $c$  blocks in all system runs, i.e., any block followed by at least  $c$  blocks in the blockchain cannot be revoked.

Between  $\mathcal{F}$  and  $\mathcal{F}^c$  we then define three other forms of deferred finality:  $\mathcal{F}^n$ , where the revocation number is bounded but not known,  $\mathcal{F}^{\diamond,c}$ , where the revocation number is constant but holds only eventually, and finally  $\mathcal{F}^{\diamond,n}$ , where the bound on the revocation number is not known and holds only eventually.  $\mathcal{F}^n$  guarantees that finality of each block is deferred by a constant  $c$  in each system run, but this constant can vary from one run to another. For  $\mathcal{F}^{\diamond,c}$  and  $\mathcal{F}^{\diamond,n}$  we have that  $\mathcal{F}^{\diamond,c}$  guarantees that eventually finality of each block is deferred by  $c$  in all system runs, while for  $\mathcal{F}^{\diamond,n}$ , eventually finality of each block is deferred by  $c$  in each system run with  $c$  varying from one run to another. Nicely, we obtain each consistency criterion by adding a proper bounded revocation property to  $\mathcal{F}$  and we prove that  $\mathcal{F}^n$ ,  $\mathcal{F}^{\diamond,c}$ ,  $\mathcal{F}^{\diamond,n}$  are all equivalent.

The rigorous formalisation of these consistency criteria enables us to easily show that solutions that guarantee  $\mathcal{F}^c$  are equivalent to Consensus, while solutions that guarantee  $\mathcal{F}^n$  (or equivalently  $\mathcal{F}^{\diamond,n}$  and  $\mathcal{F}^{\diamond,c}$ ) are not weaker than Eventual Consensus, an abstraction that captures eventual agreement among all participants. From these reductions, we show some related impossibility results in presence of Byzantine processes. Beside reductions and related impossibilities, we propose the following non-trivial results:

- $\mathcal{F}$  cannot be achieved in an asynchronous system if the reconciliation rule follows the “longest” chain rule (Theorem 22). This implies that the reconciliation rule, used in current blockchains to provide probabilistic finality in synchronous settings, cannot guarantee that participants will eventually converge to a stable prefix of the chain in asynchronous settings.
- A solution that guarantees  $\mathcal{F}$  in an asynchronous system with a possibly infinite set of processes which can append infinitely many blocks. This novel solution is simple and tolerant to an unbounded number of Byzantine processes (Theorem 23).
- A solution that solves  $\mathcal{F}^n$  in an eventually synchronous environment in presence of less than half of Byzantine processes (Theorem 24). The central point of our solution is to let correct processes blame each fork on a particular Byzantine process, which can then be excluded from the computation. Weakening the classic requirement of  $< 1/3$  to  $< 1/2$  Byzantine processes makes such a solution well adapted to large scale adversarial systems. As for the previous one, we are not aware of any such solution in the literature.

We hope that these results will better guide blockchain designs and link them to clear formal properties of finality. Hence, in the remainder of this article, Section 2 situates our work with respect to similar ones. Section 3 formally presents the sequential specification of a blockchain and the formalisation of the different finality properties we may expect from a blockchain when concurrently accessed. Section 4 presents reductions between different forms of finality, Consensus and Eventual Consensus. Section 5 first shows why  $\mathcal{F}$  is not solvable in an asynchronous environment when the “longest” chain rule is used, and then presents two original and simple algorithms that respectively solve  $\mathcal{F}$  and  $\mathcal{F}^n$ . Finally, Section 6 concludes the paper.

## 2 Related Work

Formalization of blockchains in the lens of distributed computing has been recognized as an extremely important topic [14]. Garay et al. [10] have been the first to analyze the Bitcoin backbone protocol and to define invariants this protocol has to satisfy to verify with high probability an eventual consistent prefix. The authors have analyzed the protocol in a synchronous system, while others, as for example Pass et al. [21], have extended this line of work considering a more adversarial network. In those works the specification of the consistency properties are protocol dependent and thus provide an abstraction level that does not allow us to model the blockchain as a shared object being agnostic of the way it is implemented. The objective we pursue throughout this work is to formalize the semantic of the interface between the blockchain and the users. To do so we consider the blockchain as a shared object, and thus the consistency properties are specified independently of the synchrony assumptions of underlying distributed system and the type of failures that may occur. By doing this, we offer a higher level of abstraction than well-known properties do.

This approach has been recently followed in particular by Anta et al. [3], Anceaume et al. [1] and Guerraoui et al. [13]<sup>1</sup>. In Anta et al. [3], the authors propose a formalization of distributed ledgers, modeled as an ordered list of records along with implementations for sequential consistency and linearizability using a total order broadcast abstraction. Anceaume et al. [1] have captured the convergence process of two distinct classes of blockchain systems: the class providing strong prefix as [3] (for each pair of chains returned at two different processes, one is the prefix of the other) and the class providing eventual prefix, in which multiple chains can co-exist but the common prefix eventually converges. The authors of [1] show that to solve strong prefix the Consensus abstraction is needed, however they do not address solvability of eventual prefix and do not formalise finality. Interestingly, our notion of finality and bounded revocation is able to encompass the strong and the eventual prefix consistency properties of [1].

## 3 Definitions

### 3.1 Preliminary Definitions

We describe a blockchain object as an abstract data type which allows us to completely characterize a blockchain by the operations it exports [18]. The basic idea underlying the use of abstract data types is to specify shared objects using two complementary facets: a sequential specification that describes the semantics of the object, and a consistency criterion over concurrent histories, i.e. the set of admissible executions in a concurrent environment [22]. Prior to presenting the blockchain abstract data type we first recall the formalization used to describe an abstract data type (ADT).

#### 3.1.1 Abstract data types

An abstract data type (ADT) is a tuple of the form  $T = (A, B, Z, z_0, \tau, \delta)$ . Here  $A$  and  $B$  are countable sets respectively called *input alphabet* and *output alphabet*.  $Z$  is a countable set of abstract object *states* and  $z_0 \in Z$  is the initial abstract state. The map  $\tau : Z \times A \rightarrow Z$  is the *transition function*, specifying the effect of an input on the object state and the

---

<sup>1</sup> While not related to the blockchain data structure, authors of [13] have formalized the notion of cryptocurrency showing that Consensus is not needed.



map  $\delta : Z \times A \rightarrow B$  is the *output function*, specifying the output returned for a given input and an object local state. An input represents an operation with its parameters, where (i) the operation can have a side-effect that changes the abstract state according to transition function  $\tau$  and (ii) the operation can return values taken in the output  $B$ , which depends on the state in which it is called and the output function  $\delta$ .

### 3.1.2 Concurrent histories of an ADT

Concurrent histories are defined considering asymmetric event structures, i.e., partial order relations among events executed by different processes.

► **Definition 1** (Concurrent history  $H$ ). *The execution of a program that uses an abstract data type  $T = \langle A, B, Z, \xi_0, \tau, \delta \rangle$  defines a concurrent history  $H = \langle \Sigma, E, \Lambda, \mapsto, \prec, \succ \rangle$ , where*

- $\Sigma = A \cup (A \times B)$  is a countable set of operations;
- $E$  is a countable set of events that contains all the ADT operations invocations and all ADT operation response events;
- $\Lambda : E \rightarrow \Sigma$  is a function which associates events to the operations in  $\Sigma$ ;
- $\mapsto$ : is the process order, irreflexive order over the events of  $E$ . Two events  $(e, e') \in E^2$  are ordered by  $\mapsto$  if they are produced by the same process,  $e \neq e'$  and  $e$  happens before  $e'$ , that is denoted as  $e \mapsto e'$ .
- $\prec$ : is the operation order, irreflexive order over the events of  $E$ . For each couple  $(e, e') \in E^2$  if  $e'$  is the invocation of an operation occurred at time  $t'$  and  $e$  is the response of another operation occurred at time  $t$  with  $t < t'$  then  $e \prec e'$ ;
- $\succ$ : is the program order, irreflexive order over  $E$ , for each couple  $(e, e') \in E^2$  with  $e \neq e'$  if  $e \mapsto e'$  or  $e \prec e'$  then  $e \succ e'$ .

## 3.2 The blocktree ADT

We represent a blockchain as a tree of blocks. The same representation has been adopted in [1]. Indeed, while consensus-based blockchains prevent forks or branching in the tree of blocks, blockchain systems based on proof-of-work allow the occurrence of forks to happen hence presenting blocks under a tree structure. The blockchain object is thus defined as a blocktree abstract data type (Blocktree ADT).

### 3.2.1 Sequential Specification of the Blocktree ADT (BT-ADT)

A blocktree data structure is a directed rooted tree  $bt = (V_{bt}, E_{bt})$  where  $V_{bt}$  represents a set of blocks and  $E_{bt}$  a set of edges such that each block has a single path towards the root of the tree  $b_0$  called the genesis block. A branching in the tree is called a *fork*. Let  $\mathcal{BT}$  be the set of blocktrees,  $B$  be the countable and non empty set of uniquely identified blocks and let  $\mathcal{BC}$  be the countable non empty set of blockchains, where a blockchain is a path from a leaf of  $bt$  to  $b_0$ . A blockchain is denoted by  $bc$ . The structure is equipped with two operations `append()` and `read()`. Operation `append(b)` adds block  $b \notin bt$  to  $V_{bt}$  and adds the edge  $(b, b')$  to  $E_{bt}$  where  $b' \in V_{bt}$  is returned by the append selection function  $f_a()$  applied to  $bt$ . Operation `read()` returns the chain  $bc$  selected by the read selection function  $f_r()$  applied to  $bt$  (note that in [1], the `read()` and `append()` operations are defined with a unique selection function). The read selection  $f_r()$  takes as argument the blocktree and returns a chain of blocks, that is a sequence of blocks starting from the genesis block to a leaf block of the blocktree. The chain  $bc$  returned by a `read()` operation  $r$  is called the blockchain, and is denoted by  $r/bc$ . The append selection function  $f_a()$  takes as argument the blocktree and

## 6:6 On Finality in Blockchains

returns a chain of blocks. Function  $last\_block()$  takes as argument a chain of blocks and returns the last appended block of the chain. Only blocks satisfying some validity predicate  $P$  can be appended to the tree. Predicate  $P$  is an application-dependent predicate used to verify the validity of the chain obtained by appending the new block  $b$  to the chain returned by  $f_a()$  (denoted by  $f_a(bt) \frown b$ ). In Bitcoin for instance this predicate embeds the logic to verify that the obtained chain does not contain double spending or overspending transactions. Formally,

► **Definition 2** (Sequential specification of the Blocktree ADT). *The Blocktree Abstract Data Type is the 6-tuple  $BT - ADT = \{A = \{append(b), read()/bc \in \mathcal{BC}\}, B = \mathcal{BC} \cup \{\top, \perp\}, Z = \mathcal{BT}, \xi_0 = b_0, \tau, \delta\}$ , where the transition function  $\tau : Z \times A \rightarrow Z$  is defined by*

$$\begin{aligned} \tau(bt, read()) &= bt \\ \tau(bt, append(b)) &= \begin{cases} (V_{bt} \cup \{b\}, E_{bt} \cup \{b, last\_block(f_a(bt))\}) & \text{if } P(f_a(bt) \frown b) \\ bt & \text{otherwise,} \end{cases} \end{aligned}$$

and where the output function  $\delta : Z \times A \rightarrow B$  is defined by

$$\begin{aligned} \delta(bt, read()) &= f_r(bt) \\ \delta(bt, append(b)) &= \begin{cases} \top & \text{if } P(f_a(bt) \frown b) \\ \perp & \text{otherwise.} \end{cases} \end{aligned}$$

Note that we do not need to add the validity check during the `read` operation in the sequential specification of the Blocktree ADT because in absence of concurrency the validity check during the `append` operation is enough.

### 3.2.2 Concurrent Specification and Consistency Criteria of the BlockTree ADT

The concurrent specification of the blocktree abstract data type is the set of its concurrent histories. A blocktree consistency criterion is a function that returns the set of concurrent histories admissible for the blocktree abstract data type. In this paper, we define different consistency criteria for the blocktree. We first define eventual finality, which is the weakest consistency criterion that we may expect from blockchains, along with the notion of block revocation. We then combine eventual finality with different forms of revocation to provide stronger consistency criteria. The presented family of consistency criteria is a comprehensive characterization of what we may expect from blockchains.

► **Notation 3.**

- $E(a^*, r^*)$  is an infinite set containing an infinite number of `append()` and `read()` invocation and response events;
- $E(a, r^*)$  is an infinite set containing (i) a finite number of `append()` invocation and response events and (ii) an infinite number of `read()` invocation and response events;
- $o_{inv}$  and  $o_{rsp}$  indicate respectively the invocation and response event of an operation  $o$ ; and in particular for the `read()` operation,  $r_{rsp}/bc$  denotes the returned blockchain  $bc$  associated with the response event  $r_{rsp}$  and for the `append()` operation  $a_{inv}(b)$  denotes the invocation of the `append` operation having  $b$  as input parameter;
- $length : \mathcal{BC} \rightarrow \mathbb{N}$  denotes a monotonic increasing deterministic function that takes as input a blockchain  $bc$  and returns a natural number as length of  $bc$ . Increasing monotonicity means that  $length(bc \frown \{b\}) > length(bc)$ ;

- We represent chain  $bc$  as an infinite list  $b_0b^*\perp^+$  of blocks, where the first block  $bc[0] = b_0$ , the genesis block, followed by block values  $b$ , and an infinite number of  $\perp$  values. Notation  $bc[i]$  refers to the  $i$ -th block of blockchain  $bc$ . Note that the special “ $\perp$ ” symbol counts for zero for the length function.

- $bc \sqsubseteq bc'$  if and only if  $bc$  prefixes  $bc'$ . The operator  $\sqsubseteq$  ignores all the records set to  $\perp$ .

► **Definition 4** (BT Eventual Finality Consistency criterion ( $\mathcal{F}$ )). A concurrent history  $H = \langle \Sigma, E, \Lambda, \mapsto, \prec, \nearrow \rangle$  of a system that uses a BT-ADT verifies the BT eventual finality consistency criterion if the following four properties hold:

- **Chain validity:**

$$\forall r_{rsp} \in E, P(r_{rsp}/bc).$$

Each returned chain is valid.

- **Chain integrity:**

$$\forall r_{rsp} \in E, \forall b \in r_{rsp}/bc : b \neq b_0, \exists a_{inv}(b) \in E, a_{inv}(b) \nearrow r_{rsp}.$$

If a block different from the genesis block is returned, then an **append** operation has been invoked with this block as parameter. This property is to avoid the situation in which reads return blocks never appended.

- **Eventual prefix:**

$$\forall E \in E(a, r^*) \cup E(a^*, r^*), \forall r_{rsp}/bc, \forall i \in \mathbb{N} : bc[i] \neq \perp, \exists r'_{rsp} : r_{rsp} \nearrow r'_{rsp}, \forall r''_{rsp} : r'_{rsp} \nearrow r''_{rsp}, (r'_{rsp}/bc')[i] = (r''_{rsp}/bc'')[i] \neq \perp.$$

In all the histories in which the number of read invocations is infinite, then for any read operation such that the returned chain has a block at position  $i$ , there exists a read  $r'/bc'$  from which all the subsequent reads  $r''/bc''$  will return the same block at position  $i$ , i.e.  $bc'[i] = bc''[i] \neq \perp$ .

- **Ever growing tree:**

$$\forall E \in E(a^*, r^*), \forall k \in \mathbb{N}, \exists r \in E : \text{length}(r_{rsp}/bc) > k.$$

In all the histories in which the number of **append** and **read** invocations is infinite, for each length  $k$ , there exists a **read** that returns a chain with length greater than  $k$ . This property avoids the trivial scenario in which the length of the chain remains unchanged despite the occurrence of an infinite number of **append** operations (i.e., tree built as a star with infinite branches of bounded length). Specifically the “Ever growing tree” property imposes that in presence of an infinite number of **read** and **append** operations, for any natural number  $k$ , there will always exist a **read** operation that will return a chain of at least length  $k$ . Note that the well known “Chain Growth Property” [10, 21] states that each (honest) chain grows proportionally with the number of rounds of the protocol, which in contrast to our specification, makes it protocol dependent.

## Bounded revocation

As previously said, the bounded revocation properties are at the heart of our formalisation of blockchain finality. Informally, given a history, we call the revocation number the natural number  $n$  such that for any two reads  $r/bc$  and  $r'/bc'$ , where  $r$  precedes  $r'$ , by pruning the last  $n$  blocks from  $bc$  we obtain a chain that is a prefix of  $bc'$ .

Note that the eventual finality consistency criterion presented so far does not impose any bound on the revocation number, which can be then infinite when the history goes to infinity.

To obtain stronger consistency criteria, we then introduce restrictions to the revocation number. To this aim, we define the *c*-bounded revocation property, which states that the revocation number  $n$  is bounded by a constant  $c$  in all histories. We also define the *bounded revocation property*, which states that the revocation number  $n$  is bounded by a constant

## 6:8 On Finality in Blockchains

$c$  in each history, but may be unbounded when we consider the union of all the histories, i.e., the bound can vary from a history to another. Eventual forms of  $c$ -bounded revocation and bounded revocation state that the revocation number will be equal to a constant  $c$  only eventually. More formally:

► **Definition 5** ( $c$ -Bounded Revocation).  $\exists c \in \mathbb{N}, \forall E, \forall r_{rsp}/bc, r'_{rsp}/bc' \in E : r_{rsp} \nearrow r'_{rsp}, \forall i \in \mathbb{N} : i \leq (\text{length}(bc) - c), bc[i] = bc'[i] \neq \perp$ .

► **Definition 6** (Bounded Revocation).  $\forall E, \exists c \in \mathbb{N}, \forall r_{rsp}/bc, r'_{rsp}/bc' \in E : r_{rsp} \nearrow r'_{rsp}, \forall i \in \mathbb{N} : i \leq (\text{length}(bc) - c), bc[i] = bc'[i] \neq \perp$ .

► **Definition 7** (Eventual  $c$ -Bounded Revocation).  $\exists c \in \mathbb{N}, \forall E, \exists r \in E : \forall r'_{rsp}/bc, r''_{rsp}/bc' \in E : r_{rsp} \nearrow r'_{rsp}, r'_{rsp} \nearrow r''_{rsp}, \forall i \in \mathbb{N} : i \leq (\text{length}(bc') - c), bc'[i] = bc''[i] \neq \perp$

► **Definition 8** (Eventual Bounded Revocation).  $\forall E, \exists c \in \mathbb{N}, \exists r \in E : \forall r'_{rsp}/bc, r''_{rsp}/bc' \in E : r_{rsp} \nearrow r'_{rsp}, r'_{rsp} \nearrow r''_{rsp}, \forall i \in \mathbb{N} : i \leq (\text{length}(bc') - c), bc'[i] = bc''[i] \neq \perp$

Note that Bounded Revocation properties are not protocol dependent in contrast to the well-known “Common-Prefix Property” [10, 21], which states that for any two rounds  $r$  and  $r'$  of the protocol with  $r < r'$ , the (honest) chain read at round  $r$  from which the last  $c$  blocks have been pruned is a prefix of (resp. is equal to with high probability) the one read at round  $r'$ .

Based on these different forms of bounded revocation, we define four criteria stronger than eventual finality. Nicely, we obtain each consistency criterion by adding the proper bounded revocation property to  $\mathcal{F}$ .

By adding  $c$ -bounded revocation to  $\mathcal{F}$ , we obtain the  $c$ -deferred finality form, denoted by  $\mathcal{F}^c$ . Informally,  $\mathcal{F}^c$  guarantees that finality of each block is deferred by at most  $c$  blocks in all histories, i.e., any block followed by at least  $c$  blocks in the blockchain cannot be revoked.

By adding the bounded revocation property to  $\mathcal{F}$ , we obtain the *bounded deferred finality* form, denoted by  $\mathcal{F}^n$ . Informally  $\mathcal{F}^n$  guarantees that finality of each block is deferred by a constant  $c$  in each history, but this constant can vary from history to history. In other words constant  $c$  is unknown.

Finally, by adding respectively, eventual  $c$ -bounded finality and eventual bounded finality to  $\mathcal{F}$ , we obtain other two forms of deferred finality, namely  $\mathcal{F}^{\diamond, c}$   $\mathcal{F}^{\diamond, n}$ , both equivalent to  $\mathcal{F}^n$ . Informally,  $\mathcal{F}^{\diamond, c}$  guarantees that eventually finality of each block is deferred by  $c$  in all histories. For  $\mathcal{F}^{\diamond, n}$ , eventually finality of each block is deferred by  $c$  in each history, with  $c$  varying from history to history.

In the following we formally introduce  $\mathcal{F}^c$ ,  $\mathcal{F}^n$ ,  $\mathcal{F}^{\diamond, c}$ ,  $\mathcal{F}^{\diamond, n}$ , and show equivalences between  $\mathcal{F}^{\diamond, c}$ ,  $\mathcal{F}^{\diamond, n}$  and  $\mathcal{F}^n$ .

► **Definition 9** (BT  $c$ -Deferred Finality Consistency criterion ( $\mathcal{F}^c$ )). *A concurrent history  $H = \langle \Sigma, E, \Lambda, \mapsto, \prec, \nearrow \rangle$  of the system that uses a BT-ADT verifies the BT  $c$ -deferred finality consistency criterion if chain validity, chain integrity, eventual prefix, ever growing tree, and the  $c$ -bounded revocation properties hold.*

► **Definition 10** (BT Bounded Deferred Finality Consistency criterion ( $\mathcal{F}^n$ )). *A concurrent history  $H = \langle \Sigma, E, \Lambda, \mapsto, \prec, \nearrow \rangle$  of the system that uses a BT-ADT verifies the BT bounded deferred finality consistency criterion if chain validity, chain integrity, eventual prefix, ever growing tree, and the bounded revocation properties hold.*

► **Definition 11** (BT Eventual  $c$ -Deferred Finality Consistency criterion ( $\mathcal{F}^{\diamond, c}$ )). *A concurrent history  $H = \langle \Sigma, E, \Lambda, \mapsto, \prec, \nearrow \rangle$  of the system that uses a BT-ADT verifies the BT eventual  $c$ -deferred finality consistency criterion if chain validity, chain integrity, ever growing tree, eventual prefix and the eventual  $c$ -bounded revocation properties hold.*

► **Definition 12** (BT Eventual Bounded Deferred Finality Consistency criterion ( $\mathcal{F}^{\diamond,n}$ )). A concurrent history  $H = \langle \Sigma, E, \Lambda, \mapsto, \prec, \nearrow \rangle$  of the system that uses a BT-ADT verifies the BT eventual bounded deferred finality consistency criterion if chain validity, chain integrity, ever growing tree, eventual prefix and the eventual bounded revocation properties hold.

Note that in the blockchain literature,  $\mathcal{F}^c$ , with  $c = 0$ , is also referred to as *immediate finality*. Immediate finality is equivalent to BT strong consistency defined in [1], which implies that for any two read operations, one of the returned blockchains is the prefix of the other one.

► **Notation 13.** For readability reasons, in the following we will simply say *finality* instead of *finality consistency criterion*.

► **Theorem 14.**  $\mathcal{F}^n$  and  $\mathcal{F}^{\diamond,n}$  are equivalent.

**Proof.** Trivially,  $\mathcal{F}^n$  implies  $\mathcal{F}^{\diamond,n}$ . Let us now consider the other direction. From  $\mathcal{F}^{\diamond,n}$ , we have that given any execution  $E$ , there exists  $c \in \mathbb{N}$  and a read operation  $r$  such that for all reads  $r', r''$  after  $r$ , with  $r' \nearrow r''$  the blockchain returned by  $r'$  pruned of the last  $c$  blocks is a prefix of the blockchain returned by  $r''$ . Let  $c'$  be the maximal length of blockchains returned by read operations occurring before  $r$ , and let  $c'' = \max(c, c')$ . By construction,  $\mathcal{F}^n$  is satisfied for  $E$  with revocation number  $n = c''$ . Hence  $\mathcal{F}^{\diamond,n}$  implies  $\mathcal{F}^n$ . ◀

We now show that  $\mathcal{F}^{\diamond,n}$  and  $\mathcal{F}^{\diamond,c}$  are equivalent. This equivalence is shown by first proving that  $\mathcal{F}^{\diamond,n}$  and  $\mathcal{F}^{\diamond,c=0}$  are equivalent and then that  $\mathcal{F}^{\diamond,c=0}$  and  $\mathcal{F}^{\diamond,c}$  are equivalent.

► **Theorem 15.**  $\mathcal{F}^{\diamond,c=0}$  and  $\mathcal{F}^{\diamond,n}$  are equivalent.

**Proof.** Let  $\mathcal{P}_1$  be a protocol guaranteeing  $\mathcal{F}^{\diamond,n}$ . We build protocol  $\mathcal{P}_2$  as follows: to make an `append()` operation, processes simply use the `append()` operation of  $\mathcal{P}_1$ . For the `read()` operation, processes use the `read()` operation provided by  $\mathcal{P}_1$  to obtain the blockchain and prune the second half of it before returning the first half of the blockchain. Let us show that protocol  $\mathcal{P}_2$  guarantees  $\mathcal{F}^{\diamond,c=0}$ . For this, we need to show that the properties of  $\mathcal{F}^{\diamond,c=0}$  are satisfied:

- **Chain validity:** The chain validity property is still satisfied by pruning half of the chain.
- **Chain integrity:** The chain integrity property is still satisfied by pruning half of the chain.
- **Eventual prefix:** The eventual prefix property is still satisfied by pruning half of the chain.
- **Ever growing tree:** The ever growing tree property is still satisfied by pruning half of the chain.
- **( $c = 0$ )-eventual bounded revocation:** This property follows from the removal of the second half of the chain. Indeed, if we remove the second half of the chain, then eventually for any two `read()` operations, then the first `read()` returns a prefix of the second `read()` operation.

For the other direction, we can build a solution to  $\mathcal{F}^{\diamond,n}$  using a solution to  $\mathcal{F}^{\diamond,c=0}$ . ◀

► **Theorem 16.**  $\mathcal{F}^{\diamond,c=0}$  and  $\mathcal{F}^{\diamond,c}$  are equivalent.

**Proof.** Trivially,  $\mathcal{F}^{\diamond,c=0}$  implies  $\mathcal{F}^{\diamond,c}$ . For the other direction, we apply a construction close to the one used in the proof of Theorem 15. Specifically, given a protocol  $\mathcal{P}_1$  that guarantees  $\mathcal{F}^{\diamond,c}$ , we build a protocol  $\mathcal{P}_2$  by using  $\mathcal{P}_1$  as follows. To make an `append()` operation, processes simply use the `append()` operation of  $\mathcal{P}_1$ . For the `read()` operation, processes use

the `read()` operation provided by  $\mathcal{P}_1$  to obtain the blockchain and prune its last  $c$  blocks before returning it. Note that if there are less than  $c$  blocks, processes then return the genesis block. The properties of  $\mathcal{F}^{\diamond, c=0}$  trivially follow from the properties of  $\mathcal{F}^{\diamond, c}$  and the proposed transformation. ◀

► **Corollary 17.**  $\mathcal{F}^n$ ,  $\mathcal{F}^{\diamond, n}$ ,  $\mathcal{F}^{\diamond, c}$ , and  $\mathcal{F}^{\diamond, c=0}$  are equivalent.

**Proof.** Straightforward from Theorems 14, 15 and 16. ◀

## 4 (Eventual) Consensus Reductions

In this section, we show that guaranteeing  $\mathcal{F}^c$  is equivalent to solving Consensus, while guaranteeing bounded deferred finality (or any of the equivalent forms) is not weaker than solving Eventual Consensus.

### 4.1 $c$ -Bounded Deferred Finality and Consensus

► **Theorem 18.** *Guaranteeing  $\mathcal{F}^c$  is equivalent to solving Consensus.*

**Proof.** Let us first remark that  $\mathcal{F}^{c=0}$  is equivalent to BT Strong Consistency [1], which has been shown to be equivalent to Consensus [1].

To prove the theorem it is then sufficient to give a protocol  $\mathcal{P}_2$  that guarantees  $\mathcal{F}^{c=0}$  given a solution  $\mathcal{P}_1$  that satisfies  $\mathcal{F}^c$ , the other direction being trivial. We build  $\mathcal{P}_2$  by applying the same transformation of  $\mathcal{P}_1$  described in the proof of Theorem 16. The properties of  $\mathcal{F}^{c=0}$  trivially follow from the properties of  $\mathcal{F}^c$  and the proposed transformation. ◀

► **Corollary 19.** *There does not exist any solution that solves  $\mathcal{F}^c$  in an eventual synchronous system with more than  $1/3$  of Byzantine processes.*

**Proof.** The proof follows from the equivalence between  $\mathcal{F}^c$  and Consensus (cf. Theorem 18), which is unsolvable in a synchronous (and thus also in an eventually synchronous) system with more than one third of Byzantine processes [17]. ◀

### 4.2 Bounded Deferred Finality and Eventual Consensus

In this section we show that guaranteeing bounded deferred finality is not weaker than Eventual Consensus. To this aim we first recall the Eventual Consensus problem with a small modification of the validity property to make it suitable to the blockchain context and then we show that  $\mathcal{F}^{\diamond, c=0}$  (which is equivalent to  $\mathcal{F}^{\diamond, n}$  by Corollary 17) is not weaker than Eventual Consensus.

The Eventual Consensus (EC) abstraction [9] captures eventual agreement among all participants. It exports, to every process  $p_i$ , operations `proposeEC1`, `proposeEC2`, ... that take multi-valued arguments (correct processes propose valid values) and return multi-valued responses. Assuming that, for all  $j \in \mathbb{N}$ , every process invokes `proposeECj` as soon as it returns a response to `proposeECj-1`, the abstraction guarantees that, in every admissible run, there exists  $k \in \mathbb{N}$  and a predicate  $P_{EC}$ , such that the following properties are satisfied:

- **EC-Termination.** Every correct process eventually returns a response to `proposeECj` for all  $j \in \mathbb{N}$ .
- **EC-Integrity.** No process responds twice to `proposeECj` for all  $j \in \mathbb{N}$ .
- **EC-Validity.** Every value returned to `proposeECj` is valid with respect to predicate  $P_{EC}$ .
- **EC-Agreement.** No two correct processes return different values to `proposeECj` for all  $j \geq k$ .

► **Theorem 20.** *Guaranteeing  $\mathcal{F}^{\diamond, c=0}$  (or any of the equivalent forms) is not weaker than solving Eventual Consensus.*

**Proof.** We show that there exists a protocol  $\mathcal{P}_1$  that solves Eventual Consensus assuming the existence of a protocol  $\mathcal{P}_2$  that solves  $\mathcal{F}^{\diamond, c=0}$ . We do the transformation as follows. Every correct process  $p$  invokes `proposeECj` for all  $j \in \mathbb{N}$ . We impose that the validity predicate  $P$  of the blocktree ADT (see Section 3) be equal to predicate  $P_{EC}$ . When a correct process  $p$  invokes the `proposeECj(v)` operation of  $\mathcal{P}_1$ , for any  $j \in \mathbb{N}$ , then  $p$  executes the following sequence of three steps: (i)  $p$  invokes the `append(v)` operation of  $\mathcal{P}_2$ , then (ii)  $p$  invokes a sequence of `read()` operations up to the moment the `read()` returns a chain  $bc$  such that  $bc[j] \neq \perp$ , and finally (iii)  $p$  decides chain  $bc$  (i.e., it returns chain  $bc$ ) and triggers the next operation `proposeECj+1(v')`. We now show that protocol  $\mathcal{P}_1$  solves Eventual Consensus.

- **EC-Termination** This property is guaranteed by the ever growing tree property.
- **EC-Integrity** This property follows directly from the transformation.
- **EC-Validity** This property follows by construction and by the chain validity property since predicate  $P$  equals to predicate  $P_{EC}$ .
- **EC-Agreement** This property follows by the eventual prefix property and the 0-eventual revocation property, which guarantees that there exists a `read()` operation  $r$  such that all the subsequent ones return blockchains that are each prefix of the following one. In other words, eventually there is agreement on the value contained in  $bc[j]$ . This implies that there exists  $k$  for which all `proposeECj` with  $j \geq k$  return the same value to all correct processes.

Finally, by Corollary 17, the proof of the Theorem completes. ◀

► **Theorem 21.** *There does not exist any solution that solves  $\mathcal{F}^n$  (and any of the equivalent forms) in an asynchronous system with at least one Byzantine process.*

**Proof.** The proof follows from Corollary 17 and the fact that  $\mathcal{F}^{\diamond, c=0}$  is not weaker than Eventual Consensus (cf. Theorem 20). Since Eventual Consensus is equivalent to the leader election problem [9], which cannot be solved in an asynchronous system with at least one Byzantine process [23], this completes the proof of the Theorem. ◀

## 5 Finality Solutions

In this section we first show the impossibility of solving our weakest form of finality  $\mathcal{F}$  when the `append` operation, in case of forks, selects the “longest” chain. We then provide the first solution to  $\mathcal{F}$  with an unbounded number of Byzantine processes in an asynchronous system using an alternative selection rule.

### 5.1 Impossibility to Satisfy $\mathcal{F}$ with the Longest Chain Rule

In the following we prove that, in an asynchronous environment, we cannot provide  $\mathcal{F}$  if, in case of forks, the `append` selection function  $f_a()$  follows the longest chain rule, i.e., returns the longest chain of the blockchain tree. Note that this result holds even in absence of failures. Obviously we assume that blocks are not created using the Consensus abstraction: With Consensus, immediate finality is easily ensured, and thus no fork will ever occur. Thus, when the Consensus abstraction cannot be implemented (due to the adversity of the environment), many blockchain systems adopt a selection function  $f_a$  based on the longest chain. For instance, in proof-of-work systems such as Bitcoin, selected chains are the ones that have required the most amount of work, which is equivalent to the longest chains when

the difficulty is constant. In Ethereum, while the selection rule is based on heaviest sub-tree of the blockchain tree, or in proof-of-stake systems like EOS [12] or Tezos [11], the same argument applies.

To show this impossibility result, we consider a scenario in which the occurrence of any fork produces at most two alternative chains (this is often referred to as a branching factor of 2). We consider a finite number of processes and an append selection function  $f_a$  that in case of forks deterministically selects the longest chain through the `length` function (see Section 3.2.2), and in case of a tie selects the chain following any deterministic rule (for instance the chain whose last block has the smallest digest). We show that it is impossible to guarantee  $\mathcal{F}$  for such append selection function  $f_a$ .

Intuitively, the impossibility follows from the fact that with the longest chain selection rule, races can occur between different branches in the tree. We show that as forks may occur, we can create two infinite branches sharing only the root. One or the other branch constitutes alternatively the longest chain and `append` operations select chains from each branch alternatively. This is enough to show that the only common prefix that is returned is the root hence, violating eventual finality.

► **Theorem 22.** *It is impossible to guarantee  $\mathcal{F}$  if the `append` operation is based on the longest chain rule in an asynchronous environment.*

**Proof.** The interested reader is invited to read the proof in the Appendix of this paper. ◀

## 5.2 Asynchronous Solution Satisfying $\mathcal{F}$ with an Unbounded Number of Byzantine Processes

We consider an asynchronous system with a possibly infinite set of processes which can append infinitely many blocks, and processes can be affected by Byzantine failures. Each process has a unique identifier  $i \in \mathbb{N}$  and is equipped with signatures that can be used to identify the message sender identifier. Each block is identified with the identifier of the process that created it. Block identifier is inserted in the header of the block. Moreover, since it has been proven that reliable communications are necessary to ensure eventual finality [1], we assume that each process is equipped with an Eventually Reliable Broadcast primitive that satisfies the following two properties: If a correct process  $p$  broadcasts a message  $m$  then  $p$  eventually delivers  $m$  and if a correct process  $p$  delivers  $m$  then all correct processes eventually deliver  $m$ . Such a primitive can be implemented by organizing the infinite set of processes in a topology in which for each pair of correct processes, there exists a path composed by only correct processes [19]. Thus, we do not require any assumptions on the proportion between Byzantine and correct processes in the system but on the way those processes are arranged on the network topology.

The main idea of Algorithm 1 consists in using local selection functions  $f_a$  and  $f_r$  for `append` and `read` operations respectively and characterizing blocks by their parents and producer signatures.

To perform an `append` operation of a block  $b$ , correct processes extend the chain returned by function  $f_a$  applied on their current view of  $bt$  with  $b$ , i.e.,  $f_a(bt) \frown b$ , and `rb-broadcast`  $f_a(bt) \frown b$ . When a process `rb-delivers` a blockchain  $bc$ , it calls `bt.addIfValid(bc)` that merges  $bc$  with  $bt$  if the former is valid. By merging  $bc$  with  $bt$  we mean that for each block  $b_i$  of  $bc$  starting from the genesis block  $b_0$ , if  $b_i$  is not present in  $bt$  then  $b_i$  is added to  $bt$ , i.e.,  $b_i$  is added to the block of  $bt$  whose hash is equal to the one contained in  $b_i$ 's header. A `read()` operation triggered by a correct process  $p$  returns the chain selected by  $f_r$  on the current blocktree  $bt$  of  $p$ . Given a blocktree  $bt$ , the append selection function  $f_a$  selects a chain in  $bt$



■ **Algorithm 1** Guaranteeing  $\mathcal{F}$  with an unbounded number of Byzantine processes.

---

```

1 upon rb-delivery( $bc$ )
2   | bt.addIfValid( $bc$ )
3 end
4 upon append( $b$ )
5   | rb-broadcast( $f_a(bt) \frown b$ )
6 end
7 upon read()
8   | return  $f_r(bt)$ 
9 end

```

---

by going from the root (i.e., genesis block) to a leaf, choosing at each fork  $b_i$  the edge to the child with the lowest identifier. If more than one child have the same identifier (i.e., they have been created by the same process), then all of them are ignored. If all the children have the same identifier, then  $f_a$  returns the chain from the genesis block to  $b_i$ . Blocks are ranked by the creator identifier, in the domain of the natural number and thus lower bounded by 0. Then even though, an infinite number of blocks is added continuously to a fork, there is not, for a given block, an infinite number of blocks with a smaller identifier. Thus eventually the selection function  $f_a$  will always select the same prefix. Finally, since blocks are diffused by an eventually reliable broadcast primitive, eventually all correct processes will have the same view of the blocktree. When a process invokes the `read()` operation, it returns the blockchain selected by the read selection function  $f_r$  applied to its current view of the blocktree. By imposing that  $f_r = f_a$ , then eventually all the processes, when reading, will select the same prefix.

► **Theorem 23.** *Algorithm 1 is a solution satisfying  $\mathcal{F}$  in an asynchronous system with a possibly infinite set of processes which can append infinitely many blocks, and suffer from an unbounded number of Byzantine failures.*

**Proof.** We show by construction that Algorithm 1 solves  $\mathcal{F}$  in an asynchronous system with a possibly infinite set of processes which can append infinitely many blocks, and can suffer an unbounded number of Byzantine failures. Intuitively, despite the unbounded number of blocks in each fork, by the eventually reliable broadcast, eventually for each fork all correct processes have the same block with the smallest identifier. Hence, by the read selection function  $f_r$  that at each fork selects the block with the smallest identifier in order to select the chain to return, eventually, at all correct processes, function  $f_r$  returns the blockchain having a common increasing prefix. Let  $p_1, p_2, \dots$ , be a possibly infinite set of processes, such that each one maintains its local view  $bt_i$  of blocktree  $bt$  by running Algorithm 1. Then for any correct process  $p_i$  the following properties hold.

- **Chain validity:** it is satisfied by function `bt.addIfValid( $bc$ )` that merges blockchain  $bc$  to  $bt_i$  only if the former is valid.
- **Chain integrity:** The `read()` operation returns the chain of blocks selected by function  $f_r$  applied to  $bt_i$ . By Line 2 of Algorithm 1, only valid blocks are locally added to  $bt_i$  once they have been reliably delivered. By Algorithm 1, the only place at which blocks are reliably broadcast is in the `append()` operation.
- **Eventual prefix:** This property follows from the definition of  $f_a$  and the eventually reliable broadcast primitive. Thanks to the latter, for any  $b$  in the  $bt$  of a correct process  $p$ , eventually all correct processes deliver  $b$ . Let  $t_b$  be the time after which no process can

append further blocks  $b_{child}$  to  $b$  such that  $b_{child}$  is part of the chain returned by  $f_a$ . This time  $t_b$  always exists, as for each block  $b$  having potentially infinitely many children we have, by definition of function  $f_a$ , that  $f_a(bt)$  selects a chain in  $bt$  by going from the root to a leaf, choosing at each fork  $b$  the edge to the child with the lowest identifier. Since identifiers are lower bounded by 0, eventually function  $f_a$  will always select the same child  $b'$  of  $b$ . The same argument applies for  $b'$  and its children. Hence, if any two correct processes invoke the read operation infinitely many times, then as  $f_r = f_a$ , eventually they return chains that satisfy the eventual prefix property.

- **Ever growing tree:** This property relies on the fact that each fork has a finite number of blocks since there are finitely many processes and each (Byzantine or correct) process can contribute with at most one block per parent as multiple children created by the same process are ignored by  $f_a$ . Thus, eventually, new blocks contribute to the tree growth. ◀

### 5.3 Eventually Synchronous Solution Satisfying Bounded Deferred Finality with less than half of Byzantine Processes

In this section we prove that the bounded deferred finality is solvable in an eventually synchronous message-passing system with less than  $n/2$  Byzantine processes, where  $n$  is the number of processes.

We propose an algorithm, called  $\mathcal{AF}$  for Accountable Forking. This algorithm is inspired by the Streamlet [6] algorithm. Streamlet [6] assumes the presence of less than a third of Byzantine processes and an eventually synchronous system with a known message delay  $\Delta$  after GST. Algorithm  $\mathcal{AF}$  relies on weaker assumptions: we assume the presence of only a majority of correct processes and we do not explicitly use bound  $\Delta$ . We suppose that processes have access to the eventually reliable broadcast presented in Section 5.2. Prior to presenting our algorithm, we first recall the description of the original Streamlet [6].

**The Streamlet Algorithm.** The Streamlet algorithm works in an eventually synchronous system with a known message delay  $\Delta$  and a finite set of  $n$  processes. In particular, before the Global Stabilisation Time (GST), message delays can be arbitrary; however, after GST, messages sent by correct processes are guaranteed to be received by correct processes within  $\Delta$  time units. Each epoch, composed of  $2\Delta$  time units, has a designated leader chosen at random by a publicly known hash function. Each block  $b$  is labelled with the epoch ( $b.epoch$ ) at which it has been created. This allows processes to determine whether block  $b$  has been created by a legitimate leader. Algorithm 2 presents Streamlet protocol [6].

**The Accountable Forking ( $\mathcal{AF}$ ) Algorithm.** We propose  $\mathcal{AF}$ , an algorithm that extends Streamlet.  $\mathcal{AF}$  guarantees that for any given fork, correct processes can blame the process that originates it, i.e., a Byzantine process creating a fork is accountable for it. This is achieved as follows: First, we only require that a block gains votes from a majority of distinct processes to become notarized, which means that forks can occur. The second modification we propose goes deeper: if a fork occurs, any correct processes can detect the Byzantine process that originated it, and excludes it from the voters. Specifically, when two conflicting chains are finalized (i.e., two finalized chains that are not the prefix of one another) then processes look for inconsistent blocks. By definition, two notarized blocks  $b, b'$  are inconsistent with one another if one of the following two conditions holds:

- **Condition 1.**  $b$  and  $b'$  share the same epoch, i.e.,  $b.epoch = b'.epoch$ ;
- **Condition 2.** either  $((b.epoch < b'.epoch) \text{ and } (b.height > b'.height))$  or  $((b'.epoch < b.epoch) \text{ and } (b'.height > b.height))$ . Function height counts the number of blocks from the genesis block.

■ **Algorithm 2** Streamlet algorithm [6].

- 
- **Propose-Vote.** In every epoch:
    - The epoch's designated leader proposes a new block and reliably broadcasts it, extending the longest notarized chain (defined below) it has seen, or breaking ties arbitrarily if they have the same height.
    - Each process votes (rb-broadcasts a vote) for the first proposal it sees from the epoch's leader, as long as the proposed block extends (one of) the longest notarized chain(s) that the voter has seen. A vote is a signature on the proposed block.
    - When a block gains votes from at least  $2n/3$  distinct processes, it becomes notarized. A chain is notarized if its constituent blocks are all notarized.
  - **Finalize.** Notarized does not mean final. If in any notarized chain, there are three adjacent blocks with consecutive epoch numbers, the prefix of the chain up to the second of the three blocks is considered final. When a block becomes final, all of its prefixes must be final too.
- 

If a process votes for blocks inconsistent with one another then it is detected as Byzantine. Once a correct process  $p$  detects a Byzantine process  $q$ ,  $p$  ignores all messages coming from  $q$ . Since all messages received by a correct process  $q$  are eventually received by any correct process, then all of them do the same with respect to  $q$ .

▶ **Theorem 24.** *There exists a solution that satisfies  $\mathcal{F}^{\diamond, c=0}$  (and all the equivalent forms) in an eventually synchronous system with less than half Byzantine processes.*

**Proof.** We show in the Appendix that algorithm  $\mathcal{AF}$  is such a solution. ◀

## 6 Conclusion

In this work we have defined different consistency criteria for blockchains. We have first defined eventual finality, which is the weakest consistency criterion that we may expect from blockchains, along with the notion of block revocation. By combining eventual finality with different forms of revocation we obtained stronger consistency criteria, thus providing a comprehensive characterization of what we may expect from blockchains. We have formally shown that in an asynchronous system it is not possible to provide a known bound on the number of blocks that can be revoked. On the other hand, we have proposed for the first time a solution in an eventually synchronous system with less than half of Byzantine processes guaranteeing that eventually such bound is reached. We have also shown that in an asynchronous system, finality with no bound on the number of revocable blocks cannot be solved using the reconciliation rule of Bitcoin. Still we provide an asynchronous solution with an unlimited number of Byzantine processes. We hope that this work will better guide blockchain designs.

---

### References

- 1 Emmanuelle Anceaume, Antonella Del Pozzo, Romaric Ludinard, Maria Potop-Butucaru, and Sara Tucci Piergiovanni. Blockchain abstract data type. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2019.
- 2 Elli Androulaki and et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2018.

- 3 Antonio Anta Fernández, Kishori Konwar, Chryssis Georgiou, and Nicolas Nicolaou. Formalizing and implementing distributed ledger objects. *ACM SIGACT News*, 49(2):58–76, 2018.
- 4 Lăcrămioara Aștefanoaei, Pierre Chambart, Antonella Del Pozzo, Thibault Rieutord, Sara Tucci-Piergiovanni, and Eugen Zălinescu. Tenderbake – A solution to dynamic repeated consensus for blockchains. In *Proceedings of the Fourth International Symposium of Foundations and Applications of Blockchain*, 2021.
- 5 Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *CoRR*, 2017. [arXiv:1710.09437](https://arxiv.org/abs/1710.09437).
- 6 Benjamin Y Chan and Elaine Shi. Streamlet: Textbook streamlined blockchains. <https://eprint.iacr.org/2020/088.pdf>, 2020.
- 7 Jing Chen and Silvio Micali. Algorand: A secure and efficient distributed ledger. *Theor. Comput. Sci.*, 2019.
- 8 Tyler Crain, Vincent Gramoli, Mikel Larrea, and Michel Raynal. (leader/randomization/signature)-free byzantine consensus for consortium blockchains. *CoRR*, abs/1702.03068, 2017. [arXiv:1702.03068](https://arxiv.org/abs/1702.03068).
- 9 Swan Dubois, Rachid Guerraoui, Petr Kuznetsov, Franck Petit, and Pierre Sens. The weakest failure detector for eventual consistency. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, 2015.
- 10 Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Proc. EUROCRYPT International Conference*, 2015. Updated version 2020: <https://eprint.iacr.org/2014/765.pdf>.
- 11 L.M. Goodman. Tezos – A self-amending crypto-ledger, 2014.
- 12 Ian Grigg. EOS: An introduction. <https://whitepaperdatabase.com/eos-whitepaper/>.
- 13 Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovič, and Dragos-Adrian Seredinschi. The consensus number of a cryptocurrency. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing (PODC)*, 2019.
- 14 Maurice Herlihy. Blockchains and the future of distributed computing. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, 2017.
- 15 Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Proceedings of the Advances in Cryptology*, 2017.
- 16 Artem Koltsov, Vitaly Chermensky, and Stanislav Kapulkin. Casper White Paper.
- 17 Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 1982.
- 18 B. Liskov and S. Zilles. Programming with abstract data types. *ACM SIGLAN Notices*, 9(4), 1974.
- 19 Alexandre Maurer and Sébastien Tixeuil. On byzantine broadcast in loosely connected networks. In *Proceedings of the 26th International Symposium on Distributed Computing (DISC)*, 2012.
- 20 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. [www.bitcoin.org](http://www.bitcoin.org), 2008.
- 21 Rafael Pass, Lior Seeman, and Abhi Shelat. Analysis of the blockchain protocol in asynchronous networks. In *Proceedings of the EUROCRYPT International Conference*, 2017.
- 22 Matthieu Perrin. *Distributed Systems, Concurrency and Consistency*. ISTE Press, Elsevier, 2017.
- 23 Michel Raynal. Eventual leader service in unreliable asynchronous systems: Why? how? In *Proceedings of the IEEE International Symposium on Network Computing and Applications (NCA)*, 2007.
- 24 Alistair Stewart. Poster: Grandpa finality gadget. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS '19*, pages 2649–2651, 2019.
- 25 Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. <http://gawwood.com/Paper.pdf>.

## A Appendix

► **Theorem 22.** *It is impossible to guarantee  $\mathcal{F}$  if the append operation is based on the longest chain rule in an asynchronous environment.*

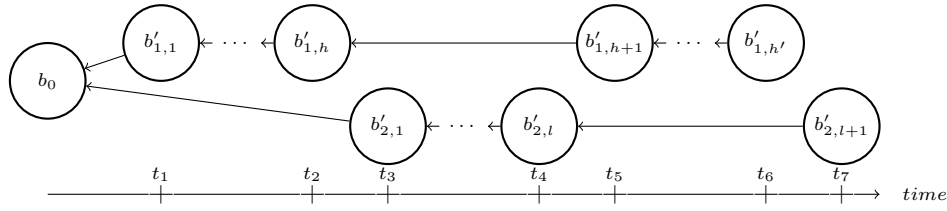
**Proof.** To capture the synchronisation power of the system, we abstract the deterministic creation of new blocks and their addition to the blockchain within an oracle. This oracle is the only generator of valid blocks, and regulates the number of appended children from a same parent. The same approach has been proposed in [1]. The branching factor of an oracle is the maximal number of children that can be appended to a block. The oracle owns a synchronization power equal to Consensus if its branching factor is equal to 1. The oracle grants access to the blocktree as a shared object, through the following three operations: `update_view()` returns the current state of the blocktree; `getValidBlock( $b_i, b_j$ )` returns a valid block  $b'_j$ , constructed from  $b_j$ , that can be appended to block  $b_i$ , where  $b_i$  is already included in the blocktree; and `setValidBlock( $b_i, b'_j$ )` appends the valid block  $b'_j$  to  $b_i$ , and returns  $\top$  when the block is successfully appended and  $\perp$  otherwise. The following theorem shows that, even with this strong oracle (that allows to have a bounded branching factor in contrast to proof-of-work (PoW) approaches), we cannot reach eventual finality if we rely on the longest chain rule to resolve forks.

In the proof we consider the stronger oracle allowing the occurrence of one fork, i.e., an oracle with branching factor equal to 2. That is, this oracle allows for two valid blocks to be appended to the same parent. If the oracle receives new requests to append additional blocks to this parent, it shall return  $\perp$  to all such requests.

Let  $p_1$  and  $p_2$  be two processes trying to append infinitely many blocks. Without loss of generality, we carry out this proof with a length function that counts the number of blocks from the genesis block.

We illustrate our proof with Figure 1. At time  $t_0$ , for both  $p_1$  and  $p_2$ , the `update_view()` of  $bt$  equals  $b_0$ , thus when both apply the append selection function  $f_a$  on it to select the leaf where to append the new block, they both get  $b_0$ . Then they both call `getValidBlock( $b_0, b_{i,1}$ ) = b'_i, where  $i = 1$  for  $p_1$  and  $i = 2$  for  $p_2$ . At time  $t_1 > t_0$ ,  $p_1$  and  $p_2$  are poised to call setValidBlock( $b_0, b'_{i,1}$ ). We then let  $p_1$  call setValidBlock( $b_0, b'_{1,1}$ ), which must return  $\top$  and hence  $b'_{1,1}$  is appended to  $b_0$ . Process  $p_1$  then proceeds to append a new block  $b_{1,2}$ , i.e., after having updated its  $bt$ 's view, through the update_view() function,  $p_1$  applies the append selection function  $f_a$  on it to select the leaf where to append its new block, in this case the only leaf is  $b'_{1,1}$ . It calls getValidBlock( $b'_{1,1}, b_{1,2}$ ) function which returns  $\{b'_{1,2}\}$  and it is poised to call setValidBlock( $b'_{1,1}, b'_{1,2}$ ).`

We let  $p_1$  continue to append new blocks until some time  $t_2$  at which it is poised to call `setValidBlock( $b'_{1,h}, b'_{1,h+1}$ )`, with  $h = 1$ , such that the length of the chain  $b_0, \dots, b'_{1,h+1}$  would be greater than or would have the same length but a larger lexicographical order than the chain  $b_0, b'_{2,1}$  if  $b'_{2,1}$  were already appended to block  $b_0$ . Afterwards, at time  $t_3 \geq t_2$ , we let  $p_2$  resume and complete its call to `setValidBlock( $b_0, b'_{2,1}$ )` which must also succeed and return  $\top$  as our oracle has a branching factor of 2. By construction,  $p_2$  sees the two branches in its following `update_view()` of  $bt$  (i.e., chain  $b_0, b'_{1,h}$  with  $h = 1$ , and chain  $b_0, b'_{2,1}$ ) of the same length thus the selection function  $f_a$  selects the branch  $b_0, b'_{2,1}$  for where to append the next block as block  $b'_{2,1}$  is smaller than  $b'_{1,h}$  in the lexicographical order. We let  $p_2$  append blocks to this branch until some time  $t_4$  at which it becomes poised to call `setValidBlock( $b'_{2,c}, b'_{2,c+1}$ )` with  $c = 2$  such that the length of the chain  $b_0, \dots, b'_{2,c}$  is smaller than the chain  $b_0, \dots, b'_{1,h+1}$ , or in case of equal length has a higher lexicographical order, and such that the length of the chain  $b_0, \dots, b'_{2,c+1}$  is greater than the chain  $b_0, \dots, b'_{1,h+1}$ , or in case of equal length has a smaller lexicographical order.



■ **Figure 1** A blocktree generated by two processes. On the x-axis the longest chain value of each chain at different time instants (from the root to the current leaf) and the relationships between those values.

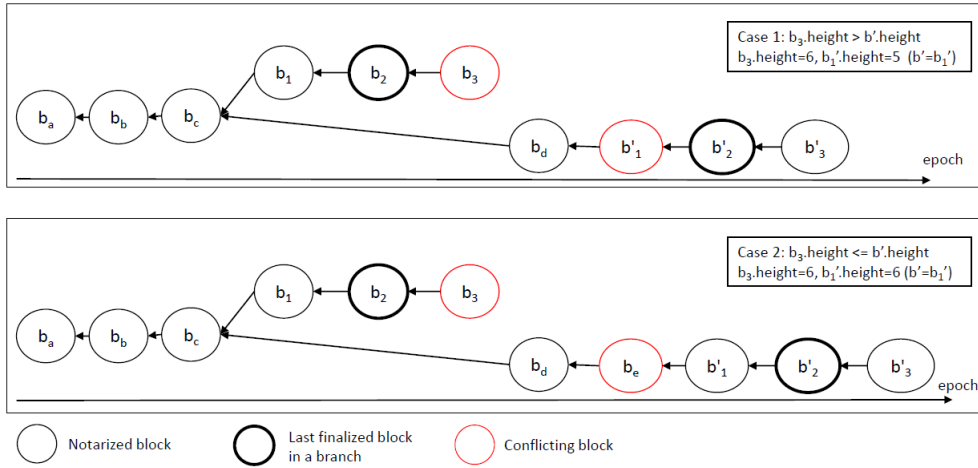
As before, it is time to stop the execution of  $p_2$  and resume the execution of  $p_1$  and to let it complete its call to  $\text{setValidBlock}(b'_{1,h}, b'_{1,h+1})$ . We can continue to create two infinite branches sharing only the root by alternatively letting  $p_1$  and  $p_2$  extend their own branch while stopping one and resuming the execution of the other each time its length would overcome the length of the other branch extended with the pending block (and the appropriate lexicographical orderings in case of equal length). This way we construct a tree composed of two infinite branches sharing only the root  $b_0$  as common prefix. It is easy to see that we can integrate read operations that may return the current chain from any branch as both branches are temporarily the longest one. Thus, the common prefix never increases, and so, the eventual finality consistency criterion is not satisfied.

It is important to note that with any **length** function that increases monotonically with prefixes (e.g. the length function could count the total number of transactions that belong to the blocks on the same branch) then this scenario still holds. In that case  $h$  and  $c$  in the proof could be larger than 1 and 2 respectively. ◀

► **Theorem 24.** *There exists a solution that satisfies  $\mathcal{F}^{\diamond, c=0}$  (and all the equivalent forms) in an eventually synchronous system with less than half Byzantine processes.*

**Proof.** Let us first demonstrate that voting for two inconsistent blocks  $b$  and  $b'$  is a Byzantine failure. We have two cases to consider. If both  $b$  and  $b'$  are inconsistent because Condition 1 holds, then the intersecting voters are Byzantine as correct processes vote only once per epoch. Hence if process  $q$  votes for  $b$  and  $b'$  then  $q$  is Byzantine. If both  $b$  and  $b'$  are inconsistent because Condition 2 is met, then the intersecting voters are Byzantine, as correct processes vote only for blocks extending one of the longest notarized chains. That is, if correct process  $p$  votes for  $b$  it means that  $b$  is extending a notarized block  $b_{pred}$  that is of height  $b.height - 1$ , therefore  $p$  cannot vote afterwards for a block  $b'$  whose height is strictly smaller than  $b.height$  because  $p$  must extend one of the longest notarized chain. It follows that if process  $q$  votes for both  $b$  and  $b'$  then  $q$  is Byzantine.

Let us now show that a fork occurs because of two inconsistent blocks. If there is a fork then this gives rise to two sequences of three adjacent blocks with consecutive epochs,  $b_1, b_2, b_3$  and  $b'_1, b'_2, b'_3$  (by construction given the finalization rule). If no blocks share the same epoch number then we can assume w.l.o.g. that  $b_3.epoch < b'_1.epoch$ . Let block  $b'$  belonging to the prefix of  $b'_3$  such that  $b'.epoch > b_1.epoch$  and  $b'.height$  is the smallest in the prefix of  $b'_3$ . Such block always exists as  $b'_1$  satisfies those two conditions. We have two cases: Either  $b'.height < b_3.height$  or  $b'.height \geq b_3.height$ . In the former case,  $b'$  is inconsistent with  $b_3$  since by assumption  $b'.epoch > b_3.epoch$ . In the latter case, the predecessor of  $b'$  is inconsistent with  $b_3$ . Indeed, the predecessor of  $b'$  has a strictly smaller height than  $b_1$  and by assumption has a larger epoch number than  $b_3$ . Figure 2 illustrates the presence



■ **Figure 2** Illustration of block inconsistencies due to the occurrence of a fork when the finalized blocks are not labelled with the same epoch. Epochs are on the  $x$  axis, and all consecutive blocks have consecutive epochs, e.g.,  $b_c$  and  $b_d$  have four epochs of difference, 4 and 7 respectively, while  $b_1$  and  $b_2$  are labelled with consecutive epochs.

of inconsistent blocks in presence of a fork at some block  $b_c$ . From  $b_c$  two chains are built, the first one consisting of the sequence of three blocks  $b_1, b_2$  and  $b_3$ , and the second chain consisting of four consecutive blocks  $b_d, b'_1, b'_2, b'_3$  (to illustrate the first case) and of five consecutive blocks  $b_d, b_e, b'_1, b'_2, b'_3$  (to illustrate the second case). In both cases block  $b'_1$  plays the role of block  $b'$ . In the first case (figure in the top),  $b_3.height = 6$  and  $b'.height = 5$  while  $b_3.epoch = 6$  and  $b'.height = 5$ . Thus Condition 2 applies. In the second case (figure in the bottom), since  $b'.height \geq b_3.height$  then there must exist some block  $b_e$  in the  $b'$  prefix. Thus  $b_e.height < b'.height$ . Given that by assumption  $b_e.epoch > b_3.epoch$ , then Condition 2 holds for  $b_e$  and  $b_3$ . Hence there is always a couple of inconsistent blocks in a fork.

Let us now conclude our proof that protocol  $\mathcal{AF}$  solves  $\mathcal{F}^{\diamond, c=0}$ . If a fork occurs, then each correct process eventually detects at least one Byzantine process and ignores its votes. Thus, the number of forks is finite since we have a finite number of Byzantine processes. As a consequence, there is always a single chain that is eventually finalized. As there is a majority of correct processes, algorithm  $\mathcal{AF}$  remains live as the original Streamlet one. Algorithm  $\mathcal{AF}$  also inherits the properties of the original Streamlet algorithm regarding the eventual finalization of blocks when the system becomes synchronous.

Finally, by applying Corollary 17, we complete the proof of the theorem. ◀





# Twins: BFT Systems Made Robust

Shehar Bano ✉

Facebook Novi, London, UK

Andrey Chursin ✉

Facebook Novi, Menlo Park, CA, USA

Zekun Li ✉

Facebook Novi, Menlo Park, CA, USA

Dahlia Malkhi ✉

Facebook Novi, Menlo Park, CA, USA

Alberto Sonnino ✉

Facebook Novi, London, UK

Dmitri Perelman ✉

Facebook Novi, Menlo Park, CA, USA

Avery Ching ✉

Facebook Novi, Menlo Park, CA, USA

---

## Abstract

This paper presents Twins, an automated unit test generator of Byzantine attacks. Twins implements three types of Byzantine behaviors: (i) leader equivocation, (ii) double voting, and (iii) losing internal state such as forgetting “locks” guarding voted values. To emulate interesting attacks by a Byzantine node, it instantiates *twin* copies of the node instead of one, giving both twins the same identities and network credentials. To the rest of the system, the twins appear indistinguishable from a single node behaving in a “questionable” manner. Twins can systematically generate Byzantine attack scenarios at scale, execute them in a controlled manner, and examine their behavior. Twins scenarios iterate over protocol rounds and vary the communication patterns among nodes. Twins runs in a production setting within DiemBFT where it can execute 44M Twins-generated scenarios daily. Whereas the system at hand did not manifest errors, subtle safety bugs that were deliberately injected for the purpose of validating the implementation of Twins itself were exposed within minutes. Twins can prevent developers from regressing correctness when updating the codebase, introducing new features, or performing routine maintenance tasks. Twins only requires a thin wrapper over DiemBFT, we thus envision other systems using it. Building on this idea, one new attack and several known attacks against other BFT protocols were materialized as Twins scenarios. In all cases, the target protocols break within fewer than a dozen protocol rounds, hence it is realistic for the Twins approach to expose the problems.

**2012 ACM Subject Classification** Security and privacy → Distributed systems security

**Keywords and phrases** Distributed Systems, Byzantine Fault Tolerance, Real-World Deployment

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2021.7

**Supplementary Material** All artifacts presented in this paper are made publicly available. Specifically, this includes: (i) the Rust implementation of LibTwins, the Twins framework we implemented for DiemBFT (Section 5); (ii) the artifacts (the AWS orchestration scripts, and microbenchmarking scripts and data) used to evaluate LibTwins (Section 6); and (iii) the Python simulator and Twins instantiation of safety flaw in Fast-HotStuff (Section 3).

*Software (Source Code):* <https://github.com/asonnino/twins-simulator>

archived at `swh:1:dir:fc8f63787defb25ffe9756fa666f9c7c49118519`

*Software (Source Code):* <https://github.com/diem/diem>

archived at `swh:1:dir:b59b22a1997118b87a99061664d6af4ce776f874`

**Funding** This work is funded by Novi, a subsidiary of Facebook.

**Acknowledgements** The authors would like to thank Ben Maurer, David Dill, Daniel Xiang, Kartik Nayak, Ling Ren, and Scott Stoller for feedback on late manuscript, and George Danezis for comments on early manuscript. We also thank the Novi Research and Engineering teams for valuable feedback.



© Shehar Bano, Alberto Sonnino, Andrey Chursin, Dmitri Perelman, Zekun Li, Avery Ching, and Dahlia Malkhi;

licensed under Creative Commons License CC-BY 4.0

25th International Conference on Principles of Distributed Systems (OPODIS 2021).

Editors: Quentin Bramas, Vincent Gramoli, and Alessia Milani; Article No. 7; pp. 7:1–7:29



Leibniz International Proceedings in Informatics

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

## 1 Introduction

Byzantine Fault Tolerant (BFT) protocols introduced in the seminal work of Lamport et al. [19] are designed to withstand attacks or arbitrary malfunction of internal nodes. However, creating Byzantine attacks in order to validate a BFT system is challenging: (*i*) Byzantine behavior is unconstrained and (*ii*) developers may be tainted by what they think that the system is designed to tolerate. Last, as a pragmatical consideration, developing code that implements Byzantine attacks might be risky.

This paper introduces Twins, a principled approach for effectuating Byzantine attacks on BFT systems and examining their behavior. Instead of coding incorrect behavior, Twins creates faulty behavior testing from the correct behavior itself, simply by duplicating **correct and unmodified** node behavior. This works as follows.

Twins creates a “faulty” nodes by deploying two (or generally,  $k$ ) instances, both having the same credentials/signing-keys but running autonomously. Thus, for example, both nodes can send messages in the same protocol round, but these messages will carry conflicting proposals or votes; to the rest of the system, this twins behavior will appear indistinguishable from an equivocating behavior by a single node. In another example, one twin may send a vote in one round, and its twin will “forget” it has voted in the next round; again, to the rest of the system, this will appear indistinguishable from a single node violating safety rules.

Twins is based on the insight that most interesting Byzantine attacks are internal and leverage knowledge of the expected behavior of participants, hence they go unnoticed. In particular, Twins foregoes trivial attacks such as sending semantically invalid messages, or sending a message without justification. Thus, leveraging existing code, Twins can automatically cover material Byzantine behaviors. Indeed, Section 3 demonstrates one new, and several known, attacks on BFT protocols materialized as Twins attacks. Crucially, in all cases, protocols break within fewer than a dozen protocol steps, hence Twins successfully exposes them. Note that Twins scenarios systematically iterate over protocol rounds and vary the communication patterns among nodes. While inherently exponential, in the above attacks, it took Twins only minutes to discover protocol flaws that in some cases, took the community decades to surface.

Twins has been incorporated into a production setting, DiemBFT [13], in which Twins can execute 44M Twins-generated scenarios daily. Whereas the system at hand did not manifest errors, subtle safety bugs that were deliberately injected for the purpose of validating the implementation of Twins itself were exposed within minutes. Twins can prevent developers from regressing correctness when updating the codebase, introducing new features, or performing routine maintenance tasks.

**Twins & attacks on BFT replication.** Twins arises in the context of BFT replication protocols. In this domain, several worrisome safety and liveness vulnerabilities were exposed recently [1, 23] in both known protocols [22, 17] and in new ones [2]. One reason that BFT replication lends itself well to analysis via Twins is as follows. A common paradigm underlying practical BFT replication protocols is a view-by-view design. Each view is driven by a designated leader proposing to the nodes and going through voting rounds by the nodes. If a leader is successful, a consensus decision is reached in the view. If not, nodes give up after a timeout and move to the next view. Transitioning to the new view/leader is tricky: A new leader must discover if the previous leader was successful, but it may be able to communicate only with a subset of the nodes. The transition logic turns out to be the source of problems in all the above cases, hence exposing the flaw requires only one or two leader rotations.

**Twins implementation.** Twins effectuates a Byzantine attack by a Byzantine node via instantiating *twin* copies of the node instead of one, giving both twins the same identities and network credentials. To the rest of the system, the twins appear indistinguishable from a single node behaving in a “questionable” manner. Twins minutely interacts with existing code to control message delivery and schedule various coarse-steps such as protocol rounds. It is practical to deploy in real systems as it uses existing node code, easily keeping up with an evolving software project.

We built an attack generator based on the Twins approach in the DiemBFT open-source project, the BFT replication core of the Diem payment system [13]. Implementing Twins in DiemBFT consists of two principal parts.

The first is a *scenario executor* that deploys a network configuration where some nodes have twins. The scenario executor hides twins behind a thin multiplexing wrapper; to the rest of the system, each pair of twins appear as a single entity. The scenario executor controls the scheduling of message deliveries according to a prescribed scenario. This is accomplished through a transport emulator in the DiemBFT repository called *Network Playground*.

The second part is a *scenario generator*. The scenario generator enumerates scenarios by varying the number of nodes and the message delivery schedule, then feeding the scenarios to the scenario executor. We describe in the paper several strategies for drastically reducing the number of scenarios through aggressive trimming of symmetrical scenarios. Among these strategies, one minimally “opens” the DiemBFT implementation and lets the scenario executor determine when a node acts as a leader in the consensus protocol. This removes duplicate scenarios that differ only in their leaders. Another strategy may allow only faulty nodes to become leader. Section 6 reports on our experience with Twins in DiemBFT.

**Coverage.** What attacks does the Twins approach capture? Developing a rigorous theory that answers this question is an intriguing question left for future work. Here, we provide anecdotal evidence of coverage in three forms:

- (i) Section 2 brings intuition and experience of several decades of work in the field. There are only a handful of ways in which a Byzantine attacker can materially deviate from the safety rules imposed by its protocol. For example, it can equivocate and send different proposals to different groups of recipients, or it can pretend it did not send/receive a message and propose or vote in a manner that conflicts with such a message.
- (ii) Evaluating within the DiemBFT production system Section 6 provides compelling validation of the Twins approach. Whereas the system at hand did not manifest errors, self-injected subtle safety bugs – for the purpose of validating the implementation of Twins itself – were exposed within minutes. In particular, we created a simple safety-violating setting by deploying  $f + 1$  (instead of  $f$ ) nodes with Twins, which led to an expected consistency violation within seconds. We further injected three subtle logical bugs, which only slightly deviated from the original specification. In all three cases, with only  $f$  twins (faults), Twins successfully exposed safety violations.
- (iii) Section 3.1 shows how Twins can instantiate a safety violation in a new protocol described in a recent manuscript [15]. This highlights the importance of systematically analyzing the properties of BFT protocols using Twins to expose subtle flaws. Section 3 reinstates several known attacks on BFT protocols using the Twins approach. These attacks cover a broad spectrum of vulnerabilities, e.g., safety, liveness, timing, and responsiveness.

In some protocol steps, a node may wait for messages to determine its next action. Under Twins, the node is forced to act according to the messages it received, as if the node provided a justification for each step in form of the history of messages it received. Deviating from

this behavior was not required to reinstate any of the attacks discussed in Section 3, though in principle, various deviating behaviors would not be covered by Twins. Another coverage challenge emerges in synchronous protocols because a node behavior may be based on real time. In such protocols, Twins essentially forces a node to behave in a timely manner. We tackle this case in one of the attacks investigated in Section 3 and demonstrate that nonetheless, a slight adaptation of the original attack reinstates the attack in Twins. However, we do not know yet which timing attacks may not be covered. We discuss some concrete future directions in Section 8 for extending Twins in the settings we explore as well as others.

## 2 Motivating the Twins Approach

We open this section with a quick primer on the Byzantine Fault Tolerant (BFT) replication problem, and describe the notation that will be used to describe attacks through the rest of this paper. We then provide high-level intuition on why Twins is a viable approach by showing the different kinds of Byzantine behaviors that can be captured by Twins. (Concrete attacks using Twins are described later in Section 3 and Section 6.1.)

**BFT Replication.** The goal of BFT replication is for a group of nodes to provide a fault-tolerant service through redundancy. Clients submit requests to the service. These requests are collectively sequenced by the nodes; this enables all nodes to execute the same chain of requests and hence agree on their (deterministic) output.

Except when specifically noted, we consider protocols that maintain safety against arbitrary delays in message transmissions. That is, we assume an *asynchronous network* setting. The main challenge is to drive *agreement* on a chain of requests (and their output) among all non-faulty nodes despite node failures. It is common to rely on leaders to populate the network with a unique proposal. During periods in which the leader is non-faulty and communication among the leader and non-faulty nodes is timely, this regime can drive consensus quickly. This approach is called *partial synchrony*, indicating that it maintains safety at all times and progress only during periods of synchrony.

In the Byzantine fault model, a node may crash or arbitrarily deviate from the protocol. In this setting, a BFT replication system implements a fault tolerant service via  $n$  nodes, of which a threshold  $f < n/3$  may be Byzantine. As Byzantine behavior is defined rather vaguely, there is no principled way to evaluate BFT systems. Twins is a new approach to systematically generate Byzantine attacks. The main idea of Twins is the following: running two (generally, up to  $k$ ) autonomous instances of a node that both use correct code and share the same identity, allows us to emulate most interesting Byzantine attacks. Two nodes share the same identity when they share the same credentials and signing keys.

**Notation.** Nodes are represented by capital alphabets (e.g.,  $A$ ) and the twin of a node is represented by the same alphabet with the prime symbol (e.g.,  $A'$ ). When referring to a set of nodes, we enclose them in parentheses e.g.,  $(A, B, B')$ . We underline a node that is serving as the leader, e.g.,  $\underline{A}$ . The adversary can delay and filter messages between nodes. We denote partitions of nodes by enclosing them in braces, e.g.,  $P_1 = \{A, B, C, D\}$  and  $P_2 = \{E, F, G\}$ , and reserve the capital letter  $P$  to denote them. Additionally, to show messages allowed in a given direction, we use the symbols  $\rightarrow$  and  $\leftrightarrow$ . For example,  $A \rightarrow (B, C)$  means  $A$  can send messages to  $B$  and  $C$ ; similarly,  $A \leftrightarrow P_2$  means  $A$  can send messages to and receive messages from any node of the partition  $P_2$ . The scenarios described below use a network configuration of 7 nodes,  $(A, B, C, D, E, F, G)$ . Byzantine nodes have twins denoted with ',

as in  $F'$ ,  $G'$ . To experiment with any of the deviating behaviors described below, one can increase the number of Byzantine faults to  $f + 1$  (say  $E, F, G$  have twins  $E', F', G'$ ) and expect to see conflicting commits.

**Equivocation.** A quintessential Byzantine behavior is for a node to *equivocate*. That is, in the same step, a Byzantine node might send different messages to different recipients.

Twins covers equivocation by splitting honest nodes between two partitions, each one communicating with only one twin of each pair. For example, we can split the system into  $P_1 = \{A, B, C, D, \underline{E}\}$ ,  $P_2 = \{C, D, E, \underline{F'}, G\}$ . The leader(s)  $F$  and  $F'$  execute correct leader code but nevertheless may generate conflicting proposals due to different inputs or randomness seeds. If there is a protocol flaw then these conflicting proposals could respectively commit in  $P_1$  and  $P_2$ , hence safety breaks.

**Amnesia.** An important role that nodes have in agreement protocols is *vote* for a single proposal per view. However, a Byzantine node might vote for a proposal and then “forget” that it has voted and vote again. Twins covers amnesia by letting one of the twins vote on one proposal. Since the other twin is oblivious to the vote happening, it may nevertheless – albeit executing correct code – vote on a different proposal.

More concretely, as in the scenario above, we can split the nodes into two partitions,  $P_1 = \{A, B, E, F, G\}$ ,  $P_2 = \{C, D, E, F', G'\}$ . If there is a protocol flaw then this double-voting behavior may result in conflicting commits in  $P_1$  and  $P_2$ , hence safety breaks.

**Losing internal states.** Another notable deviation for Byzantine nodes is to lose their internal state, particularly a *lock* that guards a value they voted for. Twins covers this deviation by letting one of the twins get locked on a value in one view, but in some subsequent view, bring the other twin who is ignorant that a lock exists.

More concretely, we can split the nodes into two partitions  $P_1 = \{A, B, E, F, G\}$ ,  $P_2 = \{C, D, E, F', G'\}$ . In one view, the adversary relays messages only among  $P_1$ . In the next view, it switches to  $P_2$ , causing  $F', G'$  – albeit executing correct code – to ignore their “previous” actions. This can repeat any number of times. If there is a protocol flaw then conflicting proposals may commit in different views, hence safety breaks.

### 3 Attacks Materialized in Twins

In this section, we demonstrate one new, and several known, attacks on BFT replication protocols, expressed as Twins scenarios. We provide insight into the attacks and defer the details of all but the linear leader-replacement attack to an appendix, due to space constraints.

#### 3.1 New Attack

Fast-HotStuff [15] is a new protocol, described in a recent manuscript. It is similar to HotStuff [32], except with a 2-phase commit rule. The safety violation we reveal using Twins is possible because Fast-HotStuff does not require consecutive rounds in order to commit. Specifically, Quorum Certificates (QCs) [32] formed by some of the (partitioned) nodes do not reach the other nodes, resulting in two parallel branches that eventually commit two conflicting blocks. We instantiate this safety violation with Twins (using only network partitions in a network with 4 nodes and within 11 rounds). This highlights the efficacy of systematically analyzing the properties of BFT protocols via Twins to expose subtle flaws. More details are provided in Appendix F.

We implemented the Fast-HotStuff BFT consensus algorithm in a Python simulator which we release as open source<sup>1</sup>. The simulator then executes Twins scenarios over the algorithm.

### 3.2 Reinstated Attacks

We present several known attacks on BFT protocols, expressed as Twins scenarios. In all cases, exposing vulnerabilities requires only a small number of nodes, partitions, rounds and leader rotations. It is worth noting that later, our evaluation (Section 6) of LibTwins, Twins implemented for DiemBFT, shows that running an automated scenario generator (Section 4.2) with these configurations would cover the described attacks within minutes. We did not undertake to re-implement all these protocols and apply a Twins scenario generator to them; our implementation covers only DiemBFT [13].

**Safety attack on Zyzzyva.** *Zyzzyva* broke new ground in BFT replication with the introduction of an optimistic single phase “fast track” commit. Eleven years elapsed from its publication until a safety flaw in *Zyzzyva* was discovered [1], during which numerous research projects and systems were built on it. Twins generates a scenario that exposes the flaw with 4 nodes and two leader rotations: the first leader equivocates via a twin, and the next two leaders drop messages to/from some nodes. The details of this attack using Twins is described in Appendix C.

**Liveness attack on FaB.** FaB [22], a precursor to *Zyzzyva*, is a view-based protocol with an optimistic fast track. Not surprisingly, a similar problem arises in FaB due to a flawed leader replacement protocol [1], albeit manifesting as a liveness bug. Twins exposes this bug in a short scenario with  $n = 4$  and three leader rotations, leading to a complete absence of leader proposals. The detailed attack using Twins is described in Appendix D.

**Timing attack on Sync HotStuff.** *Force-Locking Attack* [23] is a timing attack on a preliminary version of a synchronous BFT protocol named Sync HotStuff [2] (which was subsequently updated to resist the attack). As before, Twins captures this attack with only a small system size,  $n = 5$ , and two leader rotations. However, in order to create timing attacks, Twins needs to be aware of timing information for protocol steps and messages deliveries. Extending Twins with timing data is left for future work. In the specific attack at hand, course-grain timing at fixed intervals – fewer than ten – suffice to reinstate the attack. The detailed attack using Twins is described in Appendix E.

**Non-Responsiveness attack on linear leader-replacement.** Practical Byzantine Fault Tolerance (PBFT) [9] is a seminal work that was designed to work efficiently in the asynchronous setting. Carrying the classical PBFT solution to the blockchain world, Tendermint [7] and Casper [8] introduced a simplified *linear* strategy for leader-replacement. However, it has been observed [6, 31] that this strategy forgoes an important property of asynchronous protocols – *Responsiveness* – the ability of a leader to advance as soon as it receives messages from  $2f + 1$  nodes.<sup>2</sup> Indeed, bringing linear leader-replacement approach into PBFT, we demonstrate a liveness attack using a Twins scenario. Lack of progress is detected by observing that two consecutive views with honest leaders whose communication with a quorum is timely do not produce a decision. We present the details of this attack using Twins in the next section.

<sup>1</sup> <https://github.com/asonnino/twins-simulator/tree/master/fhs>

<sup>2</sup> Tendermint is a precursor to HotStuff [32] and DiemBFT [13] which operates in two-phase views, but has no Responsiveness. HotStuff/DiemBFT solve this by adding a third phase.

### 3.3 Non-Responsiveness Attack

We now describe in more detail the non-Responsiveness attack above on linear leader-replacement. The seminal PBFT solution operates two-phase views. A simplified, linear leader-replacement works as follows. A leader proposes to extend the highest *quorum certificate* (QC) it knows. A QC is formed on a proposed value if it gathers  $2f + 1$  votes from nodes. Nodes vote on the leader proposal if it extends the highest QC they know. A commit decision on the leader proposal forms if  $2f + 1$  nodes form a QC, and then  $2f + 1$  nodes vote for the QC. Progress is hinged on leaders obtaining the highest QC from the system, otherwise liveness is broken.

Using the notation from Section 2, the liveness attack here uses 4 replicas ( $D, E, F, G$ ), where  $D$  has a twin  $D'$ . In the first view,  $D$  and  $D'$  generate equivocating proposals. Only  $D, E$  receive a QC for  $D$ 's proposal. The next leader is  $F$  who proposes to re-propose the proposal by  $D'$ , which  $E$  and  $D$  do not vote for because they already have a QC for that height. Only  $F$  and  $D'$  receive a QC for  $F$ 's proposal. This scenario repeats indefinitely, resulting in loss of liveness. More specifically, this attack works as follows:

**View 1:** Initialize  $D$  and  $D'$  with different inputs  $v_1$  and  $v_2$ .

- Create the partitions  $P_1 = \{\underline{D}, E, G\}$ ,  $P_2 = \{\underline{D'}, F\}$ .
- Let  $D$  and  $D'$  run as leaders for one round.  $D$  proposes  $v_1$  to  $P_1$  and gathers votes from  $P_1$  creating  $QC(v_1)$ .  $D'$  proposes  $v_2$  to  $P_2$  and gathers votes but not a QC.
- Create the following partitions:  $P_1 = \{\underline{D}, E\}$ ,  $P_2 = \{\underline{D'}, F\}$ ,  $P_3 = \{G\}$ .  $D$  broadcasts  $QC(v_1)$ , which only reaches  $P_1$  i.e.,  $(D, E)$ .

**View 2:** Drop all proposals from  $D$  and  $D'$  until View 2 starts.

- Remove all partitions, i.e.,  $P = \{D, D', E, \underline{F}, G\}$ .
- Let  $F$  run as leader for one round.  $F$  re-proposes  $v_2$  (i.e.,  $D'$ 's proposal in the previous round) to  $P$ .  $(D, E)$  do not vote as they already have  $QC(v_1)$  for that height.  $F$  gathers votes from the other nodes and forms  $QC(v_2)$ .
- Create partitions  $P_1 = \{D, E\}$ ,  $P_2 = \{D', \underline{F}\}$ ,  $P_3 = \{G\}$ .
- $F$  broadcasts  $QC(v_2)$ , which only reaches  $P_2$ .

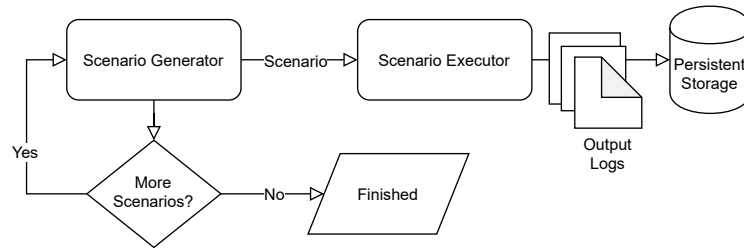
**View 3:** Drop all proposals from  $F$  until View 3 starts.

- Create the partitions  $P_1 = \{D, \underline{E}, G\}$ ,  $P_2 = \{D', F\}$ .
- Let  $E$  run as leader for one round.  $E$  proposes  $v_3$  which extends the highest QC it knows,  $QC(v_1)$ . As before,  $E$  manages to form  $QC(v_3)$ , but as a result of a partition, the QC will only reach  $(D, E)$ . Next, there is a view-change,  $F$  is the new leader, and there are no partitions.  $F$  proposes  $v_4$  which extends  $QC(v_2)$ , the highest QC it knows. However,  $(D, E)$  do not vote because  $v_4$  does not extend their highest QC i.e.,  $QC(v_3)$ . This scenario can repeat indefinitely, resulting in the loss of liveness.

## 4 Systematic Scenario Generation

Whereas the previous section demonstrated manually crafted Twins attack scenarios, this section presents a framework for systematically generating such scenarios.

Systematically and efficiently generating Twins scenarios that provide good coverage requires tailoring to the specific BFT protocol settings. We develop the Twins framework which generates and executes *scenarios* that describe the node and network configurations. Specifically, the Twins framework is comprised of two components as shown in Figure 1: (i) the scenario executor, and (ii) the scenario generator. The scenario executor runs a single scenario and generates output logs, while the scenario generator produces various scenarios that are fed to the scenario executor to check for violations. The following design goals underlie the Twins framework:



■ **Figure 1** Twins high-level design.

- **Generic & Modular.** Twins is modular with respect to the particular BFT protocol implementation being analyzed, imposes as little complexity as possible on the development, and easily keeps up with code changes.
- **Parametrizable.** The network setup (i.e., the number of nodes, leaders per round, and network configuration per round) and adversarial assumptions (i.e., how many Byzantine faults are tolerated) is configurable.
- **Feasible.** Twins allows pruning duplicate scenarios in order to provide coverage of material attacks.
- **Customizable Coverage.** The coverage of scenarios, i.e., the subset of all possible scenarios to choose for execution, is configurable by randomly sampling scenarios to run among all possible enumeration.
- **Reproducible.** Twins writes logs to persistent storage, containing sufficient information to detect and reproduce any safety violations.

Next, we describe the two main components (Figure 1) of Twins– the scenario executor and the scenario generator– in detail.

#### 4.1 Scenario Executor

In every Twins scenario, a threshold of the nodes are “misconfigured” to have a twin instance with identical transport endpoint credential and secret keys. The Twins scenario executor gets as input a scenario consisting of a node-set, a subset of which are marked *compromised* (representing Byzantine nodes); and a round-by-round message delivery schedule. The scenario executor sets up a network of nodes with a given number of compromised nodes and per round partitions and leaders. The compromised nodes correspond to the nodes for which the scenario executor creates twins (i.e., identical instances with the same credentials and signing keys), thereby emulating misbehavior.

As mentioned above, we address BFT replication protocols that proceed in rounds initiated by a designated leader, each round representing a state transition in the protocol’s state machine replicated on each node. For each round, the scenario executor creates a given network partition and assigns given leaders to the round. The scenario executor runs the BFT protocol among nodes for a pre-specified number of rounds, at the end of which, the scenario executor checks for violations. Specifically, protocol guarantees can be violated in two principal ways, safety and liveness. A safety violation is detected if two nodes commit to conflicting decisions. A liveness violation can be detected if the protocol fails to commit within a certain number of steps or a certain duration bound.



## 4.2 Scenario Generator

We build a scenario generator of round-by-round scenarios: for each round, the scenario generator enumerates possible leaders and message delivery schedules among nodes. The scenario generator produces various scenarios to be fed into the scenario executor. Each scenario represents a unique instance of executor configuration parameters, i.e., the compromised nodes and per round network partitions and leaders. Scenarios are generated systematically as follows (see notations in Section 2):

- **Step 1.** The scenario generator first produces the set of all possible partitions of nodes (called *partition scenarios*). For example, for a network of 4 nodes  $(A, B, C, D)$ , possible partition scenarios  $(P)$  include  $\{P_1 = \{A, D\}, \{B, C\}\}$ , and  $P_2 = \{\{A\}, \{B, C, D\}\}$ . This problem relates to the *Stirling Number of the Second Kind* [27] which enumerates the ways in which a set of  $N$  nodes can be divided up into  $P$  non-empty partitions, where  $P$  ranges from  $N$  (i.e., each node is self-isolated) to 1 (i.e., fully connected network without partitions).
- **Step 2.** Next the scenario generator assigns each partition scenario to all possible leaders i.e., the set of  $N$  nodes assuming any of those can be a potential leader. For example, for the example partition scenario above  $\{P_1 = \{A, D\}, \{B, C\}\}$  for a network of nodes  $(A, B, C, D)$ , possible leader-partition combinations include  $\{\underline{A}, P_1\}$ ,  $\{\underline{B}, P_1\}$ ,  $\{\underline{C}, P_1\}$ ,  $\{\underline{D}, P_1\}$ . Each leader-partition combination fully describes the Twins configuration required for each round.
- **Step 3.** The scenario generator lists scenarios by enumerating all possible ways in which the leader-partition pairs generated in the previous step can be arranged over  $R$  rounds (i.e., permutation, with or without replacement).

The scenario generator iterates over the generated scenarios linearly, and invokes the scenario executor for each scenario. For safety analysis, usually a small number of rounds ( $< 10$ ) suffices to expose logical bugs in the protocol. Scenario generators therefore need to enumerate a reasonable number of combinations.

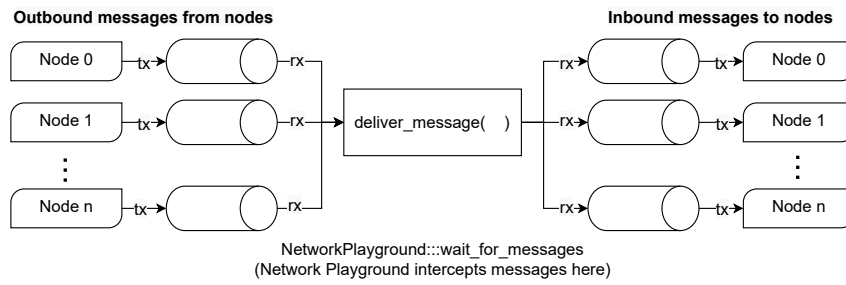
**Pruning scenarios.** Important to the success of the approach is for the scenario generator to avoid duplicate scenarios (e.g., in symmetry or node label<sup>3</sup> rotation) and generate only materially different scenarios. The implementation we describe in the Evaluation section of this paper (Section 6) employs aggressively such pruning. Certain heuristics further substantially reduce the number of scenario configurations. For example, in most safety violations the set of honest parties is split into two, hence it suffices to play with two or three partitions per round. These optimizations make it feasible to cover a broad range of meaningful scenarios. For analyzing liveness, many scenarios will obviously fail to make progress because there does not exist a super-majority quorum that has reliable and timely communication among its members. Hence, for liveness analysis the scenario generator must guarantee that eventually such a quorum exists.

**Message delays and timeouts.** We note that the scenario generator does not address message delays and timeouts, only the dropping of messages and their relative delivery order. Because the BFT protocol may employ timers, the dropping of messages implicitly implies

---

<sup>3</sup> Nodes can have designated roles in the protocol, referred to as *node labels*. Twins incorporates the label “leader”, which is the case for standard BFT protocols. Extensions of these protocols might have further hierarchy e.g., primary and secondary leaders. This is currently not supported, but the scenario generator can be easily extended to support different node labels.

## 7:10 Twins: BFT Systems Made Robust



■ **Figure 2** Design of DiemBFT's *Network Playground*.

that relevant endpoint incur a violation of presumed bounds on transmission delays. Future work may incorporate explicit message delays into the scenario generator to check specific timing violations and also to analyze BFT protocols in the synchronous model (Section 8).

## 5 Implementation

We implemented the Twins framework for DiemBFT, which we call LibTwins. Appendix A provides an overview of DiemBFT. As described in Section 4, an implementation consists of two principal ingredients, a scenario generator and an scenario executor (Figure 1). We first describe the scenario executor implementation which leverages a network emulator in DiemBFT referred to as the *network playground*. We then proceed to describe the scenario generator implementation. For completeness, the Rust code and interfaces for the main functions of LibTwins, `execute_scenario` and `scenario_generator`, are provided in Appendix B. We are open sourcing the Rust implementation of LibTwins<sup>4</sup>.

### 5.1 Scenario Executor

The LibTwins scenario executor leverages the network emulator of DiemBFT, *network playground*<sup>5</sup>. Network playground provides an apparatus for running single-host DiemBFT deployments, emulating a network and intercepting all messages exchanged between nodes. Scenarios can be written to manipulate the intercepted messages (e.g., by dropping certain messages) and observe node response. Figure 2 shows the design of the network playground. Nodes are represented by processes run on different threads (that run the full consensus protocol), and network links between them are expressed as Rust channels that provide asynchronous unidirectional communication between threads. In DiemBFT, nodes are identified by their *Account Address* (a public key that uniquely identifies a node). Channels are associated with their respective account addresses (nodes). When a node starts a new round, it checks whether it is leader for this round; if yes, then it generates on the fly a block to propose using a mock block generator. Each call to the mock block generator produces a different block. This has important implication for LibTwins, as we require a node and its twin to propose different blocks at the same round to emulate equivocation.

The scenario executor component (Section 4) of LibTwins is built on top of network playground. This required the following modifications and extensions to the original library:

<sup>4</sup> <https://github.com/diem/diem>

<sup>5</sup> [https://github.com/diem/diem/blob/master/consensus/src/network\\_tests.rs](https://github.com/diem/diem/blob/master/consensus/src/network_tests.rs)

- **Adding twins.** We wrote a new method to add nodes to the network that supports twins. The method takes “compromised nodes” as a parameter to refer to the nodes for which to create twins. For each target node, a duplicate instance is created with the same credentials and signing keys. Consequently, in the eyes of the other nodes the compromised node and its twin are indistinguishable.
- **Inferring rounds.** LibTwins requires to apply a number of filtering policies at the round level. Network playground does not have a notion of rounds – it only supports static configurations that remain unchanged throughout protocol execution. There is no global notion of rounds in a distributed system with partial synchrony; instead, nodes have their own view of which round they are in, which they include in their messages. We enable network playground to extract round from intercepted messages and accordingly apply filtering criteria.
- **Round-based message filtering.** Network playground allows writing rules to drop intercepted messages that meet certain criteria, i.e., messages to or from specified nodes and messages of specified types e.g., votes or proposals. LibTwins extends network playground to drop intercepted messages *per round*, which allows emulating different network partitions per round. The message dropping rules treat compromised nodes and their twins differently – the rules apply to account addresses (which uniquely identify nodes), not public keys (which are the same for a target node and its twins).
- **Deterministic multi-leader election.** DiemBFT currently uses a non-deterministic leader election algorithm. LibTwins requires leader election at a finer granularity, i.e., assigning a specified leader to each round, potentially assigning multiple leaders to a round (because if a compromised node is elected as a round leader, its twins becomes leader too). We wrote a new leader election algorithm for DiemBFT that supports these requirements.

To emulate running the protocol for a given number of rounds, we approximate rounds by the number of messages emitted by nodes. Note that in a system with partial synchrony, we can only make guesses about rounds as there is no global notion of rounds. Using message-count per-round (without partitions) as an “over-guesstimate”, we let the nodes vote for 3 extra rounds. Over-running a scenario has no consequence on the results of LibTwins (other than longer scenario execution time) because any safety violations would have already been detected in earlier rounds.

## 5.2 Scenario Generator

The scenario generator produces scenarios in three main steps. First, it generates all the possible ways in which a set of  $N$  nodes can be split into  $P$  partitions (partition scenarios). Second, it generates all possible ways in which  $L$  leaders can be combined with the partitions generated in the previous step. Finally, it generates all the possible ways in which the partition-leader combinations can be permuted over  $R$  rounds of consensus protocol execution. The scenario generator can operate in online or offline modes. In the online mode, scenarios are generated on the fly and fed to the scenario executor. The scenario generator can be configured to write the scenarios to a file. In the offline mode, the scenario generator reads previously generated scenarios from a file and feeds them to the scenario executor.

**Pruning scenarios.** A naïve enumeration of all combinations of  $P$  partitions,  $L$  leaders, and  $R$  rounds may explode quickly (see Table 1). In order to constrain the number of generated scenarios in a particular run, we provide hooks to control the number of  $P$  partitions,

the number of  $L$  leader-partition pairs, and the number of leader-partition configuration assignments to rounds. For all three cases, we specify whether the selection is deterministic – first  $X$  – or randomized – an  $X$  sample. In the third case – configuration assignment to rounds – the total combination space to select from is large. Therefore, the scenario generator allows randomizing the per-round configuration selection, rather than sampling over the entire space of assignments.

## 6 Evaluation

We validate the capability of LibTwins to model and detect attacks, present microbenchmarks for the main components of LibTwins, and describe our experiments at scale using Amazon Web Services (AWS) [4]. We are open sourcing the implementation of LibTwins, AWS orchestration scripts, and microbenchmarking scripts and data to enable reproducible results<sup>6</sup>.

All our evaluations correspond to 4–7 nodes, 4–7 rounds and 2–3 partitions. Intuitively, these configurations seem sufficient to expose any safety violations. Indeed, the known attacks on BFT protocols described in Section 3 were exposed with only a small number of nodes, partitions and leader rotations. A recent work [25] on the coverage of random scenarios to detect crash faults shows that coverage depends on the number of partitions and node labels (in our case, the leaders), but not on the number of nodes. For Jepsen [16], all the bugs that provide meaningful coverage have a small number of rounds, and 2–3 partitions and roles [25]. Using higher values for these parameters leads to a very large number of scenarios, which cannot be feasibly executed without some sort of filtering (Section 5.2). It is an interesting open question whether increasing the value of these parameters has a higher chance of exposing safety violations.

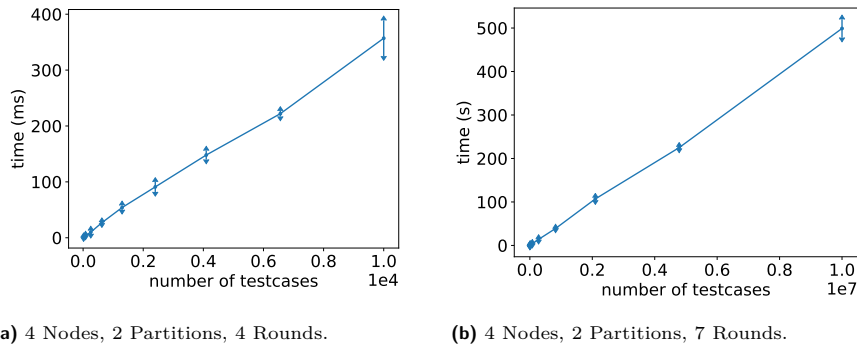
### 6.1 Validation

We deliberately introduce bugs to DiemBFT, and validate that LibTwins is able to model and detect attacks that exploit the injected vulnerabilities. This approach is similar to *mutation testing*, a well-known technique to evaluate the quality of existing tests in terms of whether they can detect programs with deliberately injected modifications (called “mutants”). While approaches such as automated mutation testing can help us to exhaustively introduce mutants, this is computationally expensive and not practical for large, complex systems. We select bugs to inject into DiemBFT based on their ability to compromise the program’s functional correctness. We note that this choice is based on our intuition and experience, and does not provide any coverage guarantees. The validation approach we use is to: (i) inject the bug into DiemBFT; and (ii) generate scenarios using the LibTwins scenario generator, checking for any safety violations. We instantiate the scenario generator with different configurations and vary them until a safety violation is exposed.

We begin with the base case: can LibTwins generate a scenario that violates safety when the BFT threshold is exceeded (i.e.,  $> f$  Byzantine nodes)? We discovered a safety violation with 4 nodes and 2 twins ( $\underline{A}, B, C, D, \underline{A}', B'$ ), 7 rounds, and static scenario configuration (i.e., each partition-leader combination is run for all  $R$  rounds). LibTwins executed 62 scenarios of which 8 led to safety violation within 86s.

---

<sup>6</sup> <https://github.com/libra/libra>



■ **Figure 3** Time taken by the scenario generator to produce LibTwins scenarios. Each data point is the average of 10 runs; error bars represent one standard deviation.

**Changing quorum size to  $2f$ .** BFT protocols consider a state transition safe if it receives votes from an honest majority of nodes (i.e., quorum). We change DiemBFT’s quorum size from  $2f+1$  to  $2f$ . LibTwins detects a safety violation with 4 nodes and 1 twin ( $\underline{A}, B, C, D, \underline{A}'$ ), 7 rounds, and static scenario configuration (i.e., where each partition-leader combination is run for all the  $R$  rounds). Within 20s, LibTwins executes 14 scenarios of which 6 lead to safety violation. These scenarios have the same pattern: Nodes are split into two partitions of size 2 and 3, with  $A$  in one partition and  $A'$  in the other. As nodes in the two partitions can form quorum, oblivious to each other they continue to generate quorum certificates on blocks proposed by  $A$  and  $A'$ , respectively. Ultimately, nodes in the two partitions commit two different blocks.

**Accepting conflicting votes.** Upon receiving a proposal, nodes vote for it only if the *block\_round* is greater than the *last\_voted\_round* (Safety Rule 1, Appendix A). We introduce a subtle bug to DiemBFT by changing this rule, so that a node votes for a block if the *block\_round* is greater than or equal to the *last\_voted\_round*. LibTwins detects the safety violation within a few seconds, with 4 nodes and 1 twin  $\{\underline{A}, B, C, D, \underline{A}'\}$ , and 7 rounds. This safety bug was detected in one-shot, with 0 partitions. Nodes vote on proposals from both  $A$  and  $A'$  and quickly end up committing two different proposals for the same round.

**Forgetting to update preferred round.** Upon receiving a proposal, nodes vote for the block if the *block\_round* is greater than *last\_voted\_round*, and the block’s *parent\_round* is greater than or equal to *preferred\_round* (Safety rules 1 and 2, Appendix A). We disable the first check, and bypass the second check by never updating *preferred\_round* so it permanently remains at 0 (Update rule 2, Section A). The main ingredient of an attack that exploits the bug described above is to propose a block in an old round, and get the nodes to *over-write* committed blocks (safety violation). The challenge for LibTwins is that as a twin node runs correct code, it cannot be made to propose blocks in arbitrary rounds. One option is to partition the twin node in an old round, and bring it back up in a later round, so it starts proposing blocks from where it left. This is, however, not possible in a “full disclosure” protocol like DiemBFT where each quorum certificate (or timeout certificate) contains the full history of previous messages that led to the certificate. That is, as soon as  $A'$  recovers from the partition, it receives a quorum certificate (or timeout certificate) from other nodes and advances its round. To emulate  $A'$  going back in time and proposing a block for an older round, we let it run as leader for a few rounds, crash it, and then recover

■ **Table 1** The number of LibTwins scenarios generated for various configurations. Steps 1, 2 and 3 correspond to the scenario generation pipeline described in Section 4. **Step 1:** The number of ways in which  $N$  nodes can be distributed among  $P$  partitions. **Step 2:** The number of ways in which the partitions generated in Step 1 can be combined with leaders. **Step 3:** The number of ways in which the partition-leader pairs generated in Step 2 can be permuted (with and without replacement) over  $R$  rounds. In **Static** configurations, each partition-leader pair is statically configured for all the  $R$  rounds.

Nodes	Twins	Partitions	Rounds	Step 1	Step 2	Step 3		
						No Repl.	Repl.	Static
4	1	2	4	15	15	$\sim 3 \times 10^4$	$\sim 5 \times 10^4$	15
4	1	3	4	25	25	$\sim 3 \times 10^5$	$\sim 4 \times 10^5$	25
4	1	2	7	15	15	$\sim 3 \times 10^7$	$\sim 2 \times 10^8$	15
4	1	3	7	25	25	$\sim 2 \times 10^9$	$\sim 6 \times 10^9$	25
7	2	2	4	255	510	$\sim 7 \times 10^{10}$	$\sim 7 \times 10^{10}$	510
7	2	3	4	3,025	6,050	$\sim 1 \times 10^{10}$	$\sim 1 \times 10^{15}$	6,050
7	2	2	7	255	510	$\sim 9 \times 10^{18}$	$\sim 9 \times 10^{18}$	510
7	2	3	7	3,025	6,050	$\sim 3 \times 10^{26}$	$\sim 3 \times 10^{26}$	6,050

it again as leader. When  $A'$  comes back up again it starts from round 0, proposing a block that builds on the *genesis* block (the first committed block). Because of our modifications to the *preferred\_round* and *last\_voted\_round* checks, the nodes re-write history.

## 6.2 Microbenchmarks

We present microbenchmarks for the two main components of LibTwins: scenario generator (Section 5.2) and scenario executor (Section 5.1). The microbenchmarks are run on an Apple laptop (MacBook Pro) with a 2.9 GHz Intel Core i9 (6 physical and 12 logical cores), and 32 GB 2400 MHz DDR4 RAM.

**Scenario generator microbenchmarks.** The scenario generator incurs a one-time computational cost – once the scenarios are generated, the scenario generator feeds them one by one to the scenario executor. Table 1 shows the number of scenarios generated with different configurations. We observe that the number of nodes and the number of rounds significantly increase the output of Step 1, which increases proportionally in the number of twins (as we only configure nodes with twins to become leaders). We find that non-static configurations in Step 3 cause the number of scenarios to explode. Therefore, of the various filters implemented for the scenario generator (Section 5.2), we find the filter at Step 2 to be most useful. We use this filter to make our at-scale Twins analysis (Section 6.3) feasible. Note that this inevitably comes at the cost of completeness of coverage – a trade-off that we cannot completely eliminate. Figure 3 shows how long the scenario generator takes to produce scenarios for the same number of nodes (4) and partitions (2), and 4 (Figure 3a) and 7 (Figure 3b) rounds. We observe that while it expectedly takes longer to generate scenarios for 7 rounds vs. 4 rounds due to a larger number of permutations, for each case the time taken increases linearly in the number of scenarios. We observe a similar linear trend in our microbenchmarks for other configurations with varying number of nodes and partitions (figures not included due to space constraints).

**Scenario executor microbenchmarks.** Table 2 shows the time the scenario executor takes to execute a scenario. We repeat each measurement over 100 randomly selected scenarios from a configuration with 2 partitions, and varying number of nodes (4 and 7) and rounds (4–12). We observe that for 4 nodes, the execution time ranges from 234–465ms for 4–12 rounds, with a maximum standard deviation of 314ms. For 7 nodes, the execution time ranges from 547–748ms for 4–12 rounds, with a maximum standard deviation of  $\sim 1.2$ s.

■ **Table 2** The time scenario executor takes to execute a scenario for 4 and 7 nodes, over varying number of rounds and fixed partitions (=2). Each measurement is repeated for 100 randomly selected scenarios.

Rounds	4 Nodes		7 Nodes	
	Mean (ms)	Std. (ms)	Mean (ms)	Std. (ms)
4	239	314	547	1,286
5	250	87	555	1,059
6	284	88	555	802
7	296	87	559	752
8	334	209	647	810
9	363	175	643	557
10	398	222	653	539
11	433	168	718	570
12	465	179	748	223

The variation observed above in execution times is expected because of how DiemBFT handles timeouts (Appendix A). For each scenario, LibTwins runs DiemBFT until it has observed a given number of messages (proposals and votes), which roughly corresponds to the number of rounds. In some scenarios, LibTwins can quickly pull out the given number of messages and finish the scenario in a timely manner. In other scenarios, we might end up with partitions where the nodes are not able to make progress and advance rounds, due to frequent round failures and increased timeout values. Some scenarios may take longer to run, waiting for the network to emit enough messages to conclude the scenario. The execution of scenarios has negligible ( $< 0.1\%$ ) memory and CPU footprints.

### 6.3 Running Scenarios at Scale

We evaluate LibTwins at scale, by running it against the correct code of DiemBFT. We executed 44M scenarios which were randomly selected from the 200M scenarios corresponding to the third row of Table 1 (that is, with 4 nodes, 2 partitions, 7 rounds, permuted with replacement). We first generated all the 200M scenarios and randomly selected 44M samples. We ran the scenario generator in offline mode so the scenarios are written to file rather than being passed to the scenario executor. We then split the generated scenarios into 20 shards. The scenarios can be easily sharded, as the scenarios are independent of each other – this implies that subject to the availability of computing power to generate and execute scenarios, LibTwins can be scaled up arbitrarily via sharding. We execute the sharded scenarios over 20 parallel instances of LibTwins on AWS. We use `t3.2xlarge` instances with 8 vCPUs, 2.5 GHz, Intel Skylake P-8175; 32 GB of RAM, and 300 GB of SSD storage. All machines run a fresh installation of Ubuntu 18.04. We did not observe any safety violations.

## 7 Related Work

There are two typical approaches to validate distributed systems. The first approach is to offer strong guarantees by building a fully verified system from the ground up [18, 26], or to show the absence or presence of bugs [29, 11, 10, 21] by exhaustively enumerating the space of system behaviors [5, 30] under systematically injected faults [3, 20].

Fully verified systems do not scale to systems deployed in the real world. Model checking and exhaustive enumeration of distributed system faults (especially, Byzantine arbitrary behavior) leads to state explosion (despite partial order reduction techniques [14]), resulting in low performance. This motivates the second approach of random validation, which underlies the discipline of *Chaos Engineering*, exemplified by systems like Chaos Monkey [24]. The

main idea is to analyze the resiliency of a distributed system by randomly injecting faults (e.g., terminating processes). Turret [20] refines this idea by focusing on performance attacks. It runs an attack-finding algorithm using different strategies, ranging from simple brute-force to more sophisticated “greedy search” algorithms. Jepsen [16] is a blackbox analysis framework that runs processes with a random, auto-generated workload and randomly injected network partitions. A related approach is to subject the system being evaluated to *trials by fire* such as Cosmos Game of Stakes [12], i.e., financially incentivizing the community to attack the “mock” network, and analyzing successful attacks to harden the network. Random validation is effective and scalable – but it is not comprehensive or reproducible, and cannot be used to evaluate distributed systems in an ongoing fashion.

Prior work (with the exception of Jepsen) focused on crash faults. Twins is a new, principled approach to validate BFT systems by emulating Byzantine behavior via twins–copies of “compromised” nodes that can send duplicate or conflicting messages. Twins advances state-of-the-art by providing a framework to systematically generate scenarios with configurable coverage, and only modeling correct executions (thus avoiding the state explosion problem associated with formal methods). We show with extensive evaluations that Twins is suitable for evaluating real-world systems, and can be scaled up arbitrarily for larger scenario coverage. Twins automatically generate scenarios that modify the interaction of components with the environment, without opening the code.

## 8 Future Work & Conclusion

Twins is a novel approach to systematically analyze BFT systems. It provides coverage for many, but not all, Byzantine attacks. The paper demonstrated anecdotal evidence of coverage with respect to several known Byzantine attacks, and an implementation of Twins for DiemBFT that exposes misconfiguration and purposely injected logical bugs within minutes. Many directions are left open for future extensions.

**Theory of Twins coverage.** As mentioned in the Introduction, it is left open to rigorously characterize the attacks that Twins can cover. In particular, we conjecture that Twins covers all Byzantine behaviors in a class of protocols that have “full disclosure”: each message includes a reference to its entire causal past and any source of non-determinism (such as local coin flips), and nodes act deterministically according to their causal past. It would seem that this class of protocols is fully covered by Twins since the only possible attack by Byzantine nodes is to select different subsets of messages to report to different targets. Similarly, we conjecture that Twins can cover timing violations in a class of “lock-step” synchronous protocols. Increasing coverage of Twins in the settings we explore as well as others, and providing a formal treatment of coverage remain interesting open challenges.

**Checking additional properties.** A different dimension for extension is the type of guarantees which Twins scenarios. While this paper focused squarely on safety of the core consensus protocol, the Twins approach can be extended to validate ancillary components of BFT systems. For example, DiemBFT switches to a new set of nodes by committing a special block that includes the new set of nodes and signals the reconfiguration event. It would be useful to investigate if Twins can cause a safety violation by creating an inconsistent node change (i.e., parts of the network believe in different nodes). Similarly, DiemBFT’s smart contract execution engine is re-instantiated via a similar mechanism, and can be subjected to a similar Twins-based attack.



**Extending Twins implementation.** With respect to the concrete DiemBFT Twins implementation presented in Section 5, several extensions are left for future work, including: (i) tackling more than a pair of twins; (ii) detecting liveness violations; and (iii) implementing process-level twins over TCP/IP.

---

## References

- 1 Ittai Abraham, Guy Gueta, Dahlia Malkhi, and Jean-Philippe Martin. Revisiting Fast Practical Byzantine Fault Tolerance: Thelma, Velma, and Zelma. arXiv preprint arXiv:1801.10022, 2018.
- 2 Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Maofan Yin. Sync HotStuff: Simple and Practical Synchronous State Machine Replication. In *IEEE Symposium on Security and Privacy*, 2020.
- 3 Peter Alvaro, Joshua Rosen, and Joseph M. Hellerstein. Lineage-Driven Fault Injection. In *SIGMOD International Conference on Management of Data*, 2015.
- 4 Inc. Amazon Web Services. AWS Whitepapers. <https://aws.amazon.com/whitepapers>, 2017.
- 5 Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.
- 6 Ethan Buchman. Tendermint: Byzantine Fault Tolerance in the Age of Blockchains. [https://cdn.relayto.com/media/files/LPgoW018TCeMIggJVakt\\_tendermint.pdf](https://cdn.relayto.com/media/files/LPgoW018TCeMIggJVakt_tendermint.pdf), 2016.
- 7 Ethan Buchman, Jae Kwon, and Zarko Milosevic. The Latest Gossip on BFT Consensus. arXiv preprint arXiv:1807.04938, 2018.
- 8 Vitalik Buterin and Virgil Griffith. Casper the Friendly Finality Gadget. arXiv preprint arXiv:1710.09437, 2017.
- 9 Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance. In *USENIX Symposium on Operating Systems Design and Implementation*, 1999.
- 10 Ang Chen, W Brad Moore, Hanjun Xiao, Andreas Haeberlen, Linh Thi Xuan Phan, Micah Sherr, and Wenchao Zhou. Detecting Covert Timing Channels with Time-Deterministic Replay. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 541–554, 2014.
- 11 Ang Chen, Yang Wu, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. The Good, the Bad, and the Differences: Better Network Diagnostics with Differential Provenance. In *ACM SIGCOMM Conference*, 2016.
- 12 Cosmos. Cosmos Game of Stakes, 2018. URL: <https://github.com/cosmos/game-of-stakes>.
- 13 Diem. DiemBFTBFT. <https://github.com/diem/diem>.
- 14 Patrice Godefroid, J. van Leeuwen, J. Hartmanis, G. Goos, and Pierre Wolper. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer-Verlag, 1996.
- 15 Mohammad M Jalalzai, Jianyu Niu, and Chen Feng. Fast-hotstuff: A fast and resilient hotstuff protocol. arXiv preprint arXiv:2010.11454, 2020.
- 16 Jepsen. Distributed Systems Safety Research. <https://jepsen.io>.
- 17 Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine Fault Tolerance. In *ACM SIGOPS Symposium on Operating Systems Principles*, 2007.
- 18 Leslie Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, May 1994.
- 19 Leslie Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4:382–401, 1982.
- 20 Hyojeong Lee, Jeff Seibert, Endadul Hoque, Charles Killian, and Cristina Nita-Rotaru. Turret: A platform for automated attack finding in unmodified distributed system implementations. In *2014 IEEE 34th International Conference on Distributed Computing Systems*, pages 660–669. IEEE, 2014.

- 21 Chia-Chi Lin, Virajith Jalaparti, Matthew Caesar, and Jacobus Van der Merwe. DEFINED: Deterministic Execution for Interactive Control-Plane Debugging. In *USENIX Technical Conference*, 2013.
- 22 J-P Martin and Lorenzo Alvisi. Fast Byzantine Consensus. *IEEE Transactions on Dependable and Secure Computing*, 3(3):202–215, 2006.
- 23 Atsuki Momose and Jason Paul Cruz. Force-Locking Attack on Sync Hotstuff. IACR Cryptology ePrint Archive, 2020.
- 24 Netflix. Chaos Monkey. URL: <https://netflix.github.io/chaosmonkey/>.
- 25 Filip Niksic. *Combinatorial Constructions for Effective Testing*. Doctoral thesis, Technische Universität Kaiserslautern, 2019.
- 26 Santhosh Prabhu, Kuan Yen Chou, Ali Kheradmand, Brighten Godfrey, and Matthew Caesar. Plankton: Scalable Network Configuration Verification Through Model Checking. In *USENIX Symposium on Networked Systems Design and Implementation*, 2020.
- 27 Basil Cameron Rennie and Annette Jane Dobson. On Stirling Numbers of the Second Kind. *Journal of Combinatorial Theory*, 7(2):116–121, 1969.
- 28 The Diem Team. State Machine Replication in the Libra Blockchain. <https://developers.libra.org/docs/assets/papers/libra-consensus-state-machine-replication-in-the-libra-blockchain/2019-11-08.pdf>, 2019.
- 29 Yang Wu, Mingchen Zhao, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. Diagnosing Missing Events in Distributed Systems with Negative Provenance. *ACM SIGCOMM Computer Communication Review*, 44(4):383–394, 2014.
- 30 Maysam Yabandeh, Nikola Knežević, Dejan Kostić, and Viktor Kuncak. Predicting and Preventing Inconsistencies in Deployed Distributed Systems. *ACM Transactions on Computer Systems (TOCS)*, 28(1):1–49, 2010.
- 31 Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: BFT Consensus in the Lens of Blockchain. arXiv preprint arXiv:1803.05069, 2018.
- 32 Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: BFT Consensus with Linearity and Responsiveness. In *ACM Symposium on Principles of Distributed Computing*, 2019.

## A Overview of DiemBFT

We now shift our attention to utilizing Twins for validating BFT replication in DiemBFT [13]. We discuss our implementation and evaluation of Twins for DiemBFT in Sections 5 and 6. In this section, we provide an overview of DiemBFT (for details, see the technical report [28]).

DiemBFT operates in a round-by-round manner, electing leaders in each round among the nodes to balance node participation. Rounds are slightly different from conventional “views” because it takes multiple rounds to reach a decision, but leaders are rotated in each round. The leader protocol is quite simple. A leader proposes an extension to the longest chain of requests that it knows already. Usually leaders collect batches of requests to propose, referred to as blocks, hence the DiemBFT protocol forms a chain of blocks (or a blockchain). Nodes vote for a proposed block, unless it conflicts with a longer chain that they believe may have reached consensus already. Nodes send their votes to the next leader to help the leader learn the longest safe chain. If there are three consecutive blocks in the chain,  $B_k$ ,  $B_{k+1}$ ,  $B_{k+2}$ , which are proposed in consecutive rounds,  $r_k$ ,  $r_{k+1}$ ,  $r_{k+2}$ , and each block has votes from  $2f + 1$  nodes (gathered in a data structure called the *quorum certificate*, or QC), then the protocol has reached consensus on block  $B_k$ .

If  $2f + 1$  send votes to the next leader in a timely manner, a QC is formed by the leader and it sends the next proposal. Nodes maintain a timer to track progress. When the timer expires and a node still has not received a proposal, it broadcasts a timeout vote on a Nil

block. When a node gathers enough timeout votes to form a timeout certificate, it advances its round. Every time a round fails, timeout periods are increased, allowing lagging nodes to catch up and enabling the protocol to eventually reach a decision.

As briefly alluded to in the Introduction, the trickiest part of BFT replication is to manage leader transition. DiemBFT maintains four parameters to ensure safety, and at the same time facilitate progress: (i) *current\_round*, the node's current round; (ii) *last\_voted\_round*, the last round for which the node voted; (iii) *parent\_round*, the round of the block certified by the QC attached with the block being processed; (iv) *grandparent\_round*, the parent of the block certified by the QC; and (iv) *preferred\_round*, the highest known grandparent round. Note that as a QC serves as a pointer to the previous certified block, *parent\_round* and *grandparent\_round* do not need to be explicitly tracked; these can be derived from the QC carried by a block.

**Upon Receiving a Proposal.** Upon receiving proposal for a block, a node processes the certificates it carries, and votes for the proposed block if it satisfies a simple voting rule: If a node voted for  $B_{k+2}$ , it *prefers* the sub-tree of proposals rooted at block  $B_k$  (regardless of round numbers). A node will not vote for a block  $B$  that does not belong to its preferred sub-tree rooted at  $B_k$ , unless  $B$ 's parent has votes from  $2f + 1$  nodes at a higher round than  $r_k$ . Concretely:

- **Safety Rule 1.** The *block\_round* is greater than *last\_voted\_round*.
- **Safety Rule 2.** The block's *parent\_round* is greater than or equal to *preferred\_round*.

If the node decides to vote for the proposed block, it updates its state as follows:

- **Update Rule 1.** Update *last\_voted\_round* to round of the proposed block.
- **Update Rule 2.** Update the node's *preferred\_round* to the proposed block's *grandparent\_round* if the latter is higher.
- **Update Rule 3.** Update the node's *current\_round* to the *parent\_round* + 1, if the latter is higher.

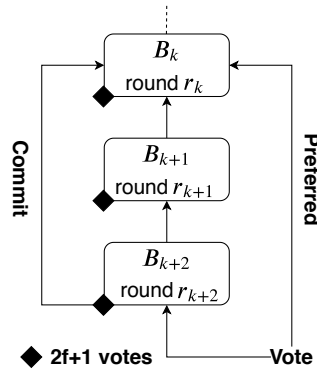
**Upon Receiving a Vote.** For every round, the nodes send their votes to the leader of the next round. When the leader receives a vote, it performs the following safety checks:

- **Safety Rule 3.** If a vote from the same node was previously received for the *same* block and round, the leader rejects the vote and generates a "duplicate vote" warning.
- **Safety Rule 4.** If a vote from the same node was previously received for a *different* block but same round, the leader rejects the vote and generates an "equivocating vote" warning.

If a vote passes both these checks, the leader considers it as valid and checks if it has enough votes to form a QC. When a QC has been formed, the leader generates a new round event, broadcasts a new block proposal and updates its state.

- **Update rule 4.** When a leader gathers enough votes to form a QC, it broadcasts a new proposal and increments *current\_round*.

*Spoiler alert:* In our evaluation in Section 6.1, we are going to deliberately modify the above rules. We will see that this enables safety violations that the Twins framework will expose.



■ **Figure 4** Consensus and preferred sub-trees in DiemBFT.

## B LibTwins Implementation of Scenario Executor and Scenario Generator

This section provides the Rust code for the two main functions of Twins, `execute_scenario` and `scenario_generator`. The code listings in Figure 5 and Figure 6 present simplified Twins interfaces, i.e., we omit Rust-specific features such as explicit typing, details of error messages returned, de-referencing, and managing variable ownership.

The scenario executor, implemented by `execute_scenario` (Figure 5), executes scenarios generated by the scenario generator. This function takes as input the number of nodes and twins, and the leaders and partitions for each round. It creates a network with the given inputs, and starts running the protocol until the nodes have emitted a given number of messages, which approximate the number of rounds for which the protocol has been run.

The `execute_scenario` function exposes a simple interface, abstracting complex underlying network and SMR configurations. To demonstrate the simplicity and flexibility of `execute_scenario`, we show how to implement a simple scenario (Figure 6) where no quorum can be formed, and therefore no block gets committed. We set up a network with 4 honest nodes ( $n_0, n_1, n_2, n_3$ ), and 1 twin ( $tw_0$ ). We split the network into two partitions  $\{n_0, tw_0, n_1\}$  and  $\{n_1, n_3\}$ . For each round  $n_0$  and  $tw_0$  (in partition 1) are leaders. We then run the protocol for enough rounds (at least 3 in DiemBFT) to get a commit on a block. In partition 1, both  $n_0$  and  $tw_0$  propose different blocks for the same rounds.  $n_1$  will only vote for one of the two proposals because the second proposal is for a round that is not greater than its `last_voted_round` (Safety rule 1, Section A). The second partition does not have enough nodes to form quorum. Consequently, no blocks are committed.

## C Detailed Safety Attack on Zyzyva

We present a summary of Zyzyva, and use Twins to reinstate a known safety attack [1] on Zyzyva [17]. We use the notation described in Section 2.

### C.1 Summary of Zyzyva

Zyzyva is an SMR protocol in the same settings as DiemBFT (partial synchrony and  $n = 3f + 1$ ). It operates in a view-by-view manner. Each view has a designated leader. Nodes vote on the leader proposal if they consider it valid (we describe the validity criteria below, which has a flaw that enables the safety attack). A commit decision on the leader proposal

```

fn execute_scenario(
  num_nodes, // number of nodes
  target_nodes, // the nodes for which to create twins
  round_partitions, // Vector of partitions for each round
  round_leaders // Vector of leaders for each round
) {
  let runtime = consensus_runtime();
  let playground = NetworkPlayground::new(runtime.handle());

  // Start nodes and twins
  let nodes = SMRNode::start_num_nodes_with_twins(
    num_nodes,
    &target_nodes,
    &playground,
    round_proposers
  );

  // Create partitions
  create_partitions(&playground, round_partitions);

  // Start running the protocol and sending messages
  block_on(async move {
    let proposals = playground
      .wait_for_messages(2, NetworkPlayground::proposals_only::<Payload>)
      .await;

    // Pull enough votes to get a commit on the first block
    let votes: Vec<VoteMsg> = playground
      .wait_for_messages(num_nodes * num_of_rounds, NetworkPlayground::votes_only
        ::<Payload>))
      .collect();

  });

  // Check that the branches are consistent at all heights
  let all_branches = vec![];

  for i in 0..nodes.len() {
    nodes[i].commit_cb_receiver.close();
    let node_commits = vec![];
    while let node_commit_id = nodes[i].commit_cb_receiver.try_next() {
      node_commits.push(node_commit_id);
    }
    all_branches.push(node_commits);
  }

  assert!(is_safe(all_branches));

  // Stop all nodes
  for each_node in nodes {
    each_node.stop();
  }
}

```

■ **Figure 5** The `execute_scenario` function which executes scenarios.

forms in either of two tracks, fast and two-phase. In the fast track, all  $n$  nodes vote for the leader proposal to commit it. In the two-phase track,  $2f + 1$  nodes form a commit-certificate ( $CC$ ), then  $2f + 1$  nodes vote for the  $CC$  to commit the proposal.

At the beginning of the view, nodes send the new leader a signed NEW-VIEW status message. The leader's first proposal carries the status of  $2f + 1$  nodes at the beginning of the view to prove the proposal validity. The (flawed) definition in Zyzyva for a valid proposal upon view change is as follows. For each sequence slot:

- **Validity Rule 1.** The leader picks among the states of  $2f + 1$  nodes, the  $CC$  from the highest view, if one exists.
- **Validity Rule 2.** Otherwise, the leader picks a proposal that has  $f + 1$  votes from the highest view, if one exists.
- **Validity Rule 3.** Finally, if none of the above exist, the leader creates a Nil proposal.

```

fn twins_no_quorum_scenario() {
  let runtime = consensus_runtime();
  let playground = NetworkPlayground::new(runtime.handle());
  let num_nodes = 4;

  // 4 honest nodes
  let n0 = 0, n1 = 1, n2 = 2, n3 = 3;
  // twin of n0
  let twin0 = node_to_twin.get(n0);
  // twin of n1
  let twin1 = node_to_twin.get(n1);

  // Index #s of nodes for which we will create twins
  let target_nodes = vec![0];

  // Specify round leaders
  let round_leaders = HashMap::new();
  for i in 1..10 {
    // Insert (n0, twin0, n3) as leaders for round i
    round_leaders.insert(i, vec![n0, twin0, n3]);
  }

  // Specify round partitions
  let round_partitions = HashMap::new();
  for r in 0..10 {
    // Insert partitions for round r
    round_partitions.insert(
      r,
      vec![
        vec![n0, twin0, n1],
        vec![n2, n3],
      ],
    );
  }

  execute_scenario(
    num_nodes,
    &target_nodes,
    &round_partitions,
    &round_leaders
  );
}

```

■ **Figure 6** Twins “No Quorum” scenario.

The flaw is to prioritize Validity Rule 1 over Validity Rule 2, which causes the leader to prefer  $CC$  even if generated in a *lower view* than  $f + 1$  votes.

## C.2 Safety Attack on Zyzzyva

The Zyzzyva flawed scenario safety demonstrated in [1] goes through a succession of three views. In the first view, a faulty leader generates conflicting proposals  $v_1, v_2$  and splits honest nodes between  $f + 1$  that vote for  $v_1$  and  $f$  that vote for  $v_2$ . The faulty leader gathers a  $CC$  on  $v_1$  but does not send it to other nodes. In the second view, a good leader adopts  $v_2$  and drives agreement in the fast track. In the third view,  $f$  faulty nodes join the  $f + 1$  honest nodes that voted for  $v_1$  in the first view. They send the leader a  $CC$  for  $v_1$ , hence the protocol proceeds with  $v_1$ , in conflict with the  $v_2$  commit. The attack on Zyzzyva needs only  $n = 4$  nodes, of which  $f = 1$  is faulty, and it is fairly easy to re-instate using the Twins framework. There are four nodes,  $(D, E, F, G)$ . To model the case that  $D$  is Byzantine, it has a twin  $D'$  initialized with different input. We drive the execution creating partitions and electing leaders at each step, according to the attack described above. We describe below the detailed attack using Twins.

**Step 1.** Initialize  $D$  and  $D'$  with different inputs  $v_1$  and  $v_2$ .

**Step 2.** During View 1:

- Create the following partitions:  $P_1 = \{\underline{D}, E, F\}$ ,  $P_2 = \{\underline{D}', G\}$ .
- Let  $D$  run as leader for one round.  $D$  proposes  $v_1$  to  $P_1$  and gathers votes from  $P_1$  creating a  $CC$ .
- Create the following partitions:  $P_1 = \{E, F\}$ ,  $P_2 = \{\underline{D}', G\}$ ,  $P_3 = \{\underline{D}\}$ .
- As a result,  $D$  does not get to share  $CC$  on  $v_1$  with  $E$  and  $F$ .
- Similarly, for one round let  $D'$  propose  $v_2$  to  $P_2$  and gather votes from  $P_2$ .

**Step 3.** Delay all messages until a new view starts. View 2:

- Create the following partitions:  $P_1 = \{D', E, \underline{G}\}$ ,  $P_2 = \{D, F\}$ .
- Run  $G$  as leader, and let it collect (NEW-VIEW) messages from  $D'$  and  $E$ . Using Validity Rule 2 (Appendix C.1),  $G$  decides to propose for  $v_2$ .
- Remove all partitions, i.e.,  $P = \{D, D', E, F, \underline{G}\}$ .
- $G$  proposes  $v_2$ , and collects votes from everyone. This leads to a commit of  $v_2$ .

**Step 4.** Delay all further messages until new view starts. View 3:

- Create the following partitions:  $P_1 = \{D, \underline{E}, F\}$ ,  $P_2 = \{D', G\}$ .
- Run  $E$  as leader, and collect (NEW-VIEW) messages from  $D$  and  $F$ . Note that  $D$  sends the  $CC$  on  $v_1$  (from view 1) to  $E$ . Using Validity Rule 1 (Appendix C.1),  $E$  decides to propose  $v_1$ .
- $E$  proposes  $v_1$  to  $P_1$ , and gathers votes from  $D$ ,  $E$  and  $F$  (who empty their local logs, undoing  $v_2$ ). This leads both  $E$  and  $F$  to commit  $v_1$ , a safety violation.

## D Detailed Liveness Attack on FaB

We present a summary of FaB, and use Twins to reinstate a known liveness attack on FaB [1]. We use the notation described in Section 2.

### D.1 Summary of FaB

FaB is a single-shot consensus protocol for the partial synchrony setting with  $n = 3f + 1$ .<sup>7</sup>

A precursor to Zyzzyva, FaB is a view-based protocol with an optimistic fast track. A leader drives a decision in the fast track if all nodes vote for it, and in the two-phase track if  $2f + 1$  nodes vote for a  $(2f + 1)$  commit-certificate ( $CC$ ). When a new leader is elected, it picks a valid proposal that does not conflict with neither  $f + 1$  votes nor a  $CC$  in the previous view.

<sup>7</sup> FaB is actually designed for a *parameterized* model with  $n = 3f + 2t + 1$ , with safety guaranteed against  $f$  Byzantine failures and fast track guaranteed against  $t$ . For brevity and uniformity, we ignore  $t$  here and set  $t = 0$ .

## D.2 Liveness Attack on FaB

The (flawed) selection criterion above leads an execution in the following scenario to become stuck. A faulty leader equivocates and proposes  $v_1, v_2$  to  $2f + 1$  and  $f$  honest nodes, respectively. In transitioning to the next view, there is a commit-certificate for  $v_1$  and  $f + 1$  votes for  $v_1$  (including an equivocation by one faulty), hence neither is safe, and the new leader is stuck. The attack on FaB needs only  $n = 4$  nodes, of which  $f = 1$  is faulty, and it can be easily re-instated using Twins. There are four nodes,  $(A, B, C, D)$  with  $D$  as a Byzantine node, for which we create a twin  $D'$  initialized with different input. We describe below the attack using Twins.

**Step 1.** Initialize  $D$  and  $D'$  with different inputs  $v_1$  and  $v_2$ .

**Step 2.** During View 1:

- Create the following partitions:  $P_1 = \{A, B, \underline{D}\}$ ,  $P_2 = \{C, \underline{D}'\}$
- Run  $D$  as leader for one round.  $D$  proposes  $v_1$  to  $P_1$  which decides to vote on  $v_1$ .
- Insert the following rule in  $P_1$ :  $(B, D) \rightarrow A$ . That is, the only messages allowed are those from  $B$  and  $D$ , to  $A$ .
- $D$ ,  $A$  and  $B$  send their votes which only reach  $A$ . Thus, only  $A$  produces a  $CC$  for  $v_1$ .
- Meanwhile, the leader  $D'$  proposes  $v_2$  to  $P_2$ .

**Step 3.** Delay all further messages until new view starts. Create the partitions:  $\{\underline{A}, C, \underline{D}'\}$ ,  $\{B, D\}$ . Let the new leader  $A$  collect NEW-VIEW status messages from  $P_1$ . These status messages block  $A$  from proposing both  $v_1$  and  $v_2$  due to the FaB proposal validity rule. The rule states that a proposal is valid if it does not conflict with neither  $f + 1$  votes nor a  $CC$  in the previous view, which is not the case for  $v_1$  (has a  $CC$ ) and  $v_2$  (has  $f + 1$  votes) as described below:

- From  $A$ , the NEW-VIEW message contains the value  $v_1$ , and a  $CC$  for it.
- From  $C$ , the NEW-VIEW message contains the value  $v_2$ , and no  $CC$ .
- From  $D'$ , the NEW-VIEW message contains the value  $v_2$ , and no  $CC$ .

## E Detailed Liveness Attack on Sync HotStuff

We present a summary of Sync HotStuff, and use Twins to reinstate the force-locking attack [23] on a preliminary version of Sync HotStuff (which was fixed in an updated version). We use the notation described in Section 2.

### E.1 Summary of Sync HotStuff

The preliminary version of Sync HotStuff [2] is an SMR solution in the synchronous model with  $n = 2f + 1$  parties.<sup>8</sup>

In synchronous protocols like Sync HotStuff, nodes execute the protocol in terms of  $\Delta$ , which is the known bound assumed on maximal network transmission delay. Sync HotStuff operates in a view-by-view regime – in each view there is a designated leader which proposes

<sup>8</sup> The description here covers the first of three variants in that paper; two other variants are designed for slightly different synchrony assumptions, but the attacks on them are similarly covered by the Twins approach.



values to nodes. If a node accepts the proposed value, it broadcasts its vote. A node creates a commit certificate ( $CC$ ) for a proposed value if it receives  $f + 1$  votes on it. Nodes track the highest  $CC$ , and only vote on a proposed value if it: (i) extends the highest  $CC$  known to the node, and (ii) does not equivocate another value proposed for the same height.

A node creates and broadcasts a *blame* against a leader: (i) if the leader does not propose a value for  $3\Delta$ , or (ii) the leader proposes an equivocating value. If a node observes  $f + 1$  blames against the leader in the current view, it broadcasts the  $f + 1$  blames, then waits  $\Delta$  (to allow the blames to reach all honest nodes), and moves to the new view. In the new view, it immediately sends the new leader the highest  $CC$  it knows of.

After a view change, the new leader waits for  $\Delta$  to receive node status messages (carrying the highest  $CC$  known to them). The leader then proposes a value that extends the highest  $CC$  from among the received status messages. Nodes proceed in the new view as previously described.

## E.2 Implementing Synchrony Attacks in Twins

Due to the synchronous settings and the nature of the attack which heavily leverages synchrony assumptions, in this case a Twins scheduler must control message delivery timing. More precisely, rather than only specifying whether a message is delivered to a party or dropped, attacks on synchronous protocols require the Twins scheduler to deliver messages to specific parties at specified times. While this is captured by the Twins approach, our current implementation (Section 5) does not support this feature (this will be implemented in future Twins extensions).

Generally, we expect that the granularity of the scheduler timing can be fairly coarse. In particular, there is a known parameter  $\Delta$ , the bound presumed by the algorithm on message transmission delays and hard-coded into it. Indeed, the force-locking attack needs to deliver messages at  $0.5\Delta$  increments, e.g., at times  $0, 0.5\Delta, \Delta, 1.5\Delta, 2.0\Delta, \dots$ . Therefore, a Twins network emulator could operate in discrete lock-step at  $0.5\Delta$  increments. With this capability in place, the force-locking attack can be re-instated in the Twins approach as described below.

## E.3 Safety Attack on Sync HotStuff

We now rebuild the force-locking attack on the preliminary version of Sync HotStuff using Twins. The crux of the attack is for a faulty leader to generate a last-minute proposal that reaches only half of the honest nodes. The other half trigger a view change, and now the system becomes split. The first half continues to commit the first leader proposal with “help” from Byzantine nodes. The second half starts a new view and fork the chain. This attack can be reinstated with Twins using 5 nodes ( $A, B, C, D, E$ ), of which ( $A, B$ ) are faulty and have twins ( $A', B'$ ).

**Notation.** We extend the notation described in Section 2 to capture message transmission in the synchronous setting as follows:  $S_t \xrightarrow{v} S'_{t'}$  denotes the transmission of a value  $v$  from a set of nodes  $S$  that generate the value at time  $t$ , to a set of nodes  $S'$  that receive the value at time  $t'$ . If a value is broadcast, we use the  $\star$  symbol instead of a set: For example,  $S_t \xrightarrow{v} \star$  means that  $S$  broadcasts a value  $v$  at time  $t$ . Additionally, to highlight the “send” or “receive” action on a value, we use bold text on the left or right side of the arrow, respectively. For example,  $\mathbf{S}_t \xrightarrow{v} S'$  means that  $S$  sends  $v$  to  $S'$  (message arrival time is not known).

## 7:26 Twins: BFT Systems Made Robust

To reinstate this attack with Twins, we deploy 5 nodes  $(A, B, C, D, E)$ , of which  $(A, B)$  are faulty and have twins  $(A', B')$ . Here,  $n = 5$ ,  $f = 2$ , and quorum size is 3 (since synchronous BFT protocols tolerate  $f$  Byzantine nodes for  $n = 2f + 1$ ). We describe below the detailed attack using Twins.

### At time $1.5\Delta$ :

- $A$  is the leader, and broadcasts a proposal with  $delay = \Delta$  for the value  $v_1$  which extends  $v_0$ .

$$(A)_{1.5\Delta} \xrightarrow{\text{propose}(v_1)} \star$$

### At time $2.5\Delta$ :

- $C$  receives  $V_1$ , and broadcasts its vote.

$$(A)_{1.5\Delta} \xrightarrow{\text{propose}(v_1)} (C)_{2.5\Delta}$$

$$(C)_{2.5\Delta} \xrightarrow{\text{vote}(v_1)} \star$$

### At time $3\Delta$ :

- $D$  blames  $A$  since it did not receive a proposal from  $A$  within  $3\Delta$ . Twins  $(A', B')$  also did not receive a proposal from  $A$ , hence they also blame with  $A$ .  $(D, A', B')$  broadcast their blames with  $delay = 0$ , receive  $f + 1$  blames from each other, and start waiting for  $\Delta$ .

$$(D, A', B')_{3\Delta} \xrightarrow{\text{blame}(A)} \star$$

$$(D, A', B')_{3\Delta} \xrightarrow{\text{blame}(A)} (D, A', B')_{3\Delta}$$

### At time $3.5\Delta$ :

- $D$  receives  $C$ 's vote on  $v_1$ , but it cannot create a  $CC$  on  $v_1$  since it has less than  $f + 1$  votes.

$$(C)_{2.5\Delta} \xrightarrow{\text{vote}(v_1)} (D)_{3.5\Delta}$$

- $(A, B)$  broadcast their votes on  $v_1$ , which arrive at  $C$  with delay 0. As a result,  $C$  gathers  $f + 1$  votes on  $v_1$  and creates  $CC(v_1)$ .

$$(A, B)_{3.5\Delta} \xrightarrow{\text{vote}(v_1)} \star$$

$$(A, B)_{3.5\Delta} \xrightarrow{\text{vote}(v_1)} (C)_{3.5\Delta}$$

### At time $4\Delta$ :

- $C$  receives  $f + 1$  blame messages from  $(D, A', B')$ , broadcasts all blame messages, and starts waiting for  $\Delta$ .

$$(D, A', B')_{3\Delta} \xrightarrow{\text{blame}(A)} (C)_{4\Delta}$$

$$(C)_{4\Delta} \xrightarrow{\text{blame}(A)} \star$$

- $D$  has waited for  $\Delta$  since it quit the old view  $w$  with leader  $A$ , so it starts the next view  $w + 1$  and sends its highest commit certificate  $CC(V_0)$  along with  $f + 1$  blames on  $A$  to the next leader  $B$ , with  $delay = 0$ .

$$(D)_{4\Delta} \xrightarrow{CC(v_0), \text{blame}(A)} (B)_{4\Delta}$$

- The new leader  $B$  receives  $CC(v_0)$  from  $D$  and  $f + 1$  blames on  $A$ , and broadcasts a proposal for value  $v_1'$  extending  $V_0$ . Note that  $B$  does not know about  $CC(v_1)$ .

$$(D)_{4\Delta} \xrightarrow{CC(v_0), \text{blame}(A)} (B)_{4\Delta}$$

$$(B)_{4\Delta} \xrightarrow{\text{propose}(v_1')} \star$$

- $D$  receives the proposal  $v_1'$  from  $B$ , and broadcasts its vote with delay  $\Delta$ , then it sets its commit timer to  $2\Delta$  and starts counting down.

$$(B)_{4\Delta} \xrightarrow{\text{propose}(v_1')} (D)_{4\Delta}$$

$$(D)_{4\Delta} \xrightarrow{\text{vote}(v_1')} \star$$

At time  $4.5\Delta$ :

- $D$  receives votes on  $v_1$  from  $(A, B)$ ; as it has now gathered  $f + 1$  votes on  $v_1$  it creates  $CC(v_1)$ . However, this certificate is too late, as we will see in the following steps.

$$(A, B)_{3.5\Delta} \xrightarrow{\text{vote}(v_1)} (D)_{4.5\Delta}$$

At time  $5\Delta$ :

- $C$  has waited for  $\Delta$  since it quit the old view with leader  $A$ , so it starts the next view  $w + 1$  and sends its highest certificate  $CC(v_1)$  to the new leader  $B$ .

$$(C)_{5\Delta} \xrightarrow{CC(v_1)} (B)$$

- $C$  receives  $D$ 's vote on  $v_1'$  but does not vote since  $v_1'$  (which extends  $CC(v_0)$ ) does not extend its highest certificate  $CC(V_1)$ .

$$(D)_{4\Delta} \xrightarrow{\text{vote}(v_1')} (C)_{5\Delta}$$

At time  $6\Delta$ :

- $D$  commits  $v_1'$  since it finished waiting for  $2\Delta$  and observed no equivocation or blame in the view  $w + 1$ . However,  $D$ 's highest certificate is  $CC(v_1)$  (see time  $4.5\Delta$ ).
- Now if the current leader  $B$  goes offline, this will result in a view change to view  $w + 2$  and the new leader will extend the blockchain from the highest certificate from the previous view,  $CC(v_1)$ . But  $D$  has committed  $v_1'$  conflicting with  $v_1$ , hence safety is violated.

## F Attack on Fast-HotStuff

We present a safety attack against Fast-HotStuff [15] and express it using Twins.

### F.1 Summary of Fast-HotStuff

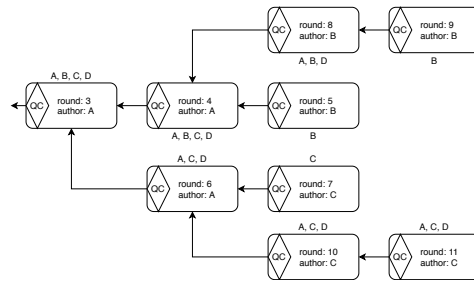
Fast-HotStuff is essentially HotStuff [32] with a 2-phase commit rule. In the happy-path, if the leader of round  $n$  is successful, the leader of round  $n + 1$  performs the same protocol as HotStuff, namely, it collects a QC from previous round and embeds it in the  $n + 1$  proposal. In the unhappy-path, if the leader of round  $n$  is unsuccessful, the protocol for leader  $n + x + 1$  ( $x > 0$ ) provides a proof in the  $n + x + 1$  proposal that it is using the highest QC from  $2f + 1$  validators. This proof incurs quadratic communication complexity. Moreover, Fast-HotStuff claims it does not require consecutive rounds in order to commit.

The benefits of Fast-HotStuff are twofold. It provides a fast 2-phase track for HotStuff whenever the leader is successful in obtaining a QC for the previous round (happy-path). Fast-HotStuff is also faster both in phases (2 phases instead of 3) and in getting to a scenario that guarantees progress, namely, it requires 3 consecutive honest leaders (instead of 4). Requiring a leader proof for the unhappy-path prevents a proposal that conflicts with an uncommitted and unlocked tail of a chain that already has a QC. Thus, dishonest leaders cannot intentionally slow down progress by overriding the latest tail.

Fast-HotStuff is however flawed as explained in Appendix F.2.

### F.2 Safety Attack on Fast-HotStuff

Figure 7 illustrates the safety attack against Fast-HotStuff that we implement using Twins. There are four nodes  $(A, B, C, D)$  all of which are honest – the safety attack can be executed leveraging only network partitions. Blocks are represented by rectangles (which are annotated



■ **Figure 7** Example of safety attack on Fast-HotStuff.

with the nodes that receive the block). Block proposers are indicated as “authors”. Diamonds refers to QCs (which are embedded into blocks). The arrows indicate the block that a QC refers to.

We execute the safety attack in 11 rounds starting at round 3 (rounds 2 carries QC for the genesis block).

**Round 3:** Initially there are no partitions, i.e.,  $P = \{\underline{A}, B, C, D\}$ .

- $A$  proposes a block. Nodes send their votes on this proposal to the leader of the next round, node  $A$ .

**Round 4:**

- $A$  gathers votes from the previous round, forms a QC, and includes the QC in a new block proposal. Nodes send their votes on the new proposal to the leader of the next round, node  $B$ .

**Round 5:** Set node  $B$  as leader, i.e.,  $P = \{A, \underline{B}, C, D\}$ .

- $B$  gathers votes from the previous round, forms a QC, and includes the QC in a new block proposal.
- Create the following partitions:  $P_1 = \{A, C, D\}$  and  $P_2 = \{\underline{B}\}$ .
- The partitions prevent  $B$  from broadcasting the new block (and the newly formed QC it embeds).  $B$  is thus the only node knowing the QC certifying the block of round 4.
- Nodes of  $P_1$  time out, and send a NEW-VIEW message to the leader of the next round (node  $A$ ) containing their highest known QC.

**Round 6:** Set node  $A$  as leader, i.e.,  $P_1 = \{\underline{A}, C, D\}$  and  $P_2 = \{B\}$ .

- $A$  selects the highest QC from the NEW-VIEW messages (i.e., the QC certifying the block of round 3), and embeds it in a new block proposal. All nodes of  $P_1$  vote on this proposal and send their votes to the leader of the next round (node  $C$ ).

**Round 7:** Set node  $C$  as leader, i.e.,  $P_1 = \{A, \underline{C}, D\}$  and  $P_2 = \{B\}$ .

- $C$  gathers votes from the previous round, forms a QC, and includes the QC in a new block proposal.
- Create partitions:  $P_1 = \{A, B, D\}$  and  $P_2 = \{\underline{C}\}$ .
- These partitions prevent  $C$  from broadcasting the new block (and the newly formed QC it embeds).  $C$  is thus the only node knowing the QC certifying the block of round 6.
- Nodes of  $P_1$  time out and send a NEW-VIEW message to the leader of the next round (node  $B$ ) containing their highest known QC.

**Round 8:** Set node  $B$  as leader, i.e.,  $P_1 = \{A, \underline{B}, D\}$  and  $P_2 = \{C\}$ .

- $B$  selects the highest QC from the NEW-VIEW messages (i.e., the QC certifying the block of round 4, presented by  $B$ ), and embeds it in a new block proposal. All nodes vote on this proposal and send their votes to the leader of the next round (node  $B$ ).

**Round 9:**

- $B$  gathers all votes from the previous round, forms a QC, and includes the QC in a new block proposal.
- Create partitions  $P_1 = \{A, C, D\}$  and  $P_2 = \{B\}$ .
- The partitions prevent  $B$  from broadcasting its newly block (and the newly formed QC it embeds).  $B$  is thus the only node knowing the QC certifying the block of round 8 and committing the block at round 4.
- Nodes of  $P_1$  time out and send a NEW-VIEW message to the leader of the next round (node  $C$ ) containing their highest known QC.

**Round 10:** Set node  $C$  as leader, i.e.,  $P_1 = \{A, \underline{C}, D\}$  and  $P_2 = \{B\}$ .

- $C$  selects the highest QC from the NEW-VIEW messages from the previous round (the QC certifying the block of round 6, presented by  $C$ ), and embeds it in its new block proposal. The highest QC in the NEW-VIEW messages.
- All nodes of  $P_1$  vote on this proposal and send their votes to the leader of the next round (node  $C$ ).

**Round 11:** Set node  $C$  as leader, i.e.,  $P_1 = \{A, \underline{C}, D\}$  and  $P_2 = \{B\}$ .

- $C$  assembles votes from the previous round into a QC certifying the block of round 10, thus committing the block of round 6.

The safety violation appears at round 11 when node  $C$  commits the block of round 6 while node  $B$  previously committed the block of round 4: both blocks have the same height and fork from the block of round 3.

### F.3 Implementation of the Attack

We implemented a Python simulator of Fast-HotStuff using the discrete event simulator *simpy*. We demonstrate the safety violation by running a manually-crafted scenario in the simulator. We are open sourcing our Fast-HotStuff simulator as well as our Twins scenario used for the attack<sup>9</sup>.

---

<sup>9</sup> <https://github.com/asonnino/twins-simulator>



# Near-Optimal Dispersion on Arbitrary Anonymous Graphs

Ajay D. Kshemkalyani  

University of Illinois at Chicago, IL, USA

Gokarna Sharma  

Kent State University, OH, USA

---

## Abstract

Given an undirected, anonymous, port-labeled graph of  $n$  memory-less nodes,  $m$  edges, and degree  $\Delta$ , we consider the problem of dispersing  $k \leq n$  robots (or tokens) positioned initially arbitrarily on one or more nodes of the graph to exactly  $k$  different nodes of the graph, one on each node. The objective is to simultaneously minimize time to achieve dispersion and memory requirement at each robot. If all  $k$  robots are positioned initially on a single node, depth first search (DFS) traversal solves this problem in  $O(\min\{m, k\Delta\})$  time with  $\Theta(\log(k + \Delta))$  bits at each robot. However, if robots are positioned initially on multiple nodes, the best previously known algorithm solves this problem in  $O(\min\{m, k\Delta\} \cdot \log \ell)$  time storing  $\Theta(\log(k + \Delta))$  bits at each robot, where  $\ell \leq k/2$  is the number of multiplicity nodes in the initial configuration. In this paper, we present a novel multi-source DFS traversal algorithm solving this problem in  $O(\min\{m, k\Delta\})$  time with  $\Theta(\log(k + \Delta))$  bits at each robot, improving the time bound of the best previously known algorithm by  $O(\log \ell)$  and matching asymptotically the single-source DFS traversal bounds. This is the first algorithm for dispersion that is optimal in both time and memory in arbitrary anonymous graphs of constant degree,  $\Delta = O(1)$ . Furthermore, the result holds in both synchronous and asynchronous settings.

**2012 ACM Subject Classification** Mathematics of computing  $\rightarrow$  Graph algorithms; Computing methodologies  $\rightarrow$  Distributed algorithms; Computer systems organization  $\rightarrow$  Robotics

**Keywords and phrases** Distributed algorithms, Multi-agent systems, Mobile robots, Local communication, Dispersion, Exploration, Time and memory complexity

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2021.8

## 1 Introduction

Given an undirected, anonymous, port-labeled graph of  $n$  memory-less nodes,  $m$  edges, and (maximum) degree  $\Delta$ , we consider the problem of dispersing  $k \leq n$  robots (or tokens) positioned initially arbitrarily on one or more nodes of the graph to exactly  $k$  different nodes of the graph, one on each node (which we call the DISPERSION problem). This problem has many practical applications, for example, in relocating self-driven electric cars (robots) to recharge stations (nodes), assuming that the cars have smart devices to communicate with each other to find a free/empty charging station [1, 18]. This problem is also important because it has the flavor of many other well-studied robot coordination problems, such as exploration, scattering, load balancing, covering, and self-deployment [1, 18, 22].

One of the key aspects of mobile-robot research is to understand how to use the resource-limited robots to accomplish some large task in a distributed manner [12, 13]. In this paper, we study trade-off between time and memory complexities to solve DISPERSION on arbitrary anonymous graphs. Time complexity is measured as the time duration to achieve dispersion and memory complexity is measured as the number of bits stored in persistent memory at each robot. The literature typically traded memory (or time) to obtain better time (or memory) bounds (for example, compare memory and time bounds of the two algorithms from [18] given in Table 1).



© Ajay D. Kshemkalyani and Gokarna Sharma;  
licensed under Creative Commons License CC-BY 4.0

25th International Conference on Principles of Distributed Systems (OPODIS 2021).

Editors: Quentin Bramas, Vincent Gramoli, and Alessia Milani; Article No. 8; pp. 8:1–8:19

Leibniz International Proceedings in Informatics



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

■ **Table 1** Algorithms solving DISPERSION for  $k \leq n$  robots on undirected, anonymous, port-labeled graphs of  $n$  memory-less nodes,  $m$  edges, and (maximum) degree  $\Delta$ . <sup>†</sup>[19] assumes  $m, k$ , and  $\Delta$  are known to the algorithm a priori.  $\ell \leq k/2$  is the number of multiplicity nodes in the initial configuration; DISPERSION is already solved if there is no multiplicity node.

Algorithm	Memory/robot (in bits)	Time (in rounds/epochs)	Single-source/ Multi-source	Setting
Lower bound	$\Omega(\log(k + \Delta))$	$\Omega(k)$	any	Asynchronous
DFS	$\Theta(\log(k + \Delta))$	$O(\min\{m, k\Delta\})$	Single-source	Asynchronous
[18]	$O(k \log \Delta)$	$O(\min\{m, k\Delta\})$	Multi-source	Asynchronous
[18]	$\Theta(\log(k + \Delta))$	$O(\min\{m, k\Delta\} \cdot \ell)$	Multi-source	Asynchronous
[19] <sup>†</sup>	$O(\log n)$	$O(\min\{m, k\Delta\} \cdot \log \ell)$ <sup>†</sup>	Multi-source	Synchronous
[31]	$\Theta(\log(k + \Delta))$	$O(\min\{m, k\Delta\} \cdot \log \ell)$	Multi-source	Synchronous
<b>Th. 1</b>	$\Theta(\log(k + \Delta))$	$O(\min\{m, k\Delta\})$	Multi-source	Synchronous
<b>Th. 2</b>	$\Theta(\log(k + \Delta))$	$O(\min\{m, k\Delta\})$	Multi-source	Asynchronous

Recent studies [19, 31] focused on minimizing time and memory complexities simultaneously. More precisely, they tried to answer the following question: *Can the time bound of  $O(\min\{m, k\Delta\})$  be obtained keeping memory optimal  $\Theta(\log(k + \Delta))$  bits at each robot?* This question can be easily answered in the single-source case of all  $k \leq n$  robots initially co-located on a node. The challenge is how to answer it in the multi-source case of the robots initially on two or more nodes of the graph. For the multi-source case, the algorithms in [19, 31] were successful in keeping memory bound optimal as in [18] and reduce time bound to  $O(\min\{m, k\Delta\} \cdot \log \ell)$ , an improvement of  $\ell/\log \ell$  factor compared to the  $O(\min\{m, k\Delta\} \cdot \ell)$  time bound of [18], where  $\ell \leq k/2$  is the number of multiplicity nodes in the initial configuration.

In this paper, we present a new algorithm for DISPERSION that settles the question completely, i.e., it obtains the time bound of  $O(\min\{m, k\Delta\})$  keeping memory optimal  $\Theta(\log(k + \Delta))$  bits at each robot, first such result for the multi-source case. The time bound is an improvement of  $O(\log \ell)$  factor compared to the best previously known algorithms [19, 31]. Furthermore, the time and memory bounds match the respective bounds for the single-source case. Thus, the proposed algorithm is the first for DISPERSION that is simultaneously optimal for arbitrary anonymous graphs of constant degree  $\Delta = O(1)$ .

**Overview of the Model and Results.** We consider  $k \leq n$  robots operating on an undirected, anonymous (no node IDs), port-labeled graph  $G$  of  $n$  memory-less nodes,  $m$  edges, and degree  $\Delta$ . The ports (leading to incident edges) at each node have unique labels from  $[0, \delta - 1]$ , where  $\delta$  is the degree of that node. ( $\Delta$  is the maximum over  $\delta$ 's of all  $n$  nodes.) The robots have unique IDs in the range  $[1, k]$ . In contrast to graph nodes which are memory-less, the robots have memory to store information (otherwise the problem becomes unsolvable). Finally, at any time, the robots co-located at the same node of  $G$  can communicate and exchange information, if needed, but they cannot communicate and exchange when located on different nodes. We call an initial configuration *single-source* if all  $k$  robots are initially positioned on a single node of  $G$ , otherwise we call it *multi-source*. Even in the multi-source initial configurations, the robots can only be on  $1 < k' < k$  nodes, since for the case of  $k' = k$ , the initial configuration is already a configuration that solves DISPERSION. In this paper, we establish the following theorem in the *synchronous* setting where all robots are activated in a round, they perform their operations simultaneously in synchronized rounds, and hence the time (of the algorithm) is measured in rounds (or steps).



► **Theorem 1.** *Given any initial configuration of  $k \leq n$  mobile robots on the nodes of an undirected, anonymous, port-labeled graph  $G$  of  $n$  memory-less nodes,  $m$  edges, and degree  $\Delta$ , dispersion can be solved deterministically in  $O(\min\{m, k\Delta\})$  rounds in the synchronous setting storing  $O(\log(k + \Delta))$  bits at each robot.*

Theorem 1 improves the time bound  $O(\min\{m, k\Delta\} \cdot \log \ell)$  of the best previously known algorithms [19, 31] by a factor of  $O(\log \ell)$  keeping the memory optimal, where  $\ell$  is the number of nodes in the initial configuration with at least two robots co-located on them. Interestingly, both time and memory bounds of Theorem 1 match asymptotically the  $O(\min\{m, k\Delta\})$  time and  $O(\log(k + \Delta))$  memory bounds for the single-source case, which is inherent for any DFS traversal based algorithm for DISPERSION. Finally, for constant-degree arbitrary anonymous graphs, i.e.,  $\Delta = O(1)$ , our algorithm is asymptotically optimal w.r.t. both time and memory, first such result for DISPERSION (Table 1).

Furthermore, we extend Theorem 1 to the *asynchronous* setting where robots become active and perform their operations in arbitrary duration, keeping the same time and memory bounds. Here we measure time in epochs (instead of rounds) – an epoch represents the time interval in which each robot becomes active at least once.

► **Theorem 2.** *Given the setting as in Theorem 1, dispersion can be solved deterministically in  $O(\min\{m, k\Delta\})$  epochs in the asynchronous setting storing  $O(\log(k + \Delta))$  bits per robot.*

**Challenges.** The single-source DISPERSION can be solved in  $\min\{4m - 2n + 2, 4k\Delta\}$  rounds in any anonymous graph  $G$  having  $n$  memory-less nodes using the well-known DFS traversal [6] storing  $O(\log(k + \Delta))$  bits at each robot. The  $k$ -source DISPERSION finishes in a single round, since  $k$  robots are already on  $k$  different nodes solving DISPERSION. Therefore, the challenging case is  $k'$ -source DISPERSION with  $1 < k' < k$ .

The early papers obtained better bounds on either time or memory, trading one for another. The first algorithm of [18] obtained  $O(\min\{m, k\Delta\})$  time bound with memory  $O(k \log \Delta)$  bits at each robot. The second algorithm of [18] kept memory optimal  $O(\log(k + \Delta))$  bits at each robot and established time  $O(\min\{m, k\Delta\} \cdot \ell)$ , where  $\ell \leq k' < k$  is the number of multiplicity nodes in the initial configuration. Their algorithm starts  $\ell$  different single-source DFS traversals in parallel from  $\ell$  sources with multiple robots on them. Each DFS traversal is given a unique ID, which is the smallest robot ID present on that source. Each DFS traversal leaves a robot on each new node it visits. If no DFS traversals meet, then  $k$  robots are on  $k$  different nodes and DISPERSION is solved in time and memory bounds akin to the single-source DFS bounds. In case of two (or more) DFS traversals meet, the higher ID DFS traversal subsumes the lower ID DFS traversal. The problem here is that if the lower ID DFS traversal meets the higher ID DFS traversal, in the subsumption process, the higher ID DFS traversal may again visit all the nodes that the lower ID DFS traversal already visited. Therefore, in the worst-case, the time becomes the multiplication of  $O(\min\{m, k\Delta\})$  rounds for the single-source DFS traversal times  $\ell$  parallel traversals, i.e.,  $O(\min\{m, k\Delta\} \cdot \ell)$  rounds.

Recent studies [19, 31] reduced the  $O(\ell)$  factor in the time bound to  $O(\log \ell)$ . Providing  $m, k$ , and  $\Delta$  parameters to the algorithm beforehand, Kshemkalyani et al. [19] run  $\ell$ -source DFS traversals in passes of interval  $O(\min\{m, k\Delta\})$  rounds. After each pass, they guaranteed that the  $\ell$ -source DFS traversal reduces to  $\ell/2$ -source DFS traversal. Therefore, in total  $\lceil \log \ell \rceil$  passes, the  $\ell$ -source DFS traversal reduces to a single-source DFS traversal, which then finishes in additional  $O(\min\{m, k\Delta\})$  rounds, giving in the worst-case,  $O(\min\{m, k\Delta\} \cdot \log \ell)$  rounds time bound. The memory requirement is  $O(\log n)$  bits at each robot, due to the memory to store  $m \leq n^2$  which dominates the memory to store  $k \leq n$  and  $\Delta < n$ . Recently,

Shintaku et al. [31] established the same time bound as in [19] avoiding the requirement for the algorithm to know  $m, k, \Delta$  beforehand. Moreover, they improved the memory bound  $O(\log n)$  bits in [19] to optimal  $\Theta(\log(k + \Delta))$  bits at each robot.

Observing the techniques of [19, 31], the algorithms developed there subsume different DFS traversals pairwise which helps in improving the sequential subsumption of the different DFS traversals in the algorithm of [18]. The implication of the pairwise subsumption is only a  $O(\log \ell)$  factor more cost is needed to subsume all  $\ell$  parallel DFS traversals to obtain a single DFS traversal. This  $O(\log \ell)$  factor is significantly better compared to the  $O(\ell)$  factor obtained due to the sequential subsumption.

Despite these benefits, the pairwise subsumption is not matching the single-source DFS traversal time bound and, more importantly, it is not clear whether the  $O(\log \ell)$  factor arising in the pairwise subsumption technique in [19, 31] can be removed from the time bound. Therefore, a new set of ideas are needed, which we develop in this paper and they constitute our main contribution.

**Techniques.** We use parallel multi-source DFS traversals as in [19, 31] but devise a novel subsumption technique, leading to  $O(\min\{m, k\Delta\})$  time with  $O(\log(k + \Delta))$  bits at each robot, removing the  $O(\log \ell)$  factor from the time bound of the best previously known algorithms [19, 31] and matching the time and memory bounds for single-source DFS traversal. Each DFS traversal constructs a *DFS tree*. Our technique executes subsumption on the two DFS traversals that meet based on the size of the DFS traversal measured as the number of settled robots with the same DFS tree ID. In fact, the larger size DFS traversal subsumes the smaller size DFS traversal. The subsumed DFS traversal is collapsed to a single node, collecting all the robots on that traversal at that node, and those robots are given to the subsuming DFS traversal allowing it to extend its DFS traversal. The benefit is two-fold: (i) the size of the subsumed traversal is smaller than the size of the subsuming traversal and hence the collapse and merge of the subsumed traversal to the subsuming one can be done in time proportional to the size of the subsumed traversal, and (ii) it avoids the need of revisiting the nodes of the subsumed traversal more than once, a crucial aspect in removing the  $O(\log \ell)$  factor from the time bound. Furthermore, one traversal always remains subsuming throughout the execution of the algorithm.

This is in contrast to the technique used in the best previously known algorithms [19, 31] that uses IDs of the DFS traversals (larger ID DFS traversal subsumes smaller ID DFS traversal). The drawback of the subsumption based on DFS ID is that the algorithm cannot limit the repeating traversal of the already built DFS tree, adding a  $\Theta(\log \ell)$  factor in the subsumption process, and hence leading to a  $O(\min\{m, k\Delta\} \cdot \log \ell)$  time bound.

We particularly tackle two major challenges: (i) how to execute the size-based subsumption, and (ii) what to do when more than two DFS traversals meet at different nodes forming a transitive chain or more generally, what we define as a *meeting graph* (Definition 4). The first challenge is due to the fact that the exact size of the DFS traversal is only known by its *head node* which is either the current node that has all not-yet-settled robots (if any,) belonging to that DFS traversal or else the node on which last robot belonging to that DFS traversal has settled. Therefore, it requires for the meeting traversal to traverse the met DFS tree to reach its head node to find its size. Our technique of collapsing the subsumed traversal successfully fulfills this requirement in time proportional to the size of the smaller size DFS traversal.

The second challenge is due to the fact that if not synchronized carefully, different DFS traversals in the transitive chain or meeting graph might run into a deadlock situation. We devise a technique that partitions the DFS traversals in the meeting graph such that in each partition, one DFS traversal subsumes the others without introducing any deadlock and in time proportional to the size of the DFS traversals that were subsumed and collapsed.

Through these techniques, we finally show that one DFS traversal (among those that meet in the meeting graph) always grows bigger and the total cost remains proportional to the total size of the DFS traversals that are subsumed by the DFS traversal, giving our claimed time bound. Interestingly, the process is executed keeping the memory at an (asymptotically) optimal number of bits per robot.

**Related Work.** Augustine and Moses Jr. [1] proved a memory lower bound of  $\Omega(\log n)$  bits at each robot and a time lower bound of  $\Omega(D)$  ( $\Omega(n)$  in arbitrary graphs) for any deterministic algorithm for DISPERSION on graphs. They then provided deterministic algorithms using  $O(\log n)$  bits at each robot to solve DISPERSION on lines, rings, and trees in  $O(n)$  time. For arbitrary graphs, they gave one algorithm using  $O(\log n)$  bits at each robot with  $O(mn)$  time and another using  $O(n \log n)$  bits at each robot with  $O(m)$  time.

Kshemkalyani and Ali [18] provided an  $\Omega(k)$  time lower bound for arbitrary graphs for  $k \leq n$ . They then provided three deterministic algorithms for DISPERSION in arbitrary graphs: (i) The first algorithm using  $O(k \log \Delta)$  bits at each robot with  $O(\min\{m, k\Delta\})$  time, (ii) The second algorithm using  $O(D \log \Delta)$  bits at each robot with  $O(\Delta^D)$  time ( $D$  is diameter of graph), and (iii) The third algorithm using  $O(\log(k + \Delta))$  bits at each robot with  $O(\min\{m, k\Delta\} \cdot \ell)$  time. Kshemkalyani et al. [19] provided an algorithm for arbitrary graph with  $O(\min\{m, k\Delta\} \cdot \log \ell)$  time using  $O(\log n)$  bits memory at each robot, with the algorithm knowing  $m, k, \Delta$  beforehand. The same time bound and improved memory bound of  $O(\log(k + \Delta))$  bits were obtained in [31], without the need of the algorithm knowing  $m, k, \Delta$  beforehand. For grid graphs, Kshemkalyani et al. [21] provided an algorithm that runs in  $O(\min\{k, \sqrt{n}\})$  time using  $O(\log k)$  bits memory at each robot. Randomized algorithms were presented in [24, 8] mainly to reduce the memory requirement at each robot.

Recently, Kshemkalyani et al. [20] provided an algorithm for arbitrary graphs with time  $O(\min\{m, k\Delta\})$  when all robots can communicate and exchange information in every round (that is even the non-co-located can communicate and exchange information, which is called the *global* communication model). The global model comes handy while dealing with subsuming the multiple DFS traversals that meet in the transient chain or meeting graph. The information each robot can have allows the head node of the highest ID DFS traversal (satisfying a certain property) in the transient chain/meeting graph to ask the head nodes of the rest of the DFS traversals to stop growing their DFS tree. This makes sure that one DFS traversal always grows and others stop as soon as they find that they were met by the DFS traversal that is of higher ID than theirs. The result presented in this paper is different since only the co-located robots can communicate and it is called the *local* communication model. In the local model, it is not possible to extend the idea that is developed for the global model. For grid graphs, Kshemkalyani et al. [21] provided a  $O(\sqrt{k})$  time algorithm with  $O(\log k)$  bits at each robot in the global model.

DISPERSION in anonymous dynamic (undirected) graphs was considered in [22] where the authors provided some impossibility, lower, and upper bound results. Dispersion under crash faults was considered in [27] and under Byzantine faults was considered in [25, 26] establishing a spectrum of interesting results.

The related problem of exploration has been quite heavily studied in the literature for specific as well as arbitrary graphs, e.g., [2, 4, 9, 15, 14, 17, 23]. It was shown that a robot can explore an anonymous graph using  $\Theta(D \log \Delta)$ -bits memory; the runtime of the algorithm is  $O(\Delta^{D+1})$  [15]. In the model where graph nodes also have memory, Cohen et al. [4] gave two algorithms: The first algorithm uses  $O(1)$ -bits at the robot and 2 bits at each node, and the second algorithm uses  $O(\log \Delta)$  bits at the robot and 1 bit at each node. The runtime of both algorithms is  $O(m)$  with preprocessing time of  $O(mD)$ . The trade-off between exploration time and number of robots is studied in [23]. The collective exploration by a team of robots is studied in [14] for trees. The dual of the DISPERSION problem is gathering, which has been extensively studied, e.g., [10, 16]. Another problem related to DISPERSION is the scattering of  $k$  robots on graphs. This problem has been mainly studied for rings [11, 30] and grids [3]. Recently, Poudel and Sharma [28, 29] provided improved time algorithms for uniform scattering on grids. Furthermore, DISPERSION is related to the load balancing problem, where a given load at the nodes has to be (re-)distributed among several processors (nodes). This problem has been studied quite heavily in graphs, e.g., see [7]. We refer readers to [12, 13] for other recent developments in these topics.

**Roadmap.** We discuss model details in Section 2. We discuss the single-source DFS traversal in Section 3. We then present our (synchronous) multi-source DFS traversal algorithm in Section 4. We prove the correctness, time, and memory complexity of our algorithm in Section 5 (i.e., Theorem 1). We then discuss the extensions to the asynchronous setting, proving Theorem 2. Finally, we conclude in Section 6 with a short discussion.

## 2 Model

**Graph.** Let  $G = (V, E)$  be a connected, unweighted, and undirected graph of  $n$  nodes,  $m$  edges, and maximum degree  $\Delta$ .  $G$  is *anonymous* – nodes do not have identifiers but, at any node, its incident edges are uniquely identified by a port number in the range  $[0, \delta - 1]$ , where  $\delta$  is the *degree* of that node. ( $\Delta$  is the maximum among the degree  $\delta$  of the nodes in  $G$ .) We assume that there is no correlation between two port numbers of an edge. Any number of robots are allowed to move along an edge at any time (i.e., unlimited edge bandwidth). The graph nodes are memory-less (do not have memory).

**Robots.** Let  $\mathcal{R} = \{r_1, r_2, \dots, r_k\}$  be the set of  $k \leq n$  robots residing on the nodes of  $G$ . No robot can reside on the edges of  $G$ , but one or more robots can occupy the same node of  $G$ , which we call co-located robots. In the initial configuration, we assume that all  $k$  robots in  $\mathcal{R}$  can be in one or more nodes of  $G$  but in the final configuration there must be exactly one robot on  $k$  different nodes of  $G$ . Suppose robots are on  $k' \leq k$  nodes of  $G$  in the initial configuration. We denote by  $\ell \leq k'$  the number of nodes in the initial configuration which have at least two robots co-located on them.

Each robot has a unique  $\lceil \log k \rceil$ -bit ID taken from the range  $[1, k]$ . When a robot moves from node  $u$  to node  $v$  in  $G$ , it is aware of the port of  $u$  it used to leave  $u$  and the port of  $v$  it used to enter  $v$ . We do not restrict time duration of local computation of the robots. The only guarantee is that all this happens in a finite cycle of “Communicate-Compute-Move” (defined below) and we measure time with respect to the number of cycles until DISPERSION is achieved. Furthermore, it is assumed that each robot is equipped with memory. The robots do not experience fault.

**Communication Model.** This paper considers the local communication model where only co-located robots at a graph node can communicate and exchange information. This model is in contrast to the global communication model where even non-co-located robots (i.e., at different graph nodes) can communicate and exchange information.

**Time Cycle.** An active robot  $r_i$  performs the “Communicate-Compute-Move” (CCM) cycle as follows. *Communicate:* Let  $r_i$  be on node  $v_i$ . For each robot  $r_j \in \mathcal{R}$  that is co-located at  $v_i$ ,  $r_i$  can observe the memory of  $r_j$ , including its own memory; *Compute:*  $r_i$  may perform an arbitrary computation using the information observed during the “communicate” portion of that cycle. This includes determination of a (possibly) port to use to exit  $v_i$ , the information to carry while exiting, and the information to store in the robot(s)  $r_j$  that stays at  $v_i$ ; *Move:*  $r_i$  writes new information (if any) in the memory of a robot  $r_j$  at  $v_i$ , and exits  $v_i$  using the computed port to reach to a neighbor node of  $v_i$ .

**Robot Activation.** In the *synchronous* setting, every robot is active in every CCM cycle. In the *asynchronous* setting, there is no common notion of time and no assumption is made on the number and frequency of CCM cycles in which a robot can be active. The only guarantee is that each robot is active infinitely often.

**Time and Memory Complexity.** For the synchronous setting, time is measured in *rounds*. Since a robot in the asynchronous settings could stay inactive for an indeterminate but finite time, we bound a robot’s inactivity introducing the idea of an epoch. An *epoch* is the smallest interval of time within which each robot is guaranteed to be active at least once [5]. Let  $t_i$  be the time at which a robot  $r_i \in \mathcal{R}$  starts its CCM cycle. Let  $t_j$  be the time at which the last robot finishes its CCM cycle. The time interval  $t_j - t_i$  is an epoch. Another important parameter is memory – the number of bits stored in persistent memory at each robot.

### 3 DFS traversal of a Graph (Algorithm $DFS(k)$ )

We describe here a single-source DFS traversal algorithm,  $DFS(k)$ , that disperses all  $k$  robots in the set  $R(v)$  situated at a node  $v$  initially to exactly  $k$  nodes of  $G$ , solving DISPERSION.  $DFS(k)$  will be heavily used in Section 4 as a basic building block.

Each robot  $r_i$  stores in its memory five variables. (i) *parent* (initially assigned  $\perp$ ), for a settled robot denotes the port through which it first entered the node it is settled at; (ii) *child* (initially assigned  $-1$ ), for an unsettled robot  $r_i$  stores the port that it has last taken (while entering/exiting the node). For a settled robot, it indicates the port through which the other robots last left the node except when they entered the node in forward mode for the second or subsequent time; (iii) *treelabel* (initially assigned  $\min\{R(v)\}$ ) stores the ID of the smallest ID robot the tree is associated with; (iv) *state*  $\in \{forward, backtrack, settled\}$  (initially assigned *forward*).  $DFS(k)$  executes in two phases, *forward* and *backtrack* [6]; (v) *rank* (initialized to 0), for a settled robot indicates the serial number of the order in which it settled in its DFS tree. The algorithm pseudo-code is shown in Algorithm 1. The robots in  $R(v)$  move together in a DFS, leaving behind the highest ID robot at each newly discovered node. They all adopt the ID of the lowest ID robot in  $R(v)$  which is the last to settle, as their *treelabel*. The algorithm executes in forward and backtrack modes.

► **Theorem 3** ([19]). *Algorithm  $DFS(k)$  solves DISPERSION for  $k \leq n$  robots initially positioned on a single node of an arbitrary anonymous graph  $G$  of  $n$  memory-less nodes,  $m$  edges, and degree  $\Delta$  in  $\min\{4m - 2n + 2, 4k\Delta\}$  rounds using  $O(\log(k + \Delta))$  bits at each robot.*

■ **Algorithm 1** Algorithm  $DFS(k)$  for DFS traversal of a graph by  $k$  robots from a rooted initial configuration. Code for robot  $i$ .  $r$  is robot settled at the current node.

---

```

1 Initialize:  $child \leftarrow -1$ ,  $parent \leftarrow \perp$ ,  $state \leftarrow forward$ ,  $treelabel \leftarrow \min\{R(v)\}$ ,  $rank \leftarrow 0$ 
2 for  $round = 1$  to  $\min\{4m - 2n + 2, 4k\Delta\}$  do
3    $child \leftarrow$  port through which node is entered
4   if  $state = forward$  then
5     if  $node$  is free then
6        $rank \leftarrow rank + 1$ 
7       if  $i$  is the highest ID robot on the node then
8          $state \leftarrow settled$ ,  $i$  settles at the node (does not move henceforth),
9          $parent \leftarrow child$ ,  $treelabel \leftarrow$  lowest ID robot at the node
10      else
11         $child \leftarrow (child + 1) \bmod \delta$ ,  $r.child \leftarrow child$ 
12        if  $child = parent$  of robot settled at node then
13           $state \leftarrow backtrack$ 
14      else
15         $state \leftarrow backtrack$ 
16    else if  $state = backtrack$  then
17       $child \leftarrow (child + 1) \bmod \delta$ ,  $r.child \leftarrow child$ 
18      if  $child \neq parent$  of robot settled at node then
19         $state \leftarrow forward$ 
20  move out through  $child$ 

```

---

#### 4 The Algorithm

The *root* of a DFS  $i$  (which equals the identifier (*treelabel*)) is the node where the first robot settles. This is the settled robot having  $rank = 1$ . The *head* of a DFS  $i$  is the node where the unsettled robots (if any) of that DFS are currently located at, or else it is the node where the last robot of that DFS settled. Node  $root(i)$  is reachable by following *parent* pointers; node  $head(i)$  is reachable by following *child* pointers.

In the initial configuration, if robots are at  $k' < k$  nodes ( $k' = k$  solves DISPERSION in the first round without any robot moving),  $k'$  DFS traversals are initiated in parallel. A DFS  $i$  *meets* DFS  $j$  if the robots of DFS  $i$  arrive at a node  $x$  where a robot from DFS  $j$  is settled. Node  $x$  is called a *junction* node of  $head(i)$ . If robots from multiple DFSs/nodes arrive at a node where there is no settled robot, a robot from the DFS with the highest ID settles in that round and the other DFSs are said to meet this DFS. If DFS  $i$  has met DFS  $j$ , we define  $head(i)$  to be *blocked*, else we define  $head(i)$  to be *free*.

The *size*  $d_i$  of a DFS  $i$  is the number of settled robots in that DFS. When DFS  $i$  meets DFS  $j$ , the first task is to determine whether  $d_i > d_j$  or  $d_j > d_i$ , where we define a total order ( $>$ ) by using the DFS IDs as tiebreakers if the number of settled robots is the same.  $d_i$  is known to robots of DFS  $i$  at  $head(i)$  by reading *rank* of DFS tree  $i$ . The unsettled robots at  $head(i)$  traverse DFS  $j$  to  $head(j)$  in an exploration to determine  $d_j$ . If they reach  $head(j)$  without encountering a node with *rank* greater than  $d_i$ , then  $d_i > d_j$ . The junction  $head(j)$  is defined to be *locked* by  $i$  if DFS  $i$ 's robots are the first to reach  $head(j)$  in such an exploration (and at this time,  $j$ 's exploratory robots have yet to return to  $head(j)$ ). However, if the exploratory robots of DFS  $i$  encounter a node with *rank* greater than  $d_i$  before reaching  $head(j)$ , they return to  $head(i)$  as  $d_j > d_i$ . A key advantage of this mechanism is that  $d_i > d_j$  can be determined in time proportional to  $\min\{d_i, d_j\}$ .

■ **Algorithm 2** Algorithm *Exploration* to explore  $parent(i)$  component on reaching junction  $head(i)$  by DFS of component  $i$ .

---

```

1 explorers move to  $root(parent(i))$  leaving retrace pointers for return path. Then they follow
  child pointers from  $root(parent(i))$  to  $head(parent(i))$ . There are 4 possibilities.
2 if  $d_{parent(i)} > d_i$ , i.e., rank  $> d_i$  is encountered, implying explorers do not reach
   $head(parent(i))$  (possibly the next junction) then
3   return to  $head(i)$  junction
4   if  $head(i)$  is not locked then
5     | Collapse_Into_Parent( $i$ )
6   else if  $head(i)$  is locked by  $j$  then
7     | Collapse_Into_Child( $i, j$ )
8 else if  $d_{parent(i)} < d_i$ , implying  $head(parent(i))$  is reached (possibly next junction) then
9   lock  $head(parent(i))$ 
10  traverse  $parent(i)$  informing each node (a) that  $parent(i)$  is locked and will be
    collapsing, and also (b) value of  $d_{parent(i)}$ , and return to  $head(parent(i))$ 
11  wait until  $parent(i)$ 's explorers return from  $parent(parent(i))$ 
12  follow action (Collapse_Into_Child( $parent(i), i$ )) which will be determined on their
    return (if  $head(parent(i))$  is not junction, execute Collapse_Into_Child( $parent(i), i$ ))
13 else if exploring robots find  $parent(i)$  is collapsing or learn that  $parent(i)$  is locked and will
    be collapsing then
14   | Parent_Is_Collapsing
15 else if explorers  $E$ 's path meets another explorers  $F$ 's path then
16   wait until  $F$  return
17   if  $parent(i)$  is collapsing then
18     | Parent_Is_Collapsing
19   else if  $parent(i)$  is not collapsing then
20     | continue  $E$ 's exploration

```

---

Knowing the sizes, the general idea is that if  $d_i$  is greater, DFS  $j$  is *subsumed* by DFS  $i$  and DFS  $j$  collapses by having all its robots collected to the  $head(i)$  to continue DFS  $i$ . This collapse however cannot begin immediately because  $j$ 's robots may be exploring the DFS  $l$  it has met and they must return to  $head(j)$  before  $j$  starts its collapse. (The algorithm ensures there are no such cyclic waits to prevent deadlocks.) However, if  $d_j$  is greater, DFS  $i$  gets *subsumed*, i.e., DFS  $j$  subsumes DFS  $i$ . The free robots of  $i$  exploring  $j$  return to  $head(i)$ , DFS  $i$  collapses by having all its robots collected to  $head(i)$ , and then they all move to  $head(j)$  to continue DFS  $j$ . Now, these above policies regarding which DFS collapses and gets subsumed by which other have to be adapted to the following fact – due to concurrent actions in different parts of  $G$ , a DFS  $j$  may be met by different other DFSs, and DFS  $j$  may in turn meet another DFS concurrently. Further, transitive chains of such meetings can occur concurrently. This leads us to formalize the notion of a *meeting graph*.

► **Definition 4** (Meeting graph). *The directed meeting graph  $G' = (V', E')$  is defined as follows.  $V'$  is the set of concurrently existing DFS IDs. There is a (directed) edge in  $E'$  from  $i$  to  $j$  if DFS  $i$  meets DFS  $j$ .*

For an edge  $(i, j)$  in the meeting graph, DFS  $j$  is defined to be  $parent(i)$  and DFS  $i$  is defined to be  $child(j)$ . The size of a node in the meeting graph is defined to be the size of the DFS for that node. Nodes in  $V'$  have an arbitrary in-degree ( $< k'$ ) but out-degree at most 1. There may also be a cycle in each connected component of  $G'$ . Henceforth, we

■ **Algorithm 3** Algorithms *Collapse\_Into\_Child*, *Collapse\_Into\_Parent*, and *Parent\_Is\_Collapsing*.

---

```

1 Collapse_Into_Child( $i, j$ )
2 explorers of  $i$  go from  $head(i)$  locked by  $j$  to  $root(i)$ 
3 explorers of  $i$  do  $i$ 's DFS tree traversal collecting all robots to collapse path ( $root(i)$  to
    $head(j)$ ) marked by retrace pointers, waiting until collapsing_children = 0 at each node
4 from  $root(i)$  collect all robots accumulated on collapse path to  $j$ 's junction  $head(j)$ 
5 collapsed robots change ID treelabel to  $j$ 
6 if  $head(j)$  is locked by  $l$  then
7   | Collapse_Into_Child( $j, l$ )
8 else if  $head(j)$  is not locked then
9   | continue  $j$ 's DFS
10 Collapse_Into_Parent( $i$ )
11 robot at  $head(i)$  increments collapsing_children
12 explorers of  $i$  go from  $head(i)$  to  $root(i)$  leaving collapse pointers
13 explorers of  $i$  do  $i$ 's DFS tree traversal collecting all robots to collapse path ( $root(i)$  to
    $head(i)$ ) marked by collapse pointers, waiting until collapsing_children = 0 at each node
14 from  $root(i)$  collect all robots accumulated on collapse path to  $i$ 's junction  $head(i)$ 
15 robot at  $head(i)$  decrements collapsing_children
16 collapsed robots change ID treelabel to  $parent(i)$ 
17 explorers and collapsed robots go to  $head(parent(i))$  by following child pointers
18 if  $parent(i)$  along the way is found to be collapsing then
19   | collapse with it; break()
20 if  $head(parent(i))$  is free then
21   | continue  $parent(i)$ 's DFS
22 else if  $head(parent(i))$  is blocked and possibly also locked then
23   | wait until  $parent(i)$  collapses (and collapse with it) or becomes unblocked (and continue
   |  $parent(i)$ 's action)
24 Parent_Is_Collapsing
25 retrace path to  $head(i)$  junction
26 if  $d_i < d_{parent(i)}$  and  $head(i)$  junction is not locked then
27   | Collapse_Into_Parent( $i$ )
28 else if  $d_i > d_{parent(i)}$  and  $head(i)$  junction is not locked and remains unlocked until
    $parent(i)$ 's collapse reaches  $head(i)$  then
29   | unsettled robots get absorbed in  $parent(i)$  during its collapse
30 else if  $head(i)$  junction of  $i$  (is locked by  $j$ ) or (gets locked by  $j$  before  $parent(i)$ 's collapse
   reaches  $head(i)$  and  $d_i > d_{parent(i)}$ ) then
31   | Collapse_Into_Child( $i, j$ )

```

---

focus on a single connected component of  $G'$  by default; other connected components are dealt with similarly. The algorithm implicitly partitions a connected component of  $G'$  into (connected) sub-components such that each sub-component is defined to have a master node  $M$  into which all other nodes of that sub-component are subsumed, directly or transitively. In this process, the at most one cycle in any connected component of  $G'$  is also broken. In each sub-component, the master node  $M$  has the highest value of  $d$  and the other smaller (or equal sized) nodes, i.e., DFSs, get subsumed. The pseudo-code is given in Algorithm 2 and in Algorithm 3. In Algorithm 2,  $j$  is explored by robots from  $i$  to determine if  $d_i > d_j$  (therefore, we sometimes call Algorithm 2 *Exploration*), and the appropriate procedures for collapsing and collecting are given in Algorithm 3 (therefore, we sometimes call Algorithm 3 *various procedures invoked*).



■ **Algorithm 4** Algorithm *Determine\_Master(i)* to identify master component in which component  $i$  will collapse.

---

```

1 master(i)
2 if  $d_{parent(i)} > d_i$  then
3    $t1 \leftarrow$  time when explorers of  $i$  return to  $head(i)$  from  $parent(i)$ 
4    $t2$  (initialized to  $\infty$ )  $\leftarrow$  the time, if any, when first child  $j$  locks  $head(i)$ 
5   if  $t1 < t2$  then  $w \leftarrow parent(i)$ 
6   else if  $t1 > t2$  then  $w \leftarrow j$ 
7   return(master(w))
8 else
9   if  $\exists$  a first child  $j$  to lock  $head(i)$  then return(master(j))
10  else return( $i$ )

```

---

For any given node  $i \in V'$ , its master node is given as per Algorithm 4. Note that this algorithm is not actually executed and the master node of a node need not be known – it is given only to aid our understanding and in the complexity proof. If *master(j)* gets invoked directly or transitively in the invocation of *master(i)* for any  $i$ , then  $i$  must be subsumed and its robots collected completely before  $j$  gets subsumed and its robots are collected completely.

A path in  $G'$  is an *increasing* (*decreasing*) path if the node sizes along the path are increasing (decreasing). For a master node  $M$ , the nodes  $x$  in its sub-component of  $G'$  that directly and transitively participate in only *Collapse\_Into\_Parent* and no *Collapse\_Into\_Child* until collapsing into  $M$  form the set  $X(M)$ . Whereas the (other) nodes  $y$  in the sub-component that directly and transitively invoke at least one *Collapse\_Into\_Child* until they collapse into  $M$  belong to the set  $Y(M)$ . The component  $C(M) = X(M) \cup Y(M) \cup \{M\}$ .

A component  $C(M)$  is acyclic. For an edge  $(i, j)$ ,  $i$  is the child and  $j$  is the parent. Nodes in the set  $X$  have an increasing path to the master node. They collapse into and get subsumed by the master node (possibly transitively) by executing *Collapse\_Into\_Parent*. Nodes in the set  $Y$  are reachable from the master node on a decreasing path – such nodes are termed *Y\_trunk* nodes, or have a increasing path to a *Y\_trunk* node – such nodes are termed *Y\_branch* nodes. Nodes in  $Y$  (i.e., in *Y\_trunk* and *Y\_branch*) collapse into and get subsumed by the master node, possibly transitively. First, the *Y\_branch* nodes collapse into and get subsumed by their ancestors on the increasing path ending in a *Y\_trunk* node by executing *Collapse\_Into\_Parent*; then the *Y\_trunk* nodes collapse and get subsumed into their child nodes along *Y\_trunk* and then into the master node by executing *Collapse\_Into\_Child*.

After nodes in  $C(M)$  get subsumed in  $M$ , the master node grows again until involved in more meetings and new meeting graphs are formed. Thus the meeting graph is dynamic. We define a related notion of a *meeting tree* that represents which nodes (DFSs) have met and been subsumed by which master node, in which meeting sequence number of meetings for each such node.

► **Definition 5** (Meeting tree). *The  $k'$  initial DFSs  $i$  form the  $k'$  leaf nodes  $(i, 0)$  at level 0. When  $\alpha$  nodes  $(a_i, h_i)$  for  $i \in [1, \alpha]$  meet in a component and get subsumed by the master node with DFS identifier  $M$  of the meeting graph, a node  $(M, h)$ , where  $h = 1 + \max_{i \in [1, \alpha]} h_i$ , is created in the meeting tree as the parent of the child nodes  $(a_i, h_i)$ , for  $i \in [1, \alpha]$ .*

For a node  $(M, h)$ ,  $h$  is the length of the longest path from some leaf node to that node. We now formally define  $X(M, h)$ ,  $Y(M, h)$ , and  $C(M, h)$ .

► **Definition 6** (Component  $C(M, h)$ ).

1.  $X(M, h)$  is the set of child nodes in the meeting tree that directly and transitively participate only in *Collapse\_Into\_Parent* until collapsing into  $(M, h)$ .
2.  $Y(M, h)$  is the set of child nodes in the meeting tree that directly and transitively participate in at least one *Collapse\_Into\_Child* until collapsing into  $(M, h)$ .
3.  $C(M, h) = X(M, h) \cup Y(M, h) \cup \{(M, \text{prev}(h))\}$ , where for any  $z \in C(M, h)$ ,  $z = (a, \text{prev}(h))$  and  $\text{prev}(h)$  is defined as the highest value less than  $h$  for which node  $(a, \text{prev}(h))$  has been created.

For any node  $(i, h)$ , we also define  $\text{next}(h)$  as the value  $h'$  such that  $(i, h) \in C(M, h')$  for some  $M$ . If such a  $h'$  does not exist, we define it to be  $k'$ .

We omit  $h$  in  $(i, h)$  and  $C(M, h)$  in places where it is understood or not required.

## 5 Analysis of the Algorithm

In our algorithm, a common module is to traverse an already identified DFS component with nodes having the same *treelabel*. This can be achieved by going to  $\text{root}(i)$  and doing a (new) DFS traversal of only those nodes (using a duplicate set of variables *state* and *parent* for DFS); if you reach a node which has no settled robot or a settled robot having a different *treelabel*, one simply backtracks along that edge. Such a DFS traversal occurs in (i) Algorithm *Exploration* when  $d_i > d_{\text{parent}(i)}$  and  $i$  locks  $\text{head}(\text{parent}(i))$  junction, (ii) procedure *Collapse\_Into\_Child*, and (iii) procedure *Collapse\_Into\_Parent*, and can be executed in  $4\Delta d_i$  steps. In (ii) and (iii), a settled robot not on the collect path gets unsettled and gets collected in the DFS traversal to the collect path when the DFS backtracks from the node where the robot was settled.

The time complexity of Algorithms 2 and 3 is as follows.

1. Algorithm 2 takes time bounded by  $8d_i\Delta + 3d_i$ . The derivation is as follows.
  - a.  $\min\{d_i, d_{\text{parent}(i)}\}$  to go from  $\text{head}(i)$  to  $\text{root}(\text{parent}(i))$ .
  - b.  $4 \min\{d_i, d_{\text{parent}(i)}\}\Delta$  to go then to  $\text{head}(\text{parent}(i))$ .
  - c. if  $d_{\text{parent}(i)} > d_i$ , then  $2d_i$  to return to  $\text{head}(i)$  via  $\text{root}(\text{parent}(i))$ .
  - d. if  $d_{\text{parent}(i)} < d_i$  and  $i$  locks  $\text{head}(\text{parent}(i))$ , then  $4d_{\text{parent}(i)}\Delta + 2d_{\text{parent}(i)}$  for DFS traversal of  $\text{parent}(i)$  component from  $\text{root}(\text{parent}(i))$  plus to  $\text{root}(\text{parent}(i))$  from  $\text{head}(\text{parent}(i))$  and back.

If explorers  $E$ 's path meets explorers  $F$ 's path, the explorers  $E$  wait until  $F$ 's return. This delay is analyzed later.
2. In Algorithm 3,
  - a. *Collapse\_Into\_Child* takes  $4d_i\Delta + 2d_i$ .  
Time  $d_i$  to go from  $\text{head}(i)$  to  $\text{root}(i)$ ;  $4\Delta d_i$  for a DFS traversal of  $i$  component from  $\text{root}(i)$ ; and  $d_i$  to collect the accumulated robots from  $\text{root}(i)$  to  $\text{head}(j)$  along the collapse path.
  - b. *Collapse\_Into\_Parent* takes  $4d_i\Delta + 2d_i + 4d_{\text{parent}(i)}\Delta$ .  
Time  $d_i$  to go from  $\text{head}(i)$  to  $\text{root}(i)$ ;  $4\Delta d_i$  for a DFS traversal of  $i$  component from  $\text{root}(i)$ ;  $d_i$  to collect the accumulated robots from  $\text{root}(i)$  to  $\text{head}(i)$ ; and  $4d_{\text{parent}(i)}\Delta$  to then go to  $\text{head}(\text{parent}(i))$ .
  - c. The cost of *Parent\_Is\_Collapsing* is  $\min\{d_i, d_{\text{parent}(i)}\}$  but is subsumed in the cost of Algorithm 2.  
This cost is to return to  $\text{head}(i)$  from the exploration point in  $\text{parent}(i)$  component where it is invoked.

The contributions to this time complexity by the various nodes in  $C(M)$  are as follows. (The cost is the sum of Algorithm *Exploration* plus appropriate invoked procedure costs.)

1. Each  $x \in X$  executes *Collapse\_Into\_Parent* after *Exploration*, as it is part of an increasing path. So it contributes the sum of the two contributions, giving  $12d_x\Delta + 5d_x + 4d_{parent(x)}\Delta$ .  
The  $4d_{parent(x)}\Delta$  is for traversing to  $head(parent(x))$  after  $x$  collapses to  $head(x)$ , and this can be done concurrently by multiple  $x$  that are children of the same parent. As each  $x$  can be thought of as the *parent* of another element in  $X$ , so the cost of subsuming the  $X$  set is  $\sum_{x \in X} 16d_x\Delta + 5d_x + (\text{if } X \neq \emptyset, 4d_M\Delta)$ .
2. Each  $y \in Y\_branch$  executes *Collapse\_Into\_Parent* after *Exploration*, as it is part of an increasing path. So it contributes the sum of the two contributions, giving  $16d_y\Delta + 5d_y$ . Each  $y \in Y\_trunk$  executes *Collapse\_Into\_Child* after *Exploration*, as it is part of a decreasing path. So it contributes the sum of the two contributions, giving  $12d_y\Delta + 5d_y$ , plus it potentially acts as a parent of a node on a  $Y\_branch$  that executed *Collapse\_Into\_Parent* so it contributes an added  $4d_y\Delta$ , giving a total of  $16d_y\Delta + 5d_y$ .
3. Node  $M$  will contribute in Algorithm *Exploration*  $4 \min\{d_M, d_{parent(M)}\}\Delta + \min\{d_M, d_{parent(M)}\}$ , plus  $4d_{parent(M)}\Delta + 2d_{parent(M)}$  as  $parent(M)$  is smaller. Thus, a total of  $8d_{parent(M)}\Delta + 3d_{parent(M)}$ . This can be counted towards a contribution by  $parent(M) = y \in Y$ , thus the contribution of each  $y \in Y$  can be bounded by  $24d_y\Delta + 8d_y$  with  $M$  contributing nil.

There is another source of time overhead contributed by nodes in  $Y\_trunk \cup \{M\}$ . Nodes  $y$ , i.e.,  $head(y) \in G$ , for  $y \in Y\_trunk$ , are locked by their child. Before this can happen, other children of  $y$  may be exploring  $y$  by leaving *retrace* pointers. However, due to the  $O(\log(k + \Delta))$  bits bound on memory at each robot, a retrace pointer at a node in  $y$  can be left by only  $O(1)$  children, not by  $O(k')$  children. Therefore in Algorithm 2, if explorers  $E$  path meets another explorers  $F$  path, they wait at the meeting node until  $F$  return. If they learn that the  $y$  is collapsing, they retrace to their *head* nodes else if they learn  $y$  is not collapsing, they continue their exploration towards  $head(y)$  but may be blocked again if their path meets another explorers' path. This waiting due to concurrently exploring children introduces delays.

A child of  $y$  outside  $Y\_trunk$  may be either locked ( $l$ ) or unlocked ( $u$ ) and is also smaller ( $S$ ) or larger ( $L$ ) than  $y$ . Thus, there are 4 classes of such children.

1.  $Su$ -type children belong to  $Y\_branch$  and their introduced delays are already accounted for above.
2. Each  $Lu$ -type and  $Ll$ -type child does not contribute any delay. This is because even though these children are larger than  $y$ , they are not the child in  $Y$  who succeeds in locking  $y$ ; the child in  $Y$  who locks  $y$  does so before such  $L*$ -type children try to explore  $y$  and try to lock  $y$ . Such  $L*$ -type children learn that  $y$  is collapsing.
3. Each  $Sl$ -type child node  $b$  contributes delay  $4d_b\Delta + 3d_b$ . The sum of such delays at  $y$  is denoted  $t_{y(M,h)}$ . Later, we show how to bound the sum of such delays across multiple  $M$ ,  $h$  and  $y$ .

Similar reasoning can be used for  $M$  delaying its children in  $X$  due to explorations of other children  $z \notin X$ . Specifically, (1) type  $Su$  child  $z$  of  $M$ :  $\nexists$  child  $z \notin X$ . (2) type  $L*$  child  $z$  of  $M$ :  $\nexists$  such a child  $z$ . If it existed, it would have succeeded in locking  $M$  and  $M$  would not be master. (3) Each type  $Sl$  child  $z$  contributes delay  $4d_z\Delta + 3d_z$ , whose sum for all  $z$  is denoted by  $t_{(M,prev(h))}$ . Later, we show how to bound the sum of such delays across multiple  $M$  and  $h$ .

## 8:14 Near-Optimal Dispersion on Arbitrary Anonymous Graphs

Note that for any  $x \in X$ , (1) each type *Su* child belongs to  $X$  and the delay is already accounted for in *Collapse\_Into\_Parent* executed by  $x$ . (2) each type *Sl* child and type *L\** child does not contribute any delay beyond that of *Collapse\_Into\_Parent* executed by  $x$  and already accounted for. (The type *L\** child does not succeed in locking  $head(x)$  and learns that  $x$  is collapsing into its parent.)

Thus far, the size  $d_i$  of node  $i$  referred to the number of settled robots in it, and is henceforth referred to as  $d_i^s$ . More specifically,  $d_{i,h}^s$  will refer to the number of settled robots up until just before the  $next(h)$  meeting of  $i$ . The number of unsettled robots in  $i$  up until just before the  $next(h)$  meeting of  $i$  is referred to as  $d_{i,h}^u$ . Let  $T(M, h)$  denote the time to settle DFS  $M$  up until meeting at depth  $h$  of the meeting tree, and from then on until the next meeting ( $next(h)$ ) for  $M$ . The collapse and collection time to  $head(M)$  has components  $c(M, h)$  and  $g(M, h)$ .  $c(M, h)$  has an upper bound factor of  $(24\Delta + 8)$  for  $x \in X$  and  $y \in Y$  as derived above. The time for dispersion/settling after collection and until the  $next(h)$  meeting is  $s(M, h)$ . These are defined as follows.

$$c(M, h) = \begin{cases} 0 & \text{if } h = 0 \\ (24\Delta + 8)(\sum_{x \in X(M, h)} d_x^s + \sum_{y \in Y(M, h)} d_y^s) & \text{if } h > 0 \\ (+4\Delta(d_{M, prev(h)}^s)) & \text{if } X(M, h) \neq \emptyset \end{cases} \quad (1)$$

$$s(M, h) = \begin{cases} 4\Delta(d_{M, h}^s - d_{M, prev(h)}^s) & \text{if } next(h) < k' \\ 4\Delta(\sum_{x \in X(M, h)} d_x^s + \sum_{y \in Y(M, h)} d_y^s) & \text{otherwise} \\ + \sum_{x \in X(M, h)} d_x^u + \sum_{y \in Y(M, h)} d_y^u & \\ + d_{M, prev(h)}^u & \end{cases} \quad (2)$$

$$g(M, h) = \begin{cases} 0 & \text{if } h = 0 \\ \sum_{y \in Y(M, h)} t_y + t_{(M, prev(h))} & \text{if } h > 0 \end{cases} \quad (3)$$

This process of collapsing and collecting for instance  $(M, h)$  began at the very latest (since the start of the algorithm) at the time at which the latest of the  $x$  nodes,  $x'$ , got blocked. Thus,

$$\begin{aligned} T(M, h) &\leq \overbrace{c(M, h) + s(M, h)}^{f(M, h)} + g(M, h) + T(x', prev(h)), \\ x' &= \operatorname{argmax}_{x \mid (x, prev(h)) \in X(M, h) \cup \{(M, prev(h))\}} T(x, prev(h)), \\ c(*, 0) &= 0, g(*, 0) = 0, s(*, 0) = d_{*, 0}^s. \end{aligned} \quad (4)$$

We break  $T(M, h)$  into two series, and bound them separately. The two series are:

$$\begin{aligned} S1 &= f(M, h) + f(x'(M, h), prev(h)) \\ &\quad + f(x'(x'(M, h), prev(h)), prev(prev(h))) + \dots + f(*, 0) \\ S2 &= g_{(M, h)} + g_{(x'(M, h), prev(h))} + \dots + (g(*, 0) = 0) \\ &= \sum_{y \in Y(M, h)} t_y + \sum_{y \in Y(x'(M, h), prev(h))} t_y + \dots + (\sum_{y \in Y(*, 0)} t_y = 0) \\ &\quad + t_{(M, prev(h))} + t_{(x'(M, h), prev(prev(h)))} + \dots + (t_{(*, prev(0))} = 0) \end{aligned} \quad (5)$$

► **Lemma 7.** *The sum in the series S1 is  $O(k\Delta)$ .*

**Proof.** We consider levels of the meeting tree from level 1 upwards to  $h$  ( $\leq k' - 1$ ). Let  $\eta$  DFS components collapse and merge into one of them, and let the size (i.e., number of settled robots) of each component be  $d$ . We consider two extreme cases and show for each that the lemma holds.

1. Case 1: At each level when components collapse and collect in a master component, immediately afterwards (before the collected unsettled robots can settle) the master component meets another component at the next level, and the collapse and collection happen at the next level. Again, immediately afterwards, the (new) master component meets another component at the yet next higher level, and so on till level  $h$ . This case assumes  $s(i, *) = 0$ .
  - a. At level 1,  $\eta$  components of size  $d$  each merge into one of size  $d$  in  $O(\eta d \Delta)$  time, leading to a total of  $\eta d$  robots in the master component.
  - b. At level 2,  $\eta$  components of size  $d$  each merge into one of size  $d$  in  $O(\eta d \Delta)$  time, leading to a total of  $\eta^2 d$  robots in the master component.
  - c. At level  $h$ ,  $\eta$  components of size  $d$  each merge into one of size  $d$  in  $O(\eta d \Delta)$  time, leading to a total of  $\eta^h d$  robots in the master component.

$\eta^h d$  is at most the maximum number of robots  $k$ . Solving  $k = \eta^h d$ ,  $h = \log_\eta \frac{k}{d}$ . Therefore the maximum total elapsed time until the  $h$ -th level meeting and collapse takes place is

$$\text{Max. elapsed time is } O(h(\eta d \Delta)) = O(\eta d \Delta \log_\eta \frac{k}{d})$$

This maximum elapsed time is  $O(k \Delta)$ , considering both extreme cases (a)  $\eta d = O(1)$  and (b)  $\eta d = O(k)$ .

2. Case 2: At each level when components collapse and collect in a master component, the collected robots (almost) fully disperse after which the master component meets another component at the next level, and the collapse and collection happen at the next level. Again, the robots collected by the (new) master component (almost) fully disperse after which the master component meets another component at the yet next higher level, and so on till level  $h$ . This case assumes  $\forall j, s(i, j)$  satisfies  $next(j) \not\leq k'$ .
  - a. At level 1,  $\eta$  components of size  $d$  each merge into one of size  $\eta d$  in  $O(\eta d \Delta)$  time, leading to a total of  $\eta d$  robots in the master component.
  - b. At level 2,  $\eta$  components of size  $\eta d$  each merge into one of size  $\eta^2 d$  in  $O(\eta^2 d \Delta)$  time, leading to a total of  $\eta^2 d$  robots in the master component.
  - c. At level  $h$ ,  $\eta$  components of size  $\eta^{h-1} d$  each merge into one of size  $\eta^h d$  in  $O(\eta^h d \Delta)$  time, leading to a total of  $\eta^h d$  robots in the master component.

$\eta^h d$  is at most the maximum number of robots  $k$ . Solving  $k = \eta^h d$ ,  $h = \log_\eta \frac{k}{d}$ . Therefore the maximum total elapsed time until the  $h$ -th level meeting and collapse/dispersion takes place is

$$\begin{aligned} O(\Delta(\eta d + \eta^2 d + \eta^3 d + \dots + \eta^h d)) &= O(\Delta \eta d \frac{\eta^h - 1}{\eta - 1}) \\ &= O(\frac{\Delta \eta d}{\eta - 1} (\eta^{\log_\eta \frac{k}{d}} - 1)) \\ &= O(\frac{\Delta \eta d}{\eta - 1} (\frac{k}{d} - 1)) \\ &= O(k \Delta) \end{aligned}$$

There is also a special case in which a single component  $M$ , each time ( $\forall h'$ ), grows and meets other fully dispersed component(s) that collapse (transitively) in to it and no component meets  $M$ . Here,  $\forall h', X(M, h') = \emptyset$  as all subsumed components belong to  $Y(M, h')$  sets. Observe that  $\sum_{h'} c(M, h') = \sum_{h'} s(M, h') = O(k \Delta)$ .

The lemma follows. ◀

► **Lemma 8.** *The sum in the series  $S_2$  is  $O(k\Delta)$ .*

Proof is deferred to Appendix due to space constraints.

► **Theorem 9.** *Algorithm Exploration (Algorithm 2) in conjunction with Algorithm DFS( $k$ ) correctly solves DISPERSION for  $k \leq n$  robots initially positioned arbitrarily on the nodes of an arbitrary anonymous graph  $G$  of  $n$  memory-less nodes,  $m$  edges, and degree  $\Delta$  in  $O(\min\{m, k\Delta\})$  rounds using  $O(\log(k + \Delta))$  bits at each robot.*

**Proof.**  $T(M, h)$  is the sum of the series  $S_1$  and  $S_2$  which are both  $O(k\Delta)$  by Lemmas 7 and 8. So the time till termination of the Algorithms 1 (DFS), 2 (Exploration), and Algorithm 3 (various procedues invoked) is  $O(k\Delta)$ . As  $k \leq n$ , this is  $O(n\Delta)$ . Now observe that in our derivations (Lemmas 7 and 8), the  $\Delta$  factor is an overestimate. The actual upper bound is  $O(\sum_{i=1}^n \delta_i)$  which is  $O(m)$ , the number of edges in the graph. This upper bound is better when  $m < k\Delta$  and hence the time complexity is  $O(\min\{m, k\Delta\})$ .

The highest level node  $(i, h)$  in each tree in the final forest of the meeting graph represents a master node that has never been subsumed and always alternated between growing and subsuming other components, and growing again. The growth happens as per Algorithm 1 (DFS) which correctly solves DISPERSION by Theorem 3. Whereas the subsuming of other components merely collects the robots of the other components to the head node  $head(i)$  (Algorithm Exploration) which subsequently get dispersed by the growing phases (Algorithm DFS). Hence, DISPERSION is achieved.

The *retrace* and *collapse* variable at each robot used in Algorithm 2 and 3 are  $O(\log \Delta)$ . *collapsing\_children* takes  $O(\log k)$  bits and a single bit each is required to track whether the component is locked and whether it is collapsing. The space requirement of Algorithm 1 was shown in Theorem 3 to be  $(\log(k + \Delta))$  bits. The theorem follows. ◀

**Proof of Theorem 1.** Follows from Theorem 9. ◀

**Proof of Theorem 2.** In the asynchronous setting, in every CCM cycle, each robot at a node  $u$  determines  $x$ , the number of co-located robots, if any, that should be moving with it to node  $v$ . It then moves as per its own schedule. On arriving at  $v$ , it does not start its next CCM cycle until  $x$  robots have arrived from  $u$ . This essentially constitutes one epoch and ensures that the robots that move together in a round in a synchronous setting move together in one epoch in the asynchronous setting. With this simple modification, the algorithm given for the synchronous setting works for the asynchronous setting. The space and time complexities, as given in Theorem 1, carry over to the asynchronous setting. ◀

## 6 Concluding Remarks

In this paper, we have presented a deterministic algorithm that solves DISPERSION, starting from any initial configuration of  $k \leq n$  robots positioned on the nodes of an arbitrary anonymous graph  $G$  having  $n$  memory-less nodes,  $m$  edges, and degree  $\Delta$ , in time  $O(\min\{m, k\Delta\})$  with  $O(\log(k + \Delta))$  bits at each robot. This is the first algorithm that is simultaneously optimal w.r.t. both time and memory in arbitrary anonymous graphs of constant degree, i.e.,  $\Delta = O(1)$ . This algorithm improves the time bound established in the best previously known results [19, 31] by an  $O(\log \ell)$  factor and matches asymptotically the time and memory bound of the single-source DFS traversal. This algorithm uses a non-trivial approach of subsuming parallel DFS traversals into single one based on their DFS tree sizes, limiting the subsumption process overhead to the time proportional to the time needed in the single-source DFS traversal. This approach might be of independent interest.

For future work, it will be interesting to improve the existing time lower bound of  $\Omega(k)$  to  $\Omega(\min\{m, k\Delta\})$  or improve the time bound to  $O(k)$  removing the  $O(\Delta)$  factor. The second interesting direction will be to consider faulty (crash and/or Byzantine) robots.

---

## References

- 1 John Augustine and William K. Moses Jr. Dispersion of mobile robots: A study of memory-time trade-offs. In *ICDCN*, pages 1:1–1:10, 2018.
- 2 Evangelos Bampas, Leszek Gasieniec, Nicolas Hanusse, David Ilcinkas, Ralf Klasing, and Adrian Kosowski. Euler tour lock-in problem in the rotor-router model: I choose pointers and you choose port numbers. In *DISC*, pages 423–435, 2009.
- 3 L. Barriere, P. Flocchini, E. Mesa-Barrameda, and N. Santoro. Uniform scattering of autonomous mobile robots in a grid. In *IPDPS*, pages 1–8, 2009.
- 4 Reuven Cohen, Pierre Fraigniaud, David Ilcinkas, Amos Korman, and David Peleg. Label-guided graph exploration by a finite automaton. *ACM Trans. Algorithms*, 4(4):42:1–42:18, August 2008.
- 5 Andreas Cord-Landwehr, Bastian Degener, Matthias Fischer, Martina Hüllmann, Barbara Kempkes, Alexander Klaas, Peter Kling, Sven Kurras, Marcus Märten, Friedhelm Meyer auf der Heide, Christoph Raupach, Kamil Swierkot, Daniel Warner, Christoph Weddemann, and Daniel Wonisch. A new approach for analyzing convergence algorithms for mobile robots. In *ICALP*, pages 650–661, 2011.
- 6 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.
- 7 G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *J. Parallel Distrib. Comput.*, 7(2):279–301, October 1989.
- 8 Archak Das, Kaustav Bose, and Buddhadeb Sau. Memory optimal dispersion by anonymous mobile robots. In *CALDAM*, pages 426–439, 2021.
- 9 Dariusz Dereniowski, Yann Disser, Adrian Kosowski, Dominik Pajak, and Przemyslaw Uznański. Fast collaborative graph exploration. *Inf. Comput.*, 243(C):37–49, August 2015.
- 10 Anders Dessmark, Pierre Fraigniaud, Dariusz R. Kowalski, and Andrzej Pelc. Deterministic rendezvous in graphs. *Algorithmica*, 46(1):69–96, 2006. doi:10.1007/s00453-006-0074-2.
- 11 Yotam Elor and Alfred M. Bruckstein. Uniform multi-agent deployment on a ring. *Theor. Comput. Sci.*, 412(8-10):783–795, 2011.
- 12 Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro. *Distributed Computing by Oblivious Mobile Robots*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2012. doi:10.2200/S00440ED1V01Y201208DCT010.
- 13 Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro. *Distributed Computing by Mobile Entities*, volume 1 of *Theoretical Computer Science and General Issues*. Springer International Publishing, 2019.
- 14 Pierre Fraigniaud, Leszek Gasieniec, Dariusz R. Kowalski, and Andrzej Pelc. Collective tree exploration. *Networks*, 48(3):166–177, 2006.
- 15 Pierre Fraigniaud, David Ilcinkas, Guy Peer, Andrzej Pelc, and David Peleg. Graph exploration by a finite automaton. *Theor. Comput. Sci.*, 345(2-3):331–344, November 2005.
- 16 Dariusz R. Kowalski and Adam Malinowski. How to meet in anonymous network. *Theor. Comput. Sci.*, 399(1-2):141–156, 2008. doi:10.1016/j.tcs.2008.02.010.
- 17 Ajay D. Kshemkalyani and Faizan Ali. Fast graph exploration by a mobile robot. In *First IEEE International Conference on Artificial Intelligence and Knowledge Engineering, AIKE*, pages 115–118, 2018. doi:10.1109/AIKE.2018.00025.
- 18 Ajay D. Kshemkalyani and Faizan Ali. Efficient dispersion of mobile robots on graphs. In *ICDCN*, pages 218–227, 2019.
- 19 Ajay D. Kshemkalyani, Anisur Rahaman Molla, and Gokarna Sharma. Fast dispersion of mobile robots on arbitrary graphs. In *ALGOSENSORS*, pages 23–40, 2019.

- 20 Ajay D. Kshemkalyani, Anisur Rahaman Molla, and Gokarna Sharma. Dispersion of mobile robots in the global communication model. In *ICDCN*, pages 12:1–12:10, 2020.
- 21 Ajay D. Kshemkalyani, Anisur Rahaman Molla, and Gokarna Sharma. Dispersion of mobile robots on grids. In *WALCOM*, pages 183–197, 2020.
- 22 Ajay D. Kshemkalyani, Anisur Rahaman Molla, and Gokarna Sharma. Efficient dispersion of mobile robots on dynamic graphs. In *ICDCS*, pages 732–742, 2020.
- 23 Artur Menc, Dominik Pajak, and Przemyslaw Uznanski. Time and space optimality of rotor-router graph exploration. *Inf. Process. Lett.*, 127:17–20, 2017.
- 24 Anisur Rahaman Molla and William K. Moses Jr. Dispersion of mobile robots: The power of randomness. In *TAMC*, pages 481–500, 2019.
- 25 Anisur Rahaman Molla, Kaushik Mondal, and William K. Moses Jr. Efficient dispersion on an anonymous ring in the presence of weak byzantine robots. In *ALGOSENSORS*, pages 154–169, 2020.
- 26 Anisur Rahaman Molla, Kaushik Mondal, and William K. Moses Jr. Byzantine dispersion on graphs. In *IPDPS*, pages 1–10, 2021.
- 27 Debasish Pattanayak, Gokarna Sharma, and Partha Sarathi Mandal. Dispersion of mobile robots tolerating faults. In *ICDCN*, pages 133–138, 2021.
- 28 Pavan Poudel and Gokarna Sharma. Time-optimal uniform scattering in a grid. In *ICDCN*, pages 228–237, 2019.
- 29 Pavan Poudel and Gokarna Sharma. Fast uniform scattering on a grid for asynchronous oblivious robots. In *SSS*, pages 211–228, 2020.
- 30 Masahiro Shibata, Toshiya Mega, Fukuhito Ooshita, Hirotsugu Kakugawa, and Toshimitsu Masuzawa. Uniform deployment of mobile agents in asynchronous rings. In *PODC*, pages 415–424, 2016.
- 31 Takahiro Shintaku, Yuichi Sudo, Hirotsugu Kakugawa, and Toshimitsu Masuzawa. Efficient dispersion of mobile agents without global knowledge. In *SSS*, pages 280–294, 2020.

## A Appendix

**Proof of Lemma 8.** The series  $S_2$  is the sum of all the waits introduced by children  $a$  of a  $Y\_trunk$  node  $y$  and of  $M$ , that are of type  $Sl$ . Such a  $Sl$  child contributes delay up to  $4d_a\Delta + d_a$  ( $\leq 4d_y\Delta + 3d_y$  or  $\leq 4d_M\Delta + 3d_M$ , respectively) and then collapses and gets subsumed by the node  $b$  that has locked it. Thus  $Sl$  type children can occur at most  $k' - 1$  times in the lifetime of the execution. Note also that  $d_b \geq d_a$  as  $b$  to  $a$  is a decreasing path.

If all the  $Sl$  children were never involved in any meeting until now, then  $\sum d_a \leq k$  and the lemma follows. However we need to also analyze the case where a  $Sl$  node gets subsumed by another node  $b$ , and then the node  $b$  becomes a  $Sl$  node later. In this case, the robots subsumed from  $a$  may be double-counted in the size of  $b$  when  $b$  later becomes a type  $Sl$  node. This can happen at most  $k' - 1$  times.

Let  $\eta$  DFS components, including the  $Sl$  component, collapse and merge into one of them, and let the size (i.e., number of settled robots) of each component be  $d$ . We consider two extreme cases and show for each that the lemma holds.

1. Case 1: When components collapse and are collected, immediately afterwards (before the collected unsettled robots can settle) the master component becomes a  $Sl$ -type node, and the collapse and collection happen again. Again, immediately afterwards, the new master component becomes a type  $Sl$  node, and so on.
  - a. The first time,  $\eta$  components of size  $d$  each merge into one of size  $d$  in  $O(\eta d\Delta)$  time, leading to a total of  $\eta d$  robots in the master component.
  - b. The second time,  $\eta$  components of size  $d$  each merge into one of size  $d$  in  $O(\eta d\Delta)$  time, leading to a total of  $\eta^2 d$  robots in the new master component.



- c. The  $j$ -th time,  $\eta$  components of size  $d$  each merge into one of size  $d$  in  $O(\eta d \Delta)$  time, leading to a total of  $\eta^j d$  robots in the master component.  $\eta^j d$  is at most the maximum number of robots  $k$ . Solving  $k = \eta^j d$ ,  $j = \log_\eta \frac{k}{d}$ . Therefore the total delay introduced in series  $S2$  which is linearly proportional to  $\Delta$  times the sum of sizes of the type  $Sl$  components, is  $O(\eta \Delta dj)$ .

$$\text{Sum of delays is } O(\eta \Delta dj) = O(\eta \Delta d \log_\eta \frac{k}{d})$$

This maximum elapsed time is  $O(k\Delta)$ , considering both extreme cases (a)  $\eta d = O(1)$  and (b)  $\eta d = O(k)$ .

2. Case 2: When components collapse and are collected, the collected robots (almost) fully disperse after which the master component becomes a type  $Sl$  node, and the collapse and collection happen again. Again, the collected robots in the new master component (almost) fully disperse after which the (new) master component becomes a type  $Sl$  node and collapses and gets collected, and so on.
- a. The first time,  $\eta$  components of size  $d$  each merge and settle into one of size  $\eta d$  in  $O(\eta d \Delta)$  time, leading to a total of  $\eta d$  robots in the master component.
- b. The second time,  $\eta$  components of size  $\eta d$  each merge and settle into one of size  $\eta^2 d$  in  $O(\eta^2 d \Delta)$  time, leading to a total of  $\eta^2 d$  robots in the master component.
- c. The  $j$ -th time,  $\eta$  components of size  $\eta^{j-1} d$  each merge and settle into one of size  $\eta^j d$  in  $O(\eta^j d \Delta)$  time, leading to a total of  $\eta^j d$  robots in the master component.  $\eta^j d$  is at most the maximum number of robots  $k$ . Solving  $k = \eta^j d$ ,  $j = \log_\eta \frac{k}{d}$ . Therefore the total delay introduced in series  $S2$  which is linearly proportional to  $\Delta$  times the sum of sizes of the type  $Sl$  components, is

$$\begin{aligned} O(\Delta(\eta d + \eta^2 d + \eta^3 d + \dots + \eta^j d)) &= O(\Delta \eta d \frac{\eta^h - 1}{\eta - 1}) \\ &= O(\frac{\Delta \eta d}{\eta - 1} (\eta^{\log_\eta \frac{k}{d}} - 1)) \\ &= O(\frac{\Delta \eta d}{\eta - 1} (\frac{k}{d} - 1)) \\ &= O(k\Delta) \end{aligned}$$

The lemma follows. ◀



# Asynchronous Gathering in a Torus

**Sayaka Kamei**

Hiroshima University, Japan

**Anissa Lamani**

Strasbourg University, CNRS, ICUBE, France

**Fukuhito Ooshita**

Nara Institute of Science and Technology, Japan

**Sébastien Tixeuil**

Sorbonne University, CNRS, LIP6, France

**Koichi Wada**

Hosei University, Tokyo, Japan

---

## Abstract

We consider the gathering problem for asynchronous and oblivious robots that cannot communicate explicitly with each other but are endowed with visibility sensors that allow them to see the positions of the other robots.

Most investigations on the gathering problem on the discrete universe are done on ring shaped networks due to the number of symmetric configurations. We extend in this paper the study of the gathering problem on torus shaped networks assuming robots endowed with local weak multiplicity detection. That is, robots cannot make the difference between nodes occupied by only one robot from those occupied by more than one robot unless it is their current node. Consequently, solutions based on creating a single multiplicity node as a landmark for the gathering cannot be used. We present in this paper a deterministic algorithm that solves the gathering problem starting from any rigid configuration on an asymmetric unoriented torus shaped network.

**2012 ACM Subject Classification** Theory of computation → Distributed algorithms

**Keywords and phrases** Autonomous distributed systems, Robots gathering, Torus

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2021.9

**Related Version** *Full Version:* <https://arxiv.org/abs/2101.05421> [17]

**Funding** This work was partially funded by the ANR project ESTATE, ref. ANR-16-CE25-0009-03, the ANR project SAPPORO, ref. 2019-CE25-0005-1, JSPS KAKENHI No. 19K11828, 20H04140, 20K11685, and 21K11748, and by JST SICORP (Grant#JPMJSC1806).

## 1 Introduction

We consider autonomous robots [21] that are endowed with visibility sensors and motion actuators, yet are unable to communicate explicitly. They evolve in a discrete environment, i.e., their space is partitioned into a finite number of locations, conveniently represented by a graph, where the nodes represent the possible locations that a robot can be, and the edges denote the possibility for a robot to move from one location to another.

Those robots must collaborate to solve a collective task despite being limited to computing capabilities, inputs from the environment, etc. In particular, the robots we consider are anonymous, uniform, yet they can sense their environment and make decisions according to their own ego-centered view. In addition, they are oblivious, i.e., they do not remember their past actions. Robots operate in *cycles* that include three phases: *Look*, *Compute*, and *Move* (LCM for short). The Look phase takes a snapshot of the other robots' positions using a robot's visibility sensors. During the Compute phase, a robot computes a target destination



© Sayaka Kamei, Anissa Lamani, Fukuhito Ooshita, Sébastien Tixeuil, and Koichi Wada; licensed under Creative Commons License CC-BY 4.0

25th International Conference on Principles of Distributed Systems (OPODIS 2021).

Editors: Quentin Bramas, Vincent Gramoli, and Alessia Milani; Article No. 9; pp. 9:1–9:17

Leibniz International Proceedings in Informatics



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

based on its previous observation. The Move phase consists in moving toward the computed destination using motion actuators. Three execution models have been considered in the literature using LCM cycles, capturing the various degrees of synchrony between robots. According to current taxonomy [11], they are denoted as FSYNC, SSYNC, and ASYNC, from the stronger to the weaker. FSYNC stands for *fully synchronous*. In this model, all robots execute the LCM cycle synchronously and atomically. In the SSYNC (*semi-synchronous*) model, robots are asynchronously activated to perform cycles, yet at each activation, a robot executes one cycle atomically. With the weaker model, ASYNC (*asynchronous*), robots execute LCM in a completely independent manner. Of course, the ASYNC model is the most realistic.

In the context of robots evolving on graphs, the two benchmarking tasks are *exploration* [13] and *gathering* [4]. In this paper, we address the *gathering* problem, which requires that robots eventually all meet at a single node, not known beforehand, and terminate upon completion.

We focus on the case where the network is an *anonymous unoriented torus* (or simply *torus*, for short). The terms *anonymous* and *unoriented* mean that no robot has access to any kind of external information (*e.g.*, node identifiers, oracle, local edge labeling, etc.) allowing to identify nodes or to determine any (global or local) direction, such as North-South/East-West. Torus networks were investigated for the purpose of exploration by Devismes et al.[9].

## 1.1 Related Work

Mobile robot gathering on graphs was first considered for ring-shaped graphs. Klasing *et al.* [18], proposed gathering algorithms for rings with *global-weak* multiplicity detection. Global-weak multiplicity detection enables a robot to detect whether the number of robots on each node is one or more than one. However, the exact number of robots on a given node remains unknown if more than one robot is on the node. Then, Izumi *et al.* [14] provided a gathering algorithm for rings with *local-weak* multiplicity detection under the assumption that the initial configurations are non-symmetric and non-periodic, and that the number of robots is less than half the number of nodes. Local-weak multiplicity detection enables a robot to detect whether the number of robots on its *current* node is one or more than one. This condition was slightly relaxed by Kamei et al. [15]. D'Angelo *et al.* [6] proposed unified ring gathering algorithms for most of the solvable initial configurations, using local-weak multiplicity detection. Overall, for rings, relatively few open cases remain [1], as algorithm synthesis was demonstrated feasible [19].

The case of gathering in tree-shaped networks was investigated by D'Angelo *et al.* [7] and by Di Stefano et al.[20]. Hypercubes were the focus of Bose et al. [2]. Complete and complete bipartite graphs were outlined by Cicerone et al. [5], and regular bipartite by Guilbault et al. [12]. Finite grids were studied by D'Angelo et al. [7], Das et al. [8], and Castenow et al. [3], while infinite grids were considered by Di Stefano et al. [20], and by Durjoy et al. [10]. Results on grids and infinite grids do not naturally extend to tori. On the one hand, the proof arguments for impossibility results on the grid can be extended for the torus, since their indistinguishability criterium remains valid. So, if a torus admits an edge symmetry (the robot positions are mirrored over an axial symmetry traversing an edge), is periodic (a non-trivial translation leaves the robot positions unchanged), or admits a rotation whose center is not a robot, the gathering is impossible on a torus. On the other hand, both the finite and the infinite grid allow algorithmic tricks to be implemented. For example, the finite grid has three classes of nodes: corners (of degree 2), borders (of degree 3), and inner nodes (of degree 4), and those three classes permit the robots to obtain some sense of direction. By

contrast, the infinite grid makes a difference between two locations: the inner space (the set of nodes within the convex hull formed by the robot positions) and the outer space (the rest of the infinite grid), which also give some sense of direction. Now, every node in a torus has degree 4, and no notion of inner/outer space can be defined. To our knowledge, torus-shaped networks were never considered before for the gathering problem. The previous work by Devismes et al [9] only considers the exploration task.

## 1.2 Our Contribution

We consider the problem of gathering on torus-shaped networks. In more detail, for initial configurations that are *rigid* (i.e. where each robot has a unique view of the configuration), we propose a distributed algorithm that gathers all robots to a single node, not known beforehand. We only make use of local-weak multiplicity detection: robots may only know whether at least one other robot is currently hosted at their hosting node but cannot know the exact number and are also unable to retrieve multiplicity information from other nodes. Furthermore, robots have no common notion of North and no common notion of handedness. Finally, robots operate in the most general and realistic ASYNC execution model.

## 2 Model

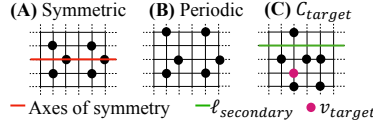
We consider a distributed system that consists of a collection of  $\mathcal{K} \geq 3$  robots evolving on a non-oriented and anonymous  $(\ell, L)$ -torus (or simply torus for short) of  $n$  nodes. Values  $\ell$  and  $L$  are two integers such that  $L < \ell$  and (definition borrowed from Devismes et al. [9]):

1.  $n = \ell \times L$ .
2. Let  $E$  be a finite set of edges. There exists an ordering  $v_1, \dots, v_n$  of the nodes of the torus such that  $\forall i \in \{0, \dots, n-1\}$ :
  - if  $i + \ell < n$ , then  $\{v_i, v_{(i+\ell)}\} \in E$ , else  $\{v_i, v_{(i+\ell) \bmod n}\} \in E$ .
  - if  $i + 1 \bmod \ell \neq 0$ , then  $\{v_i, v_{i+1}\} \in E$ , else  $\{v_i, v_{i-\ell+1}\} \in E$ .

Given the previous ordering  $v_0, \dots, v_{n-1}$ , for every  $j \in \{0, \dots, L-1\}$ , the sequence  $v_{j \times \ell}, v_{1+j \times \ell}, \dots, v_{\ell-1+j \times \ell}$  is called an  $\ell$ -ring. Similarly, for every  $k \in \{0, \dots, \ell-1\}$ , the sequence  $v_k, v_{k+\ell}, v_{k+2 \times \ell}, \dots, v_{k+(L-1) \times \ell}$  is called an  $L$ -ring.

On the torus operate  $\mathcal{K} \geq 3$  identical robots, *i.e.*, they all execute the same algorithm using no local parameters, and one cannot distinguish them using their appearance. In addition, they are oblivious, *i.e.*, they cannot remember the operations performed before. No direct communication is allowed between robots; however, we assume that each robot is endowed with visibility sensors that allow him to see the position of the other robots on the torus. Robots operate in cycles that comprise three phases: *Look*, *Compute* and *Move*. During the first phase (Look), each robot takes a snapshot to see the positions of the other robots on the torus. In the second phase (Compute), they decide to either stay idle or move. In the case they decide to move, a neighboring destination is computed. Finally, in the last phase (Move), they move to the computed destination (if any).

At each instant, a subset of robots is activated for the execution by an external entity called *scheduler*. We assume that the scheduler is fair, *i.e.*, all robots are activated infinitely many times. The model considered in this paper is the asynchronous model (ASYNC), where the time between Look, Compute, and Move phases is finite but unbounded. We however assume that the move phase is instantaneous, so that when a robot performs a look operation, it sees all robots on nodes and none on edges. Still, even with instant moves, each robot may move according to an outdated view, *i.e.*, the robot takes a snapshot to see the positions of the other robots, but when it decides to move, some other robots may have moved already.



■ **Figure 1** Instance of some defined configurations.

Let  $\ell_0, \ell_1, \dots, \ell_{L-1}$  be the sequence of  $\ell$ -rings and let  $v_{i,0}, v_{i,1}, \dots, v_{i,\ell-1}$  be the sequence of nodes on  $\ell_i$  (all operations on the indices are modulo  $\ell$  for the nodes and modulo  $L$  for the  $\ell$ -rings). By  $\#r_{i,j}(t)$  we denote the number of robots on  $v_{i,j}$  at time  $t$ . Node  $v_{i,j}$  is empty if  $\#r_{i,j}(t) = 0$ . Otherwise,  $v_{i,j}$  is occupied. In the case where  $\#r_{i,j}(t) = 1$ , we say that there is a single robot on  $v_{i,j}$ . By contrast, if  $\#r_{i,j}(t) \geq 2$ , we say that there is a multiplicity on  $v_{i,j}$ .

In this paper, we assume that robots have a local weak multiplicity detection *i.e.*, for any robot  $r$ , located at node  $u$ ,  $r$  can only detect a multiplicity on its current node  $u$  (local). Moreover,  $r$  cannot be aware of the exact number of robots part of the multiplicity (weak).

During the process, some robots move and occupy some nodes of the torus, and their positions form the configuration of the system at that time. Initially, we assume that each node hosts at most one robot, *i.e.*, the initial configuration contains no multiplicities.

For each robot  $r$ , only a degraded vision of occupied locations is available. So, the local vision  $d_{i,j}(t)$  of  $r$  about node  $v_{i,j}$  at time  $t$  is 1 if  $\#r_{i,j}(t) > 0$  and 0 otherwise.

For any  $i, j \geq 0$ , let  $\delta_{i,j}^+(t)$  denote the sequence  $\langle d_{i,j}(t), d_{i,j+1}(t), \dots, d_{i,j+\ell-1}(t) \rangle$ , and let  $\delta_{i,j}^-(t)$  denote the sequence  $\langle d_{i,j}(t), d_{i,j-1}(t), \dots, d_{i,j-(\ell-1)}(t) \rangle$ . Similarly, let  $\Delta_{i,j}^{+s}(t)$  be the sequence  $\langle \delta_{i,j}^s(t), \delta_{i+1,j}^s(t), \dots, \delta_{i+(L-1),j}^s(t) \rangle$  and  $\Delta_{i,j}^{-s}(t)$  to be the sequence  $\langle \delta_{i,j}^s(t), \delta_{i-1,j}^s(t), \dots, \delta_{i-(L-1),j}^s(t) \rangle$  with  $s \in \{+, -\}$ .

The view of a given robot  $r$  located on node  $v_{i,j}$  at time  $t$  is defined as the pair  $view_r(t) = (\mathcal{V}_{i,j}(t), m_j)$  where  $\mathcal{V}_{i,j}(t)$  consists of the four sequences  $\Delta_{i,j}^{++}, \Delta_{i,j}^{+-}, \Delta_{i,j}^{-+}, \Delta_{i,j}^{--}$  ordered in the lexicographical order and  $m_j = 1$  if  $v_{i,j}$  hosts a multiplicity and  $m_j = 0$  otherwise.

By  $view_r(t)(1)$ , we refer to  $\mathcal{V}_{i,j}(t)$  in  $view_r(t)$ . Given two robots  $r$  and  $r'$ , we say that  $r$  has a larger view than  $r'$  at time  $t$ , denoted  $view_r(t)(1) > view_{r'}(t)(1)$ , if  $view_r(t)$  is lexicographically larger than  $view_{r'}(t)$ . Similarly,  $r$  is said to have the largest view at time  $t$ , if for any robots  $r' \neq r$ , not located on the same node as  $r$ ,  $view_r(t)(1) > view_{r'}(t)(1)$  holds.

A configuration is said to be rigid at time  $t$ , if for any two robots  $r$  and  $r'$ , located on two different nodes of the torus,  $view_r(t)(1) \neq view_{r'}(t)(1)$  holds.

A configuration is said to be periodic at time  $t$  if there exist two integers  $i$  and  $j$  such that  $i \neq j$ ,  $i \not\equiv 0 \pmod{\ell}$ ,  $j \not\equiv 0 \pmod{L}$ , and for every robot  $r_{(x,w)}$  located on  $\ell_x$  at node  $v_{x,w}$ ,  $view_{r_{(x,w)}}(t)(1) = view_{r_{(x+i,w+j)}}(t)(1)$  (An example is given in Figure 1).

As defined by D'Angelo et al. [7], a configuration is said to be symmetric at time  $t$ , if the configuration is invariant after a reflection with respect to either a vertical or a horizontal axis. This axis is called the axis of symmetry (An example is given in Figure 1).

In this paper, we consider asymmetric  $(\ell, L)$ -torus, *i.e.*,  $\ell \neq L$ . We assume *w.l.o.g.* that  $L < \ell$ . In this case, we can differentiate two sides of the torus. We denote by  $nb_{\ell_i}(C)$  the number of occupied nodes on  $\ell$ -ring  $\ell_i$ , in configuration  $C$ . An  $\ell$ -ring  $\ell_i$  is said to be maximal in  $C$  if  $\forall j \in \{0, \dots, L-1\} \setminus \{i\}$ ,  $nb_{\ell_j}(C) \leq nb_{\ell_i}(C)$ .

Given a configuration  $C$  and two  $\ell$ -rings  $\ell_i$  and  $\ell_j$ . We say that  $\ell_j$  is adjacent to  $\ell_i$  if  $|i - j| = 1 \pmod{L}$  holds. Similarly, we say that  $\ell_j$  is neighbor of  $\ell_i$  in configuration  $C$  if  $nb_{\ell_j}(C) > 0$  and  $nb_{\ell_k}(C) = 0$  for any  $k \in \{i+1, i+2, \dots, j-1\}$  or  $k \in \{i-1, i-2, \dots, j+1\}$ . We also define  $dis(x_i, x_j)$  to be a function which returns the shortest distance, in terms of hops, between  $x_i$  and  $x_j$  where  $x_i$  and  $x_j$  are two nodes of the torus. We sometimes write  $x_i = r_i$  where  $r_i$  is a robot. In this case,  $x_i$  refers to the node that hosts  $r_i$ . Finally, we use

the notion of  $d$ .block to refer to a sequence of consecutive nodes in which there are occupied nodes each  $d$  hops (distance) with no other robot in between. The size of a  $d$ .block is the number of its occupied nodes.

Due to the lack of space, some details and proofs are omitted but can be found in [17].

### 3 Impossibility Results

This section presents impossibility results that motivate our settings.

Given a graph  $G = (V, E)$  and a function  $m : V \rightarrow \mathbb{N}$  associating the number of robots on a vertex  $v$  of  $V$  to  $v$ ,  $(G, m)$  is a configuration whenever  $\sum_{v \in V} m(v)$  is bounded and greater than zero. Let  $\phi$  be a permutation of  $G$ 's vertices that preserves its adjacency relation, so if  $(u, v) \in E$ , then  $(\phi(u), \phi(v)) \in E'$ , with  $\phi(G) = (V', E')$ . Note that  $\phi$  always exists as the identity permutation fits this definition. Similarly, given a configuration  $(G, m)$ , let  $\psi$  be a permutation of  $G$ 's vertices that preserves its adjacency relation and such that for every node  $v$  of  $V$ ,  $m(v) = m(\psi(v))$ . Again,  $\psi$  always exists as the identity permutation fits this definition. Given such a permutation  $\psi$ , the *cycle*  $C_\psi$  of order  $p$  that is generated by  $\psi$  is  $\{\psi^0, \psi^1 = \psi, \psi^2 = \psi \circ \psi, \dots, \psi^{p-1}\}$  such that  $\psi^p = \psi^0$ , where  $\psi^0$  is the identity. Note that  $C_\psi$  has order 1 if and only if  $\psi$  is the identity. Given a cycle  $C_\psi$ , the *orbit* of a vertex  $v$  of  $V$  is  $C_\psi(v) = \{\gamma(v) | \gamma \in C_\psi\}$ . Now, given a configuration  $(G = (V, E), m)$ ,  $\psi$  is *partitive* if  $C_\psi$  has order  $p > 1$ , and for every  $v \in V$ ,  $|C_\psi(v)| = p$ . That is,  $\psi$  is not reduced to the identity, and all nodes have the same orbit size. We now recall the Theorem of Di Stefano and Navarra for general topologies:

► **Theorem 1** ([20] Restated). *If a configuration  $(G, m)$  admits a partitive permutation  $\psi$ , then  $(G, m)$  cannot be gathered.*

We specialize the general theorem to our setting:

► **Corollary 1.** *If a torus configuration is invariant by a non-empty series of non-null translations, a reflection through an edge-axis, or a non-empty series of non-null rotations whose center does not hold a robot; it is not gatherable.*

Next, we show that two robots cannot gather on a torus, even in FSYNC.

► **Theorem 2.** *Starting from a configuration with two robots  $a$  and  $b$  on different vertices in a torus with at least two vertices, gathering cannot occur, even in FSYNC.*

Finally, we show by induction that without multiplicity detection, the gathering is impossible.

► **Theorem 3.** *Starting from any configuration with  $\mathcal{K} \geq 2$  robots with no multiplicity detection, gathering in a torus is impossible, even in SSYNC.*

### 4 Algorithm

When robots have only local weak multiplicity detection, multiplicities should be carefully created as the gathering becomes impossible from a configuration in which there are only two occupied nodes that both host a multiplicity. In ASYNC model, we need to be extra careful when it comes to robots with outdated views as they might create unwanted multiplicities (recall that when a robot moves the configuration might have changed as one or several robots might have moved once or many times).

Our strategy is to create a sense of direction on a torus to identify the gathering node and keep this node invariant, preventing the creation of unwanted multiplicities (if the configuration contains only two occupied nodes, one of these two nodes hosts for sure a single robot). For this purpose, robots proceed in two phases: first, they create the desired direction allowing them to identify a single node and then gather on the identified node. More precisely, let  $\mathcal{C}_{target}$  be the set of configurations such that  $C \in \mathcal{C}_{target}$  if the following properties are satisfied:  $C$  contains three  $\ell$ -rings  $\ell_{secondary}$ ,  $\ell_{max}$  and  $\ell_{target}$  such that:

- (1)  $\ell_{max}$  is the unique maximal  $\ell$ -ring in  $C$ ,
- (2)  $\ell_{secondary}$  and  $\ell_{target}$  are adjacent to  $\ell_{max}$ .
- (3)  $nb_{\ell_{secondary}}(C) = 0$ ,
- (4)  $\ell_{target}$  satisfies exactly one of the following conditions:
  - $nb_{\ell_{target}}(C) = 1$ . We refer to the occupied node on  $\ell_{target}$  by  $v_{target}$ .
  - $nb_{\ell_{target}}(C) = 2$  and  $\ell_{target}$  hosts a 2.block. We refer to the unique empty node in the 2.block by  $v_{target}$ .
  - $nb_{\ell_{target}}(C) = 3$  and  $\ell_{target}$  hosts a 1.block of size 3. By  $v_{target}$ , we refer to the occupied node in the middle of the 1.block.

From a configuration  $C \in \mathcal{C}_{target}$ , a direction can be identified: from  $v_{target}$  to  $\ell_{max}$ . The idea is to make all robots neither on  $\ell_{max}$  nor on  $\ell_{target}$  move to join  $v_{target}$  and then make the remaining robots gather on the node that is on  $\ell_{max}$  which is adjacent to  $v_{target}$ . To summarize, the proposed algorithm consists of two phases:

1. **Preparation Phase.** This phase starts from an arbitrary rigid configuration  $C_0$  in which each node hosts at most one robot. Its aim is to reach a configuration  $C \in \mathcal{C}_{target}$ .
2. **Gathering Phase.** Starting from a configuration  $C \in \mathcal{C}_{target}$ , the gathering node is identified, and all robots eventually move to join it *i.e.*, the gathering is achieved.

Let us refer by  $\mathcal{C}_{p_1}$  (respectively  $\mathcal{C}_{p_2}$ ) to the set of configurations that appear during the Preparation (respectively the Gathering) phase. Let  $C$  be the current configuration, robots execute Protocol 1. Observe that  $\mathcal{C}_{p_1} \cap \mathcal{C}_{p_2} = \emptyset$  and  $\mathcal{C}_{target} \subset \mathcal{C}_{p_2}$ .

■ **Protocol 1** Main protocol.

---

```

if  $C \in \mathcal{C}_{p_2}$  then
    Execute Gathering phase
else
    Execute Preparation phase
    
```

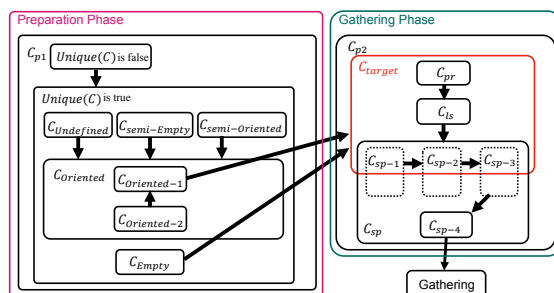
---

- To ease the description of our strategy, we define predicates on a given configuration  $C$ :
- **Unique**( $C$ ): There exists a unique  $i \in \{0, \dots, L-1\}$  such that  $\forall j \in \{0, \dots, L-1\} \setminus \{i\}$ ,  $nb_{\ell_j}(C) < nb_{\ell_i}(C)$ .
  - **Empty**( $C$ ):  $(C \in \mathcal{C}_{target}) \wedge (\forall i \in \{0, \dots, L-1\}, \text{ such that } \ell_i \neq \ell_{target} \text{ and } \ell_i \neq \ell_{max}, nb_{\ell_i}(C) = 0)$ .
  - **Partial**( $C$ ):  $(C \in \mathcal{C}_{target}) \wedge (\exists i \in \{0, \dots, L-1\}, \text{ such that } \ell_i \neq \ell_{target} \text{ and } \ell_i \neq \ell_{max}, nb_{\ell_i}(C) \neq 0)$ .

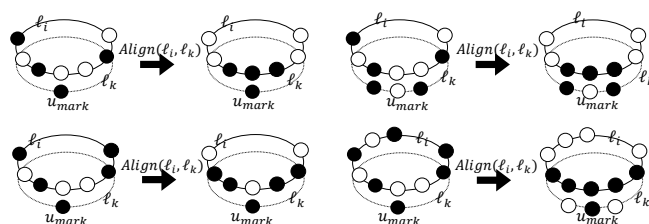
Given a configuration  $C$ , **Unique**( $C$ ) indicates that  $C$  contains a unique maximal  $\ell$ -ring. **Empty**( $C$ ) indicates that  $C \in \mathcal{C}_{target}$  and all the  $\ell$ -rings, except for  $\ell_{max}$  and  $\ell_{target}$ , are empty. By contrast, **Partial**( $C$ ) indicates that  $C \in \mathcal{C}_{target}$  and there exists at least one  $\ell$ -ring besides  $\ell_{max}$  and  $\ell_{target}$  that is occupied (hosts at least one occupied node).

In our algorithm, in several cases, robots in a single  $\ell$ -ring, say  $\ell_i$ , need to move and align themselves with respect to the positions of other robots which are on another  $\ell$ -ring, say  $\ell_k$ . To ease the description of the algorithm, we define a procedure referred to by **Align**( $\ell_i, \ell_k$ ) which makes the robots to perform such alignment *i.e.*, align robots on  $\ell_i$  with respect to robots positions on  $\ell_k$ . When the procedure is called in a configuration  $C$ , the following properties hold on both  $\ell_i$  and  $\ell_k$ :





■ **Figure 2** Transitions among all configurations.



■ **Figure 3** Some examples of  $\text{Align}(\ell_i, \ell_k)$ .

1.  $nb_{\ell_i}(C) = j$  with  $j \in \{2, \dots, 5\}$ , *i.e.*, there are at least two and at most five robots on  $\ell_i$ .
2.  $nb_{\ell_i}(C) > nb_{\ell_k}(C)$  holds, and either (1)  $nb_{\ell_k}(C) = 1$  or (2)  $nb_{\ell_k}(C) = 2$  and  $\ell_k$  contains a 2.block or (3)  $nb_{\ell_k}(C) = 3$  and  $\ell_k$  contains a 1.block of size 3.

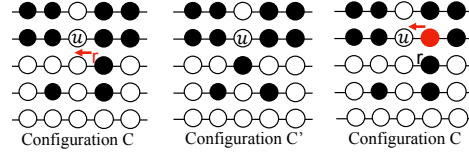
Let  $u_{mark}$  be the node on  $\ell_k$  that is: occupied if  $nb_{\ell_k}(C) = 1$ , empty in the 2.block if  $nb_{\ell_k}(C) = 2$ , occupied in the middle of the 1.block if  $nb_{\ell_k}(C) = 3$ . Node  $u_{mark}$  is used as a land mark to align robots on  $\ell_i$  (a detailed description can be found in [17]). To give a better idea on the purpose of procedure **Align**, some examples are given in Figure 3. The procedure makes sure that if a multiplicity is created on  $\ell_i$  then it is adjacent to  $u_{mark}$ . This allows the robots to keep track on multiplicities' positions and also make sure that the occupied nodes at a border of a 1.block on  $\ell_i$  host only a single robot.

Figure 2 presents an overview of our strategy showing all the transitions among the different defined configurations in sections 4.1-4.2.

## 4.1 Preparation Phase

Let  $C \in \mathcal{C}_{p1}$ . The purpose of this phase is to reach a configuration  $C' \in \mathcal{C}_{target}$  from  $C$  so that a direction is defined and the gathering node is identified. For this aim, robots first need to decrease the number of maximal  $\ell$ -rings to reach a configuration  $C''$  in which  $\text{Unique}(C'')$  is true. Then, from configuration  $C''$ , robots need to create both  $\ell_{target}$  and  $\ell_{secondary}$  to reach a configuration  $C' \in \mathcal{C}_{target}$ . To prevent the creation of unwanted multiplicities due to robots with outdated views, most of the configurations in this phase are kept rigid.

First, let us address the case in which  $\text{Unique}(C)$  is false ( $C$  contains at least two maximal  $\ell$ -rings). Robots need to decrease the number of maximal  $\ell$ -rings to reach a configuration  $C'$  in which  $\text{Unique}(C')$  holds. Two cases are possible depending on whether there is an empty node on a maximal  $\ell$ -ring: if a maximal  $\ell$ -ring hosts at least one empty node then, the idea is to fill one of these empty nodes on a single maximal  $\ell$ -rings. By contrast, if all the nodes of the maximal  $\ell$ -rings are occupied, the idea is to create a single multiplicity on one of the maximal  $\ell$ -rings to decrease their number gradually. Robots to move are chosen carefully



■ **Figure 4** On the left,  $r$  is suppose to move but by moving, it creates a symmetric configuration  $C'$  shown in the middle. The robot on target- $\ell$  on the same  $L$ -ring as  $r$  moves to  $u$ .

in both cases, so that the configuration remains rigid. This is important to prevent having robots with outdated views. In the following, we refer to a maximal  $\ell$ -ring by  $\ell_{\max}$ . Robots behavior in a configuration  $C$  in which  $\text{Unique}(C)$  holds is as follows:

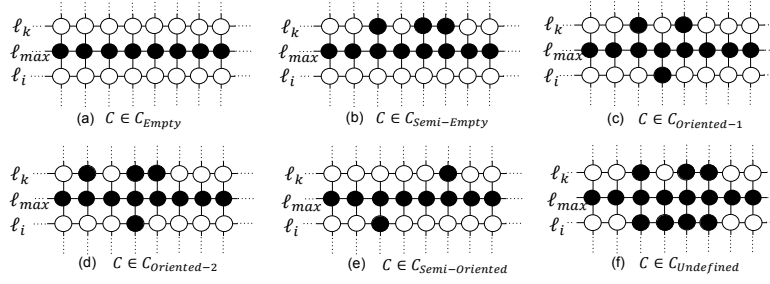
1. If  $nb_{\ell_{\max}}(C) = \ell$  (all the nodes of  $\ell_{\max}$  are occupied). Let  $R_{\max}(C)$  be the set of robots on a maximal  $\ell$ -ring. As  $C$  is rigid, all robots in  $R_{\max}(C)$  have a unique view. Let  $\ell_m$  be the maximal  $\ell$ -ring in  $C$  that hosts the robot with the maximal view in  $R_{\max}(C)$ . One robot  $r$  is elected on  $\ell_m$  to move. Its destination is one of its adjacent occupied nodes on  $\ell_m$ . Robot  $r$  is selected as follows: Let  $R_m(C) \subset R_{\max}(C)$  be the set of robots on  $\ell_m$  which by moving to one of their adjacent occupied node on  $\ell_m$ , the configuration reached remains rigid. Robot  $r$  is the robot in  $R_m(C)$  which has the biggest view ( $|R_m(C)| > 0$ ).
2. If  $nb_{\ell_{\max}}(C) < \ell$  (There is at least one empty node on  $\ell_{\max}$ ), the idea is to fill exactly one of the empty nodes on exactly one of the maximal  $\ell$ -ring. Let  $R(C)$  be the set of robots closest to an empty node on a maximal  $\ell$ -ring in  $C$ . Under some conditions, using the rigidity of  $C$ , one robot of  $R(C)$ , say  $r$ , is elected to move (the one with the largest view). Its destination is its adjacent empty node toward the closest empty node on a maximal  $\ell$ -ring, say  $u$ , taking the shortest path. Among robots in the set  $R(C)$ , the one to move is the one that does not create a symmetric configuration. If no such robot exists in  $R(C)$ , some extra steps are taken beforehand to ensure that the configuration remains rigid. We discuss the various cases:
  - If  $C$  contains exactly two occupied  $\ell$ -rings then,  $C$  contains only two maximal  $\ell$ -rings. Robot  $r$  (the one to move) is the robot with the maximal view in  $C$ . Its destination is its adjacent empty node on an empty  $\ell$ -ring (Note that this  $\ell$ -ring exists since  $L > 4$ ).
  - If  $C$  contains more than two occupied  $\ell$ -rings then: let  $r$  be the robot in  $R(C)$  with the largest view. By  $u$  and target- $\ell$  we refer to the closest empty node on a maximal  $\ell$ -ring to  $r$  and the  $\ell$ -ring including  $u$ . If by moving,  $r$  does not create a symmetric configuration, then  $r$  simply moves to its adjacent node toward  $u$  taking the shortest path. By contrast, if  $r$  creates a symmetric configuration by moving, then let  $C'$  be the configuration reached once  $r$  moves. Using configuration  $C'$  that each robot can compute without  $r$  moving, another robot  $r'$  in  $C$  is selected to move. We show later on that a symmetric configuration can only be reached when  $r$  either joins an empty node on the same  $L$ -ring as  $u$  for the first time or when it joins  $u$ . For the other cases, the configuration remains rigid. Hence, we only address the following two cases:
    - a. Robot  $r$  joins an empty node on the same  $L$ -ring as  $u$  for the first time in  $C'$ . In this case, in  $C$ , the robot that is on target- $\ell$  being on the same  $L$ -ring as  $r$  moves to  $u$  (refer to Figure 4).
    - b. Robot  $r$  joins  $u$  in  $C'$ . If in  $C'$  there are only two occupied  $\ell$ -rings. The robot with the largest view which does not create a symmetric configuration is elected to move. Its destination is its adjacent empty node on an empty  $\ell$ -ring. By contrast, if there are more than two occupied  $\ell$ -rings in  $C'$  then robots proceed as follows:

- If the axis of symmetry lies on the unique  $\ell_{max}$  in  $C'$  then, we are sure that there are two  $\ell$ -rings which are maximal in  $C$  and that are symmetric with respect to the unique maximal  $\ell$ -ring in  $C'$ . Let  $r$  be the robot located on a maximal  $\ell$ -ring which is not on the axis of symmetry in  $C'$ , that has the smallest view. Robot  $r$  is the one to move; its destination is its adjacent empty node on its  $\ell$ -ring.
- If the axis of symmetry is perpendicular to the unique maximal  $\ell$ -ring in  $C'$  then let  $T$  be the set of occupied  $\ell$ -rings in  $C$  without target- $\ell$ . If there is an  $\ell$ -ring in  $T$  which does not contain two 1.blocks separated by a single empty node on each side, then using the rigidity of  $C$ , a single robot on such an  $\ell$ -ring which is the closest to the biggest 1.block is elected to move. Its destination is the closest 1.block. If there no such  $\ell$ -ring in  $T$  (all  $\ell$ -rings contains two 1.blocks separated by a unique empty node), then using the rigidity of  $C$ , one robot being on an  $\ell$ -ring of  $T$  who has an empty node as a neighbor on its  $\ell$ -ring is elected to move. Its destination is its adjacent empty node on its current  $\ell$ -ring.

Note that we have only discussed the cases in which the reached configuration is either rigid or symmetric. This is because when  $r$  moves, it create neither a periodic nor an edge-edge symmetric configuration. This is mainly due to the fact that in  $C'$ , there is a unique maximal  $\ell$ -ring and  $C$  is assumed to be rigid.

We address now the case in which **Unique**( $C$ ) holds *i.e.*,  $C$  contains a unique maximal  $\ell$ -ring,  $\ell_{max}$ . To reach a configuration  $C' \in \mathcal{C}_{target}$ , robots need to move to build both  $\ell_{secondary}$  and  $\ell_{target}$  *i.e.*, one of the two adjacent  $\ell$ -rings to  $\ell_{max}$  needs to become empty while the other one needs to host either a single occupied node, a 2.block of size 2 or a 1.block of size 3. Let  $\ell_i$  and  $\ell_k$  be the two adjacent  $\ell$ -rings to  $\ell_{max}$ . Assume *w.l.o.g.* that  $nb_{\ell_i}(C) \leq nb_{\ell_k}(C)$ . To ease the description of this phase, we distinguish five main cases describing the possible states of  $\ell_i$  and  $\ell_k$ : (i) the case in which both  $\ell_i$  and  $\ell_k$  are empty ( $C \in \mathcal{C}_{Empty}$ ). The idea, in this case, is to elect a single robot to join either  $\ell_i$  or  $\ell_k$ . (ii) the case in which  $\ell_i$  is empty and  $\ell_k$  hosts more than one occupied node ( $C \in \mathcal{C}_{Semi-Empty}$ ). The idea is to make the robots on  $\ell_k$  gather in a single node. Note that in both cases (i) and (ii), a configuration  $C' \in \mathcal{C}_{target}$  is created. (iii) the case in which  $\ell_i$  hosts a single occupied node while  $\ell_k$  hosts at least two robots ( $C \in \mathcal{C}_{Oriented}$ ). The unique occupied node on  $\ell_i$  is used as a landmark to make robots on  $\ell_k$  move and create either a 2.block of size 2 or a 1.block of size 3 ( $C \in \mathcal{C}_{Oriented-2}$ ). Once such a block is created ( $C \in \mathcal{C}_{Oriented-1}$ ), it is easy to free  $\ell_i$  as the robots move to their adjacent node on  $\ell_{max}$  (since the configuration reached  $C' \in \mathcal{C}_{target}$ , the multiplicity created on  $\ell_{max}$  can be identified as it is adjacent to  $v_{target}$ ). (iv) the case in which both  $\ell_i$  and  $\ell_k$  host a unique occupied node ( $C \in \mathcal{C}_{Semi-Oriented}$ ). The idea is to add a single robot to either  $\ell_i$  or  $\ell_k$ . Finally, (v) the case in which both  $\ell_i$  and  $\ell_k$  host more than one robot ( $\mathcal{C}_{Undefined}$ ). The idea is to make robots elect either  $\ell_i$  or  $\ell_k$  and then make the robots on the elected ring gather on a single node. Both cases (iv) and (v) aim at reaching a configuration in  $\mathcal{C}_{Oriented}$ . More formally:

1. Set  $\mathcal{C}_{Empty}$ :  $C \in \mathcal{C}_{Empty}$  if  $nb_{\ell_i}(C) = nb_{\ell_k}(C) = 0$ .
2. Set  $\mathcal{C}_{Semi-Empty}$ :  $C \in \mathcal{C}_{Semi-Empty}$  if *w.l.o.g.*  $nb_{\ell_i}(C) = 0$  and  $nb_{\ell_k}(C) > 1$ .
3. Set  $\mathcal{C}_{Oriented}$ :  $C \in \mathcal{C}_{Oriented}$  if *w.l.o.g.*  $nb_{\ell_i}(C) = 1$  and  $nb_{\ell_k}(C) > 1$ . Set  $\mathcal{C}_{Oriented}$  includes:
  - a.  $\mathcal{C}_{Oriented-1}$ . In this case either (i)  $nb_{\ell_k}(C) = 3$  and  $\ell_k$  contains a 1.block of size 3 whose middle robot is on the same  $L$ -ring as the unique occupied node on  $\ell_i$ . (ii)  $nb_{\ell_k}(C) = 2$  and  $\ell_k$  contains a 2.block. Moreover, the unique empty node in the 2.block is on the same  $L$ -ring as the unique robot on  $\ell_i$ .



■ **Figure 5** Instance of configurations  $C$  when  $\text{Unique}(C)$  is true.

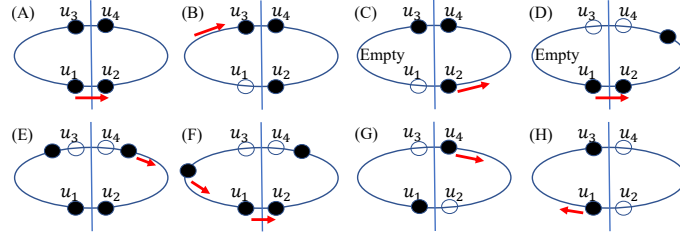
- b.  $\mathcal{C}_{\text{Oriented-2}}$ . Contains all the configuration in  $\mathcal{C}_{\text{oriented}}$  that are not in  $\mathcal{C}_{\text{Oriented-1}}$ .  
That is,  $\mathcal{C}_{\text{Oriented-2}} = \mathcal{C}_{\text{oriented}} - \mathcal{C}_{\text{Oriented-1}}$ .
- 4. Set  $\mathcal{C}_{\text{Semi-Oriented}}$ :  $C \in \mathcal{C}_{\text{Semi-Oriented}}$  if *w.l.o.g.*  $nb_{\ell_i}(C) = 1$  and  $nb_{\ell_k}(C) = 1$ .
- 5. Set  $\mathcal{C}_{\text{Undefined}}$ :  $C \in \mathcal{C}_{\text{Undefined}}$  if  $nb_{\ell_i}(C) > 1$  and  $nb_{\ell_k}(C) > 1$ .

Figure 5 presents instances of configurations in which there is a unique maximal  $\ell$ -ring.

The behavior of the robots in each set of configurations is as follows:

1.  $C \in \mathcal{C}_{\text{Empty}}$ . Let  $\ell_{n_i}$  and  $\ell_{n_k}$  be the two neighboring  $\ell$ -rings of  $\ell_{max}$  (one neighboring  $\ell$ -ring from each direction). In the case in which  $\ell_{n_i} = \ell_{n_k} = \ell_{max}$  ( $C$  contains a single occupied  $\ell$ -ring) then, using the rigidity of  $C$ , one robot from  $C$  is selected to move to its adjacent empty node outside its  $\ell$ -ring (the scheduler chooses the direction to take). Otherwise, let  $R_m$  be the set of robots which are the closest to either  $\ell_i$  or  $\ell_k$ . If  $|R_m| = 1$  then, the unique robot in  $R_m$ , referred to by  $r$ , is the one allowed to move. Assume *w.l.o.g.* that  $r$  is the closest to  $\ell_i$ . The destination of  $r$  is its adjacent empty node outside its current  $\ell$ -ring on the shortest empty path toward  $\ell_i$ . If  $r$  is the closest to both  $\ell_i$  and  $\ell_k$  then the scheduler chooses the direction to take (it moves either toward  $\ell_i$  or  $\ell_k$ ). In the case where  $|R_m| > 1$  ( $R_m$  contains more than one robot) then, by using the rigidity of  $C$ , one robot  $r$  is selected. Its behavior is the same as  $r$  in the case where  $|R_m| = 1$ .
2.  $C \in \mathcal{C}_{\text{Semi-Empty}}$ . Assume *w.l.o.g.*  $nb_{\ell_k}(C) > 1$  and  $nb_{\ell_i}(C) = 0$ . We consider two cases:
  - a.  $nb_{\ell_k}(C) > 3$  or  $nb_{\ell_k}(C) = 2$ . Recall that  $C \notin \mathcal{C}_{\text{target}}$ . Let  $\uparrow$  be the direction defined from  $\ell_{max}$  to  $\ell_k$  taking the shortest path and let  $\ell_n$  be the  $\ell$ -ring that is neighbor of  $\ell_i$ . Observe that  $\ell_n = \ell_k$  is possible (if only two  $\ell$ -rings are occupied in  $C$ ). Using the rigidity of configuration  $C$ , one robot from  $\ell_n$  is elected to move. Its destination is its adjacent node outside  $\ell_n$  and towards  $\ell_i$  with respect to the direction  $\uparrow$ .
  - b.  $nb_{\ell_k}(C) = 3$ . Again, recall that  $C \notin \mathcal{C}_{\text{target}}$ . The aim is to make the three robots form a single 1.block. To this end, if the configuration contains a single  $d$ .block of size 3 with  $d > 1$  then the robot in the middle of the  $d$ .block moves to its adjacent node on  $\ell_k$  (the scheduler chooses the direction to take). By contrast, if the configuration contains a single  $d$ .block of size 2 ( $d \geq 1$ ) then the robot not part of the  $d$ .block moves towards its adjacent empty node towards the  $d$ .block taking the shortest empty path.
3.  $C \in \mathcal{C}_{\text{Oriented}}$ . Let  $r_i$  be the single robot on  $\ell_i$ .
  - a.  $C \in \mathcal{C}_{\text{Oriented-1}}$ . If  $nb_{max}(C) > 4$  then the unique robot on  $\ell_i$  moves to its adjacent node on  $\ell_{max}$ . Otherwise, let  $u$  be the node on  $\ell_{max}$  adjacent to a robot on  $\ell_i$ .
    - If  $nb_{max}(C) = 3$  and the robots form a 1.block of size 3 whose middle robot is adjacent to  $u$  then the unique robot on  $\ell_i$  moves to its adjacent node on  $\ell_{max}$ . Otherwise, robots on  $\ell_{max}$  execute **Align**( $\ell_{max}, \ell_i$ ).
    - If  $nb_{max}(C) = 4$  and  $u$  is empty, then the unique robot on  $\ell_i$  moves to  $u$ . Otherwise ( $u$  is occupied), then let  $r$  be the robot on  $u$ .

- If  $r$  has an adjacent empty node on  $\ell_{max}$  then  $r$  moves to one of its adjacent nodes (the scheduler chooses the node to move to in case of symmetry).
  - If  $r$  does not have an adjacent empty node on  $\ell_{max}$ , then let  $r'$  be the robot on  $\ell_{max}$  which is adjacent to  $r$  and which does not have a neighboring robot on  $\ell_{max}$  at distance  $\lfloor \ell/2 \rfloor$ . Robot  $r'$  moves to its adjacent empty node on  $\ell_{max}$ .
- b.  $C \in \mathcal{C}_{Oriented-2}$ . If  $nb_{\ell_k}(C) = 2$  or  $nb_{\ell_k}(C) = 3$  then **Align**( $\ell_k, \ell_i$ ) is executed. Otherwise, if  $nb_{\ell_k}(C) > 3$  then,  $nb_{\ell_k}(C) - 2$  robots gather on the node  $u_k$  located on  $\ell_k$  and which is on the same  $L$ -ring as the unique occupied node on  $\ell_i$ . For this purpose, the robot on  $\ell_k$  which is the closest to  $u_k$  with the largest view is the one allowed to move. Its destination is its adjacent node on  $\ell_k$  toward  $u_k$ .
4.  $C \in \mathcal{C}_{Semi-Oriented}$ . Let  $\ell_{n_i}$  and  $\ell_{n_k}$  be the two neighboring  $\ell$ -rings of  $\ell_i$  and  $\ell_k$  respectively. First, if *w.l.o.g.*  $\ell_i = \ell_{n_k}$  ( $\ell_k = \ell_{n_i}$ ) then,  $C$  contains only 3 occupied  $\ell$ -rings  $\ell_i$ ,  $\ell_{max}$  and  $\ell_j$ . Using the rigidity of  $C$ , one robot from either  $\ell_{n_i}$  or  $\ell_{n_k}$  (not both) is selected to move. Its destination is its adjacent empty node outside its current  $\ell$ -ring in the opposite direction of  $\ell_{max}$ . Next, if  $\ell_i \neq \ell_{n_k}$  ( $\ell_k \neq \ell_{n_i}$ ) then, using the rigidity of  $C$ , a unique robot is selected to move from either  $\ell_{n_i}$  or  $\ell_{n_k}$  (not both). Its destination is its adjacent empty node outside its current  $\ell$ -ring toward  $\ell_i$  (respectively  $\ell_k$ ) if the robot was elected from  $\ell_{n_i}$  (respectively  $\ell_{n_k}$ ). If  $\ell_{n_i} = \ell_{n_k}$ , the scheduler chooses the direction to take.
5.  $C \in \mathcal{C}_{Undefined}$ . Depending on the number of robots on  $\ell_i$  and  $\ell_k$ , we consider two cases:
- a.  $nb_{\ell_i}(C) < nb_{\ell_k}(C)$ . The idea is to make robots on  $\ell_i$  gather on  $\ell_i$ . We define a configuration, denoted  $\Gamma(C)$ , built from  $C$  ignoring some  $\ell$ -rings that will be used to identify a single node on  $\ell_i$  on which all robots on  $\ell_i$  will gather. If there are at least four occupied  $\ell$ -rings in  $C$  then  $\Gamma(C)$  is the configuration built from  $C$  ignoring both  $\ell_i$  and  $\ell_k$ . By contrast, if there are only three occupied  $\ell$ -rings then  $\Gamma(C)$  is the configuration built from  $C$  ignoring only  $\ell_i$ . The following cases are possible:
- i. Configuration  $\Gamma(C)$  is rigid. Using the rigidity of  $\Gamma(C)$ , one node on  $\ell_i$ , say  $u$ , is elected as the gathering node. Robots on  $\ell_i$  move in turn to the elected node.
- ii. Configuration  $\Gamma(C)$  has exactly one axis of symmetry. The axis of symmetry of  $\Gamma(C)$  either intersects with  $\ell_i$  on a single node (edge-node symmetric), or on two nodes (node-node symmetric) or only on edges (edge-edge symmetric):
- $\Gamma(C)$  is node-edge symmetric: The node on  $\ell_i$  that is on the axis of symmetry of  $\Gamma(C)$  is the gathering node. Robots on  $\ell_i$  move in turn to join it.
  - $\Gamma(C)$  is node-node symmetric: Let  $u_1$  and  $u_2$  be the two nodes on  $\ell_i$  on which the axis of symmetry passes through. If both nodes are occupied, then using the rigidity of  $C$ , exactly one of the two nodes is elected. Assume *w.l.o.g.* that  $u_1$  is elected. Robots on  $u_1$  move to their adjacent node. If both  $u_1$  and  $u_2$  are empty then let  $R$  be the set of robots on  $\ell_i$  that are at the smallest distance from either  $u_1$  or  $u_2$ . If  $|R| = 1$  (Let  $r \in R$  and assume *w.l.o.g.* that  $r$  is the closest to  $u_1$ ) then,  $r$  moves on  $\ell_i$  toward  $u_1$  taking the shortest path. By contrast, if  $|R| > 1$  then using the rigidity of  $C$ , exactly one robot of  $R$  is elected to move. The elected robot moves on  $\ell_i$  toward the closest node among  $u_1$  and  $u_2$  taking the shortest path.
  - $\Gamma(C)$  is edge-edge symmetric: assume *w.l.o.g.* that  $\Gamma(C)$ 's axis of symmetry of passes through  $\ell_i$  on the two edges  $e_1 = (u_1, u_2)$  and  $e_2 = (u_3, u_4)$  with  $u_1$  and  $u_3$  being on the same side. Let  $U = \{u_j, j \in [1 - 4]\}$ . We consider the following cases:
    - For all  $u \in U$ ,  $u$  is occupied. Using the rigidity of  $C$ , a single node  $u \in U$  is elected. Robots on  $u$  move to their adjacent node  $u' \in U$  (refer to Figure 6, (A)).
    - Three nodes of  $U$  are occupied. Assume *w.l.o.g.* that  $u_1 \in U$  is the one empty. If there are robots on  $\ell_i$  which are located on the same side as  $u_1$  and  $u_3$  with respect to  $\Gamma(C)$ 's axis of symmetry then, the robots among these which are the



■ **Figure 6** Case in which  $\Gamma(C)$  is edge-edge symmetric.

closest to  $u_3$  move to their adjacent node on  $\ell_i$  toward  $u_3$  taking the shortest path (refer to Figure 6, (B)). By contrast, if there are no robots on  $\ell_i$  which are on the same side of  $u_1$  and  $u_3$  then, robots on  $u_2$  move to their adjacent node in the opposite direction of  $u_1$  (refer to Figure 6, (C)).

- Two nodes of  $U$  are occupied. First, assume *w.l.o.g.* that  $u_1$  and  $u_2$  are occupied (the case in which the two nodes are neighbors). If all robots on  $\ell_i$  are on the same side of the axis of symmetry (assume *w.l.o.g.* that they are at the same side as  $u_2$ ). Robots on  $u_1$  are the ones to move. Their destination is  $u_2$  (refer to Figure 6, (D)). By contrast, if there are robots on both sides of  $\Gamma(C)$ 's axis of symmetry then, let  $U'$  be the set of occupied nodes on  $\ell_i$  which are the farthest from the occupied node of  $U$  which is on the side (of the axis of symmetry). If there are two such nodes (one at each side), as  $C$  is rigid, the scheduler elects exactly one of these two nodes. Let us refer to the elected node by  $u$ . Robots on  $u$  are the ones to move. Their destination is their adjacent node on  $\ell_i$  towards the occupied node of  $U$  being on their side (refer to Figure 6, (E)). By contrast, if there is only one node in  $U'$  then, robots on the other side of the axis of symmetry are the ones to move to start from the robots that are the closest to the occupied node of  $U$  being on their side. Their destination is their adjacent  $\ell_i$  toward the occupied node of  $U$  on their side (refer to Figure 6, (F)). Finally, if there are no robots on both sides of the axis of symmetry, then using the rigidity of  $C$ , one occupied node of  $U$  is elected. Robots on the elected node are the ones to move. Their destination is their adjacent occupied node in  $U$ .

Next, assume *w.l.o.g.* that  $u_1$  and  $u_3$  are occupied (the case in which the two nodes of  $U$  are not neighbors but are at the same side of  $\Gamma(C)$ 's axis of symmetry). Robots on a node of  $U$  with the largest view are the ones to move. Their destination is their adjacent node in the opposite direction of a node of  $U$  (refer to Figure 6, (H)). Finally, assume *w.l.o.g.* that  $u_1$  and  $u_4$  are occupied (the case in which the two nodes of  $U$  are not neighbors and are in opposite sides of  $\Gamma(C)$  axis of symmetry). Robots on a node of  $U$  with the largest view are the ones to move. Their destination is their adjacent node on  $\ell_i$ , in the opposite direction of their adjacent node in  $U$  (refer to Figure 6, (G)).

- There is only one node of  $U$  that is occupied. Assume *w.l.o.g.* that  $u_1$  is occupied. If all robots on  $\ell_i$  are on the same side as  $u_1$  with respect to  $\Gamma(C)$ 's axis of symmetry then, the closest robot to  $u_1$  on  $\ell_i$  are the ones to move. Its destination is its adjacent node towards  $u_1$  taking the shortest path. By contrast, if all robots on  $\ell_i$  are in the opposite side of the axis of symmetry of  $u_1$  then robots on  $u_1$  are the ones to move. Their destination is  $u_2$ . Finally, if robots on  $\ell_i$  are on both

sides of the axis of symmetry then the closest robot to  $u_1$  being on the same side of  $\Gamma(C)$ 's axis of symmetry as  $u_1$  are the ones to move. Their destination is their adjacent node on  $\ell_i$  towards  $u_1$  taking the shortest path.

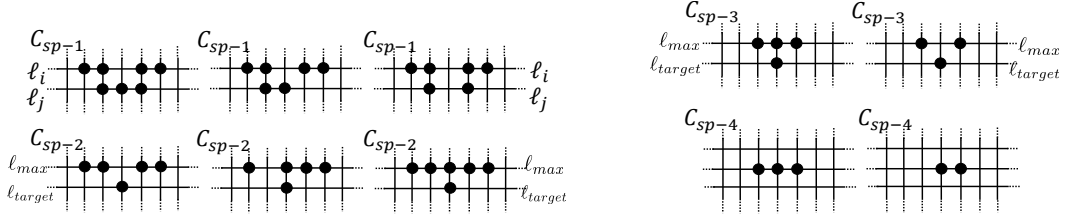
- All nodes of  $U$  are empty. Let  $d$  be the smallest distance between a node of  $u \in U$  and a robot on the same side of  $\Gamma(C)$ 's axis of symmetry as  $u$ . Let  $R$  be the set of robots at distance  $d$  from a node  $u \in U$ . If  $|R| = 1$  then the robot in  $R$  moves towards the closest node  $u \in U$ . By contrast, if  $|R| > 1$  then, using the rigidity of  $C$ , a unique robot in  $R$  is selected to move. Its destination is its adjacent node on  $\ell_i$  toward the closest node  $u \in U$ .
- iii. Configuration  $\Gamma(C)$  has more than one axis of symmetry. Using the rigidity of  $C$ , a single robot from  $\Gamma(C)$  is elected to move. Its destination is its adjacent empty node on its current  $\ell$ -ring. This reduces the number of axis of symmetries to either 1 or 0.
- b.  $nb_{\ell_i}(C) = nb_{\ell_k}(C)$ . The strategy is similar to the one used in the case in which  $nb_{\ell_i}(C) \neq nb_{\ell_k}(C)$ . That is, by using the state of configuration  $\Gamma(C)$ , robots on either  $\ell_i$  or  $\ell_k$  gather in a single node. The difference in this case is that the robots need to elect either  $\ell_i$  or  $\ell_k$ . The detailed description of this case can be found in [17].

► **Lemma 1.** *From any initial rigid configuration  $C_0 \in \mathcal{C}_{p_1}$ , a configuration  $C' \in \mathcal{C}_{target}$  which does not contain any robot with an outdated view, is eventually reached. Moreover, the unique maximal  $\ell$ -ring in  $C'$  hosts at most one multiplicity node. This node (if any) is adjacent to  $v_{target}$ .*

## 4.2 Gathering Phase

This phase starts from a configuration  $C \in \mathcal{C}_{target}$  in which a direction is defined in  $C$  (from  $\ell_{target}$  to  $\ell_{max}$ ). The idea is to make all robots that are neither on  $\ell_{target}$  nor  $\ell_{max}$  move to join  $v_{target}$ . Then, make some robots on  $\ell_{max}$  move to join  $v_{target}$  while the other align themselves with respect to  $v_{target}$  to finally gather all on the node of  $\ell_{max}$  adjacent to  $v_{target}$ . To ease the description of our algorithm, we define the following set of configurations:

1. Set  $\mathcal{C}_{sp}$  which includes the following four sub-sets:
  - a. SubSet  $\mathcal{C}_{sp-1}$ :  $C \in \mathcal{C}_{sp-1}$  if there are exactly two occupied  $\ell$ -rings in  $C$  denoted  $\ell_i$  and  $\ell_j$  respectively on which the following conditions hold: (1)  $\ell_i$  and  $\ell_j$  are adjacent. (2)  $nb_{\ell_j}(C) < nb_{\ell_i}(C)$  (3) either :
    - $nb_{\ell_i}(C) = 4$  and  $\ell_i$  contains two 1.blocks of size 2 being at distance 2 from each other. Let  $u$  be the unique node between the two 1.blocks on  $\ell_i$ .
    - $nb_{\ell_i}(C) = 3$  or  $5$  and  $\ell_i$  contains a 1.block of size  $nb_{\ell_i}(C)$ . Let  $u$  be the middle node of the 1.block of size  $nb_{\ell_i}(C)$ .
  - (4) Either  $nb_{\ell_j}(C) = 3$  and  $\ell_j$  contains a 1.block of size 3 whose middle node is adjacent to  $u$  or  $nb_{\ell_j}(C) = 2$  and  $\ell_j$  contains either a 2.block of size 2 whose middle node is adjacent to  $u$  or a 1.block of size 2 having one extremity adjacent to  $u$  (refer to Figure 7 for some examples).
- b. SubSet  $\mathcal{C}_{sp-2}$ :  $C \in \mathcal{C}_{sp-2}$  if  $C \in \mathcal{C}_{target}$  and  $nb_{\ell_{target}}(C) = 1$ . In addition either one of the following conditions are verified: (1)  $nb_{\ell_{max}}(C) = 4$  and on  $\ell_{max}$  there are two 1.blocks of size 2 being at distance 2 from each other. Let  $u$  be the unique node between the two 1.blocks then  $u$  is adjacent to  $v_{target}$ . (2)  $nb_{\ell_{max}} = 5$  and on  $\ell_{max}$  there is a 1.block of size 5 whose middle robot is adjacent to  $v_{target}$ . (3)  $nb_{\ell_{max}}(C) = 4$  and on  $\ell_{max}$  there is a 1.block of size 3 having a unique occupied node at distance 2. Let  $u$  be the unique empty node between the 1.block of size 3 and the 1.block of size 1. Then  $u$  is adjacent to  $v_{target}$  (refer to Figure 7).



■ **Figure 7** Set  $C_{sp}$ .

- c. SubSet  $C_{sp-3}$ :  $C \in C_{sp-3}$  if  $C \in C_{target}$ ,  $Empty(C)$  is true,  $nb_{l_{target}}(C) = 1$  and one of the two following conditions holds: (1)  $nb_{l_{max}}(C) = 3$  and  $l_{max}$  contains a 1.block of size 3 whose middle robot is adjacent to  $v_{target}$ . (2)  $nb_{l_{max}}(C) = 2$  and the two robots form a 2.block on  $l_{max}$ . Let  $u$  be the unique empty node between the two robots on  $l_{max}$ , then  $u$  is adjacent to  $v_{target}$  (refer to Figure 7).
- d. SubSet  $C_{sp-4}$ :  $C \in C_{sp-4}$  if there is a unique  $l$ -ring that is occupied and on this  $l$ -ring there are either two or three occupied nodes that form a 1.block (refer to Figure 7).
2. Set  $C_{pr}$ :  $C \in C_{pr}$  if  $C \in C_{target}$  and  $Partial(C)$  is true. That is,  $\exists i \in \{0, \dots, L-1\}$  such that  $l_i \neq l_{max}$  and  $l_i \neq l_{target}$  and  $nb_{l_i}(C) > 0$ . Note that we are sure that  $C \notin C_{sp}$ .
3. Set  $C_{ls}$ :  $C \in C_{ls}$  if  $C \in C_{target}$  and  $C \notin C_{sp}$  and  $Empty(C)$ . In other words, there are only two  $l$ -rings that are occupied:  $l_{max}$  and  $l_{target}$ .

We now present the behavior of robots during the gathering phase. If the current configuration  $C \in C_{target}$ , then we define  $\uparrow$  as the direction from  $l_{target}$  to  $l_{max}$  taking the shortest path. Observe that  $\uparrow$  can be computed by all robots and  $\uparrow$  is unique (recall that  $l_{max}$  is unique, and  $\forall C \in C_{target}$ ,  $nb_{l_{target}}(C) \neq nb_{l_{secondary}}(C)$ ). Using  $\uparrow$ , we define a total order on the  $l$ -rings of the torus such that  $l_i \leq l_j$  if  $l_i$  is not further from  $l_{target}$  than  $l_j$  with respect to  $\uparrow$ . Note that  $C_{p2} = C_{pr} \cup C_{ls} \cup C_{sp}$ . Let  $C$  be the current configuration, robots behavior for each defined set is as follows:

1.  $C \in C_{pr}$ . Let us refer by  $l_i$  to the  $l$ -ring that is adjacent to  $l_{target}$  such that  $l_i \neq l_{max}$ . Depending on the number of robots on  $l_i$ , two cases are possible:
  - a.  $nb_{l_i}(C) > 0$ . Let  $R_m$  be the set of robots on  $l_i$  that are the closest to  $v_{target}$ , if
    - i. there is an occupied node  $u_i$  on  $l_i$  that is adjacent to  $v_{target}$ , then robots on  $u_i$  are the ones to move. Their destination is  $v_{target}$ .
    - ii. there is no robot on  $l_i$  that is adjacent to  $v_{target}$  and  $nb_{l_i}(C) < \ell - 1$ , then robots in  $R_m$  are the ones to move. Their destination is their adjacent empty node on  $l_i$  on the empty path toward  $v_{target}$ .
    - iii. there is no robot on  $l_i$  adjacent to  $v_{target}$  and  $nb_{l_i}(C) = \ell - 1$ , then let  $R_{m'}$  be the set of robots that share a hole with  $u_i$ , where  $u_i$  is the node on  $l_i$  that is adjacent to  $v_{target}$ . Robots in  $R_{m'}$  are allowed to move only if they are not part of a multiplicity location. Their destination is the node towards  $u_i$  on the empty path.
  - b.  $nb_{l_i}(C) = 0$ . Let  $l_k$  be the closest neighboring  $l$ -ring to  $l_{target}$  with respect to  $\uparrow$ . Let  $R_m$  be the set of robots on  $l_k$  that are closest to  $v_{target}$ . Robots on  $R_m$  are the ones to move, their destination is the node outside  $l_k$  and toward  $l_{target}$  with respect to  $\uparrow$ .
2.  $C \in C_{ls}$ . Robots aims at reaching a configuration  $C' \in C_{sp}$ . If  $nb_{l_{max}}(C) \leq 5$ , robots on  $l_{max}$  execute  $Align(l_{max}, l_{target})$ . Otherwise, robots behave as follows: Let  $u_1, u_2, u_3, u_4$  and  $u_5$  be a sequence of five consecutive nodes on  $l_{max}$  such that  $u_3$  is adjacent to  $v_{target}$ . If  $u_3$  is occupied and has exactly one adjacent occupied node on  $l_{max}$  (assume *w.l.o.g.* that this node is  $u_2$ ) then the robot on  $u_2$  is the one to move. Its destination is  $u_3$ . By contrast, if  $u_3$  has either no adjacent occupied nodes on  $l_{max}$ , or two adjacent occupied



nodes on  $\ell_{max}$ , then robots on  $u_3$  move to  $v_{target}$ . Finally, if  $u_3$  is empty then let  $R$  be the set of robots that are closest to  $u_3$  on  $\ell_{max}$ . If  $|R| = 2$  then both robots move to their adjacent node on  $\ell_{max}$  toward  $u_3$ . By contrast, if  $|R| = 1$ , then first assume that the distance between the robot in  $R$  and  $u_3$  is  $d$ . If there is a robot  $r_m$  on  $\ell_{max}$  that shares a hole with  $u_3$  and at distance  $d+1$  from  $u_3$ , then  $r_m$  moves towards  $u_3$  taking the shortest path. If no such robot exists, the robot in  $R$  moves toward  $u_3$  taking the shortest path.

3.  $C \in \mathcal{C}_{sp}$ . We distinguish:
  - a.  $C \in \mathcal{C}_{sp-1}$ . If  $C \in \mathcal{C}_{target}$ , then the robots on  $\ell_{target}$  that are at the extremities of the 1.block or the 2.block move to their adjacent occupied node on  $\ell_{max}$ . By contrast, if  $C \notin \mathcal{C}_{target}$ , then the robot not on  $\ell_{max}$  that has two adjacent occupied nodes moves to its adjacent node on  $\ell_{max}$ .
  - b.  $C \in \mathcal{C}_{sp-2}$ . If there is a 1.block of size 3 on  $\ell_{max}$  then the robots that are in the middle of the 1.block of size 3 move to their adjacent occupied node that has one robot at distance 2. If  $\ell_{max}$  contains a 1.block of size 5 then the robots on  $\ell_{max}$  that are adjacent of the extremities of the 1.block move on  $\ell_{max}$  in the opposite direction of the extremities of the 1.block. Finally, if  $\ell_{max}$  contains two 1.blocks of size 2 then the robots that share a hole of size 1 move toward each other.
  - c.  $C \in \mathcal{C}_{sp-3}$ . Robots on  $v_{target}$  move to their adjacent node on  $\ell_{max}$  (note that  $v_{target}$  can be occupied by either a single robot or a multiplicity).
  - d.  $C \in \mathcal{C}_{sp-4}$ . If  $C$  contains a 1.block of size 3 then the robots at the extremities of the 1.block move to their adjacent occupied node. By contrast, if  $C$  contains a 1.block of size 2 then the robot that is not part of a multiplicity moves to its adjacent occupied node (it will be shown that one of the occupied nodes hosts only one robot).

We now state our main positive result.

► **Theorem 4.** *Assuming an  $(\ell, L)$ -torus in which  $L < \ell$  and  $L > 4$  and starting from an arbitrary rigid configuration, Protocol 1 solves the gathering problem for any  $K \geq 3$ .*

## 5 Concluding Remarks

We presented the first algorithm for gathering asynchronous oblivious mobile robots in a fully asynchronous model in a torus-shaped space graph. Our work raises several open questions:

1. What is the exact set of initial configurations that are gatherable? Our work considers initial rigid configurations only, and we know that periodic, edge-symmetric, and invariant through rotation (with no center robot) configurations make the problem impossible to solve. As in the case of the ring, special classes of non-rigid configuration may exist that are still gatherable.
2. The case of a square torus is intriguing: the robots would lose the ability to distinguish between the big side and the small side of the torus, so additional constraints are likely to hold if gathering remains feasible.
3. Following recent work by Kamei et al. [16] on the ring, it would be interesting to consider myopic (i.e. robot whose visibility radius is limited) yet luminous (i.e. robots that maintain a constant size state that can be communicated to other robots in the visibility range) robots in a torus.

---

## References

- 1 François Bonnet, Maria Potop-Butucaru, and Sébastien Tixeuil. Asynchronous gathering in rings with 4 robots. In *Ad-hoc, Mobile, and Wireless Networks - 15th International Conference, ADHOC-NOW 2016, Lille, France, July 4-6, 2016, Proceedings*, volume 9724 of *Lecture Notes in Computer Science*, pages 311–324. Springer, 2016. doi:10.1007/978-3-319-40509-4\_22.

- 2 Kaustav Bose, Manash Kumar Kundu, Ranendu Adhikary, and Buddhadeb Sau. Optimal gathering by asynchronous oblivious robots in hypercubes. In *Algorithms for Sensor Systems - 14th International Symposium on Algorithms and Experiments for Wireless Sensor Networks, ALGOSENSORS 2018, Helsinki, Finland, August 23-24, 2018, Revised Selected Papers*, volume 11410 of *Lecture Notes in Computer Science*, pages 102–117. Springer, 2018. doi:10.1007/978-3-030-14094-6\_7.
- 3 Jannik Castenow, Matthias Fischer, Jonas Harbig, Daniel Jung, and Friedhelm Meyer auf der Heide. Gathering anonymous, oblivious robots on a grid. *Theoretical Computer Science*, 815:289–309, 2020. doi:10.1016/j.tcs.2020.02.018.
- 4 Serafino Cicerone, Gabriele Di Stefano, and Alfredo Navarra. Asynchronous robots on graphs: Gathering. In *Distributed Computing by Mobile Entities, Current Research in Moving and Computing*, volume 11340 of *Lecture Notes in Computer Science*, pages 184–217. Springer, 2019. doi:10.1007/978-3-030-11072-7\_8.
- 5 Serafino Cicerone, Gabriele Di Stefano, and Alfredo Navarra. Gathering robots in graphs: The central role of synchronicity. *Theoretical Computer Science*, 849:99–120, 2021. doi:10.1016/j.tcs.2020.10.011.
- 6 Gianlorenzo D’Angelo, Alfredo Navarra, and Nicolas Nisse. A unified approach for gathering and exclusive searching on rings under weak assumptions. *Distributed Computing*, 30(1):17–48, 2017. doi:10.1007/s00446-016-0274-y.
- 7 Gianlorenzo D’Angelo, Gabriele Di Stefano, Ralf Klasing, and Alfredo Navarra. Gathering of robots on anonymous grids and trees without multiplicity detection. *Theoretical Computer Science*, 610:158–168, 2016. doi:10.1016/j.tcs.2014.06.045.
- 8 Shantanu Das, Nikos Giachoudis, Flaminia L. Luccio, and Euripides Markou. Gathering of robots in a grid with mobile faults. In *SOFSEM 2019: Theory and Practice of Computer Science - 45th International Conference on Current Trends in Theory and Practice of Computer Science, Nový Smokovec, Slovakia, January 27-30, 2019, Proceedings*, volume 11376 of *Lecture Notes in Computer Science*, pages 164–178. Springer, 2019. doi:10.1007/978-3-030-10801-4\_14.
- 9 Stéphane Devismes, Anissa Lamani, Franck Petit, and Sébastien Tixeuil. Optimal torus exploration by oblivious robots. *Computing*, 101(9):1241–1264, 2019. doi:10.1007/s00607-018-0595-8.
- 10 Durjoy Dutta, Tandrima Dey, and Sruti Gan Chaudhuri. Gathering multiple robots in a ring and an infinite grid. In *Distributed Computing and Internet Technology - 13th International Conference, ICDCIT 2017, Bhubaneswar, India, January 13-16, 2017, Proceedings*, volume 10109 of *Lecture Notes in Computer Science*, pages 15–26. Springer, 2017. doi:10.1007/978-3-319-50472-8\_2.
- 11 Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro, editors. *Distributed Computing by Mobile Entities, Current Research in Moving and Computing*, volume 11340 of *Lecture Notes in Computer Science*. Springer, 2019. doi:10.1007/978-3-030-11072-7.
- 12 Samuel Guilbault and Andrzej Pelc. Gathering asynchronous oblivious agents with local vision in regular bipartite graphs. *Theoretical Computer Science*, 509:86–96, 2013. doi:10.1016/j.tcs.2012.07.004.
- 13 David Ilcinkas. Oblivious robots on graphs: Exploration. In *Distributed Computing by Mobile Entities, Current Research in Moving and Computing*, volume 11340 of *Lecture Notes in Computer Science*, pages 218–233. Springer, 2019. doi:10.1007/978-3-030-11072-7\_9.
- 14 Tomoko Izumi, Taisuke Izumi, Sayaka Kamei, and Fukuhito Ooshita. Time-optimal gathering algorithm of mobile robots with local weak multiplicity detection in rings. *IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences*, 96-A(6):1072–1080, 2013. doi:10.1587/transfun.E96.A.1072.
- 15 Sayaka Kamei, Anissa Lamani, Fukuhito Ooshita, and Sébastien Tixeuil. Gathering an even number of robots in an odd ring without global multiplicity detection. In *Mathematical Foundations of Computer Science 2012 - 37th International Symposium, MFCS 2012, Bratislava, Slovakia, August 27-31, 2012. Proceedings*, volume 7464 of *Lecture Notes in Computer Science*, pages 542–553. Springer, 2012. doi:10.1007/978-3-642-32589-2\_48.

- 16 Sayaka Kamei, Anissa Lamani, Fukuhito Ooshita, Sébastien Tixeuil, and Koichi Wada. Gathering on rings for myopic asynchronous robots with lights. In *23rd International Conference on Principles of Distributed Systems, OPODIS 2019, December 17-19, 2019, Neuchâtel, Switzerland*, volume 153 of *LIPICs*, pages 27:1–27:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICs.OPODIS.2019.27.
- 17 Sayaka Kamei, Anissa Lamani, Fukuhito Ooshita, Sébastien Tixeuil, and Koichi Wada. Asynchronous gathering in a torus. *CoRR*, abs/2101.05421, 2021. arXiv:2101.05421.
- 18 Ralf Klasing, Adrian Kosowski, and Alfredo Navarra. Taking advantage of symmetries: Gathering of many asynchronous oblivious robots on a ring. *Theoretical Computer Science*, 411(34-36):3235–3246, 2010. doi:10.1016/j.tcs.2010.05.020.
- 19 Laure Millet, Maria Potop-Butucaru, Nathalie Sznajder, and Sébastien Tixeuil. On the synthesis of mobile robots algorithms: The case of ring gathering. In *Stabilization, Safety, and Security of Distributed Systems - 16th International Symposium, SSS 2014, Paderborn, Germany, September 28 - October 1, 2014. Proceedings*, volume 8756 of *Lecture Notes in Computer Science*, pages 237–251. Springer, 2014. doi:10.1007/978-3-319-11764-5\_17.
- 20 Gabriele Di Stefano and Alfredo Navarra. Optimal gathering of oblivious robots in anonymous graphs and its application on trees and rings. *Distributed Computing*, 30(2):75–86, 2017. doi:10.1007/s00446-016-0278-7.
- 21 Ichiro Suzuki and Masafumi Yamashita. Distributed anonymous mobile robots: Formation of geometric patterns. *SIAM Journal on Computing*, 28(4):1347–1363, 1999. doi:10.1137/S009753979628292X.



# Pattern Formation by Robots with Inaccurate Movements

Kaustav Bose<sup>1</sup> ✉ 

Advanced Computing and Microelectronics Unit, Indian Statistical Institute, Kolkata, India

Archak Das ✉ 

Department of Mathematics, Jadavpur University, Kolkata, India

Buddhadeb Sau ✉ 

Department of Mathematics, Jadavpur University, Kolkata, India

---

## Abstract

ARBITRARY PATTERN FORMATION is a fundamental problem in autonomous mobile robot systems. The problem asks to design a distributed algorithm that moves a team of autonomous, anonymous and identical mobile robots to form any arbitrary pattern  $F$  given as input. In this paper, we study the problem for robots whose movements can be inaccurate. Our movement model assumes errors in both direction and extent of the intended movement. Forming the given pattern exactly is not possible in this setting. So we require that the robots must form a configuration which is close to the given pattern  $F$ . We call this the APPROXIMATE ARBITRARY PATTERN FORMATION problem. With no agreement in coordinate system, the problem is unsolvable, even by fully synchronous robots, if the initial configuration 1) has rotational symmetry and there is no robot at the center of rotation or 2) has reflectional symmetry and there is no robot on the reflection axis. From all other initial configurations, we solve the problem by 1) oblivious, silent and semi-synchronous robots and 2) oblivious, asynchronous robots that can communicate using externally visible lights.

**2012 ACM Subject Classification** Theory of computation → Distributed algorithms; Theory of computation → Computational geometry; Computing methodologies → Distributed algorithms; Computing methodologies → Robotic planning

**Keywords and phrases** Distributed Algorithm, Mobile Robots, Movement Error, Approximate Arbitrary Pattern Formation, Look-Compute-Move, Minimum Enclosing Circle

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2021.10

**Related Version** *Full Version:* <https://arxiv.org/abs/2010.09667>

**Funding** *Archak Das:* supported by University Grants Commission, India.

**Acknowledgements** We would like to thank the anonymous reviewers for their comments and suggestions which helped us to improve the presentation of the paper.

## 1 Introduction

A robot swarm is a distributed system of autonomous mobile robots that collaboratively execute some complex tasks. Distributed coordination of robot swarms has attracted considerable research interest. Early investigations of these problems were experimental in nature with the main emphasis being on designing good heuristics. However, the last two decades have seen a series of theoretical studies on the computability and complexity issues related to distributed computing by robot swarms. These studies are aimed at providing provably correct algorithmic solutions to fundamental coordination problems. The robots are assumed to be *anonymous* (they have no unique identifiers that they can use in a

---

<sup>1</sup> This work was done when the author was at Jadavpur University, Kolkata, India.



## 10:2 Pattern Formation by Robots with Inaccurate Movements

computation), *homogeneous* (they execute the same distributed algorithm) and *identical* (they are indistinguishable by their appearance). The robots do not have access to any global coordinate system. They have either no memory or very little memory available to remember past observations and calculations. Also, they either have no means of direct communication or have some very weak communication mechanism (e.g., an externally visible light that can assume a small number of predefined colors). The model assumes robots with such weak features because the theoretical studies usually intend to find the minimum capabilities necessary for the robots to solve a given problem. The objective of this approach is to obtain a clear picture of the relationship between different features and capabilities of the robots (such as memory, communication, sensing, synchronization, agreement among local coordinate systems etc.) and their exact role in solvability of fundamental problems. Adopting such restrictive model also makes sense from a practical perspective since the individual units of robot swarms are low-cost generic robots with limited capabilities. Although certain assumptions, such as obliviousness (having no memory of past observations and calculations), may seem to be overly restrictive even for such weak robots, there are specific motivations for these assumptions. For example, the assumption of oblivious robots ensures self-stabilization. This is because any algorithm that works correctly for oblivious robots is inherently self-stabilizing as it tolerates errors that alter the local states of the robots. While the robots are assumed to be very weak with respect to memory, communication etc., certain aspects of the model are overly strong. In particular, the assumed mobility features of the robots are very strong. Two standard models regarding the movement of the robots are RIGID and NON-RIGID. In RIGID, if a robot  $x$  wants to go to any point  $y$ , then it can move to exactly that point in one step. This means that the robots are assumed to be able to execute error-free movements in any direction and by any amount. Certain studies also permit the robots to move along curved trajectories. The algorithms in this model rely on the accurate execution of the movements and are not robust to movement errors that real life robots are susceptible to. Furthermore, the error-free movements of the robots have surprising theoretical consequences as shown in the remarkable results obtained in [11]. A “positional encoding” technique was developed in [11] that allows a robot, that has very limited or no memory to store data, to implicitly store unbounded amount of information by encoding the data in the binary representation of its distance from another robot or some other object, e.g., the walls of the room inside which it is deployed. Exact movements allow the robots to preserve and update the data. This gives the robots remarkable computational power that allows them to solve complex problems which appear to be unsolvable by robots with limited or no memory, e.g., constructing a map of a complex art gallery by an oblivious robot. Obviously these techniques are impossible to implement in practice. Also, for problems that we expect to be unsolvable by real life robots with certain restrictions in memory, communication etc., it may become difficult or impossible to theoretically establish a hardness or impossibility result due to the strong model. The NON-RIGID model assumes that a robot may stop before reaching its intended destination. However,  $\exists$  a constant  $\delta > 0$  such that if the destination is at most  $\delta$  apart, the robot will reach it; otherwise, it will move towards the destination by at least  $\delta$ . Notice that in the NON-RIGID model, 1) the movement is still error-free if the destination is close enough, i.e., within  $\delta$ , and 2) there is no error whatsoever in the direction of the movement even if the destination is far away. In [1], it was shown that these two properties allow robots to implement positional encoding even in the NON-RIGID model. This motivates us to consider a new movement model allowing inaccurate movements.

We consider a movement model that assumes errors in both direction and extent of the intended movement. Also, the errors can occur no matter what the extent of the attempted movement is. In this model, we study the ARBITRARY PATTERN FORMATION problem.

ARBITRARY PATTERN FORMATION is a fundamental robot coordination problem that has been extensively studied in the literature [12, 14, 9, 8, 6, 13, 5, 10, 15, 2, 4]. The objective of the problem is to design a distributed algorithm that allows the robots to form any pattern  $F$  given as input. This problem is well-studied in the literature in the RIGID and NON-RIGID model. However, the techniques used in these algorithms are not readily portable in our setting. For example, in most of these algorithms, the minimum enclosing circle of the configuration plays an important role. The center of the minimum enclosing circle is set as the origin of the coordinate system with respect to which the pattern is to be formed. So the minimum enclosing circle is kept invariant throughout the algorithm. The robots inside the minimum enclosing circle move to form the part of the pattern inside the circle, without disturbing it. For the pattern points on the minimum enclosing circle, robots from the inside may have to move on to the circle. Also, the robots on the minimum enclosing circle, in order to reposition themselves in accordance with the pattern to be formed, will move along the circumference so that the minimum enclosing circle does not change. Notice that while moving along the circle, an error prone robot might skid off the circle. Also, when a robot from the inside attempts to move exactly on to the circle, it may move out of the circle due to the error in movement. In both cases, the minimum enclosing circle will change and the progress made by the algorithm will be lost. In fact, we face difficulty at a more fundamental level: exactly forming an arbitrary pattern is impossible by robots with inaccurate movements. Therefore, we consider a relaxed version of the problem called APPROXIMATE ARBITRARY PATTERN FORMATION where the robots are required to form an approximation of the input pattern  $F$ . We show that with no agreement in coordinate system, the problem is unsolvable, even by fully synchronous robots, if the initial configuration 1) has rotational symmetry and there is no robot at the center of rotation, or 2) has reflectional symmetry and there is no robot on the reflection axis. From all other initial configurations, we solve the problem in *OBLLOT* + *SSYNC* (the robots are oblivious, silent and semi-synchronous) and *FCOM* + *ASYNC* (the robots are oblivious, asynchronous and can communicate using externally visible lights).

Movement error was previously considered in [7], but in the context of the CONVERGENCE problem which requires the robots to converge towards a single point. The error model in [7] also considers errors in both direction and extent of the intended movement. However, there is some difference between the error model of [7] and the one introduced in this paper. In particular, the maximum possible error in direction is independent of the extent of the intended movement in [7]. In our model, the maximum possible error in both direction and extent, depend upon the extent of the intended movement. We believe that this is a reasonable assumption as the error is expected to be less if the destination of the intended movement is not far away.

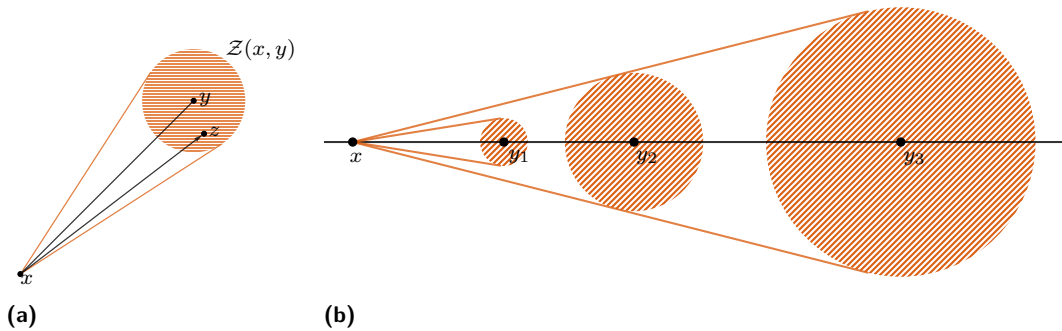
## 2 Robot Model

A set of  $n$  mobile computational entities, called *robots*, are initially positioned at distinct points in the plane. The robots are anonymous, identical, autonomous and homogeneous. The robots are modeled as dimensionless points in the plane. They do not have access to any global coordinate system. Each robot has its own local coordinate system centered at its current position. There is no consistency among the local coordinate systems of the robots except for a common unit of distance. We call this the *standard unit of distance*. Based on the memory and communication capabilities, we consider two standard models: *OBLLOT* and *FCOM*. In *OBLLOT*, the robots are *silent* (they have no means of communication) and

## 10:4 Pattern Formation by Robots with Inaccurate Movements

*oblivious* (they have no memory of past observations and computations). In  $\mathcal{FCOM}$ , each robot is equipped with a light which can assume a constant number of colors and is only visible to other robots. The lights serve as a weak communication mechanism. The robots, when active, operate according to the so-called LOOK-COMPUTE-MOVE cycles. In each cycle, a previously idle or inactive robot wakes up and executes the following steps. In the LOOK phase, the robot takes the snapshot of the positions (and their lights in case of  $\mathcal{FCOM}$ ) of the robots. Based on the perceived configuration, the robot performs computations according to a deterministic algorithm to decide a destination point (and a color in case of  $\mathcal{FCOM}$ ). Based on the outcome of the algorithm, the robot (sets its light to the computed color in case of  $\mathcal{FCOM}$ , and ) either remains stationary or attempts to move to the computed destination. Based on the activation and timing of the robots, there are three types of schedulers. In  $\mathcal{FSYNC}$  or fully synchronous, time can be logically divided into global rounds. In each round, all the robots are activated and they perform their actions at the same time.  $\mathcal{SSYNC}$  or semi-synchronous coincides with  $\mathcal{FSYNC}$ , with the only difference that not all robots are necessarily activated in each round. The most general model is  $\mathcal{ASYNC}$  or asynchronous where there are no synchronicity assumptions.

We now describe our movement model. There are known constants  $0 < \lambda < 1$ ,  $0 < \Delta < 1$ , such that if a robot at  $x$  attempts to move to  $y$ , then it will reach a point  $z$  where  $d(z, y) < \mu(x, y)d(x, y)$  where  $\mu(x, y) = \min\{\Delta, \lambda d(x, y)\}$ . Here  $d(x, y)$  denotes (the numerical value of) the distance between the points  $x$  and  $y$  measured in the standard unit of distance. The movement of the robot will be along the straight line joining  $x$  and  $z$ . We denote by  $\mathcal{Z}(x, y)$  the set of all points where a robot may reach if it attempts to move from  $x$  to  $y$ . So  $\mathcal{Z}(x, y)$  is the open disk  $\{z \in \mathbb{R}^2 \mid d(z, y) < \mu(x, y)d(x, y)\}$  (see Fig. 1a). We denote by  $error_d(x, y)$  and  $error_a(x, y)$  the supremums of the possible distance errors (i.e., the deviation from the intended amount of distance to be traveled) and angle errors (i.e., the angular deviation from the intended trajectory) respectively when a robot intends to travel from  $x$  to  $y$ . Notice that  $error_d(x, y)$  is equal to the radius of  $\mathcal{Z}(x, y)$  and  $error_a(x, y)$  is equal to the angle between  $line(x, y)$  and a tangent on  $\mathcal{Z}(x, y)$  passing through  $x$ . Hence,  $error_d(x, y) = \mu(x, y)d(x, y)$  and  $error_a(x, y) = \sin^{-1}\left(\frac{\mu(x, y)d(x, y)}{d(x, y)}\right) = \sin^{-1}(\mu(x, y))$ . Also notice that 1)  $error_d(x, y)$  increases with  $d(x, y)$ , and 2)  $error_a(x, y)$  increases with  $d(x, y)$  only up to a certain value, i.e.,  $\sin^{-1}(\Delta)$  and then remains constant (see Fig. 1b). So,  $error_a(x, y) \leq \sin^{-1}(\Delta)$ , for any  $x, y$ .

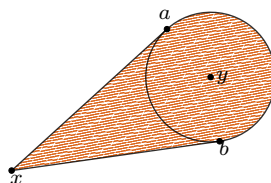


■ **Figure 1** a) If a robot attempts to move from  $x$  to  $y$ , then it will reach at some point  $z$  in the shaded region  $\mathcal{Z}(x, y)$ . b) If a robot attempts to move from  $x$  to  $y_i$ , then it will reach at some point in  $\mathcal{Z}(x, y_i)$  which is the shaded region around  $y_i$ . Observe that  $error_d(x, y_1) < error_d(x, y_2) < error_d(x, y_3)$ , but  $error_a(x, y_1) < error_a(x, y_2) = error_a(x, y_3)$ .



### 3 Definitions and Notations

We denote the configuration of robots by  $R = \{r_1, r_2, \dots, r_n\}$  where each  $r_i$  denotes a robot as well as the point in the plane where it is situated. The input pattern given to the robots will be denoted by  $F = \{f_1, f_2, \dots, f_n\}$  where each  $f_i$  is an element from  $\mathbb{R}^2$ . Given two points  $x$  and  $y$  in the Euclidean plane, let  $d(x, y)$  denote the distance between the points  $x$  and  $y$  measured in the standard unit of distance. We denote by  $line(x, y)$  the straight line passing through  $x$  and  $y$ . By  $seg(x, y)$  ( $\overline{seg}(x, y)$ ) we denote the line segment joining  $x$  and  $y$  excluding (resp. including) the end points. If  $\ell_1$  and  $\ell_2$  are two parallel lines, then  $\mathcal{S}(\ell_1, \ell_2)$  denotes the open region between these two lines. For any point  $c$  in the Euclidean plane and a length  $l$ ,  $C(c, l) = \{z \in \mathbb{R}^2 \mid d(c, z) = l\}$ ,  $B(c, l) = \{z \in \mathbb{R}^2 \mid d(c, z) < l\}$  and  $\overline{B}(c, l) = \{z \in \mathbb{R}^2 \mid d(c, z) \leq l\} = B(c, l) \cup C(c, l)$ . If  $C$  is a circle then  $encl(C)$  and  $\overline{encl}(C)$  respectively denote the open and closed region enclosed by  $C$ . Also,  $ext(C) = \mathbb{R}^2 \setminus \overline{encl}(C)$  and  $\overline{ext}(C) = \mathbb{R}^2 \setminus encl(C)$ . Hence,  $\overline{encl}(C) = encl(C) \cup C$  and  $\overline{ext}(C) = ext(C) \cup C$ . Let  $x, y$  be two points in the plane and  $d(x, y) > l$ . Suppose that the tangents from  $x$  to  $C(y, l)$  touches  $C(y, l)$  at  $a$  and  $b$ . The  $Cone(x, B(y, l))$  is the open region enclosed by  $B(y, l)$ ,  $\overline{seg}(x, a)$  and  $\overline{seg}(x, b)$ , as shown in Fig. 2. Also,  $\angle Cone(x, B(y, l)) = \angle axb$ . We denote by  $\mathcal{F}(x, y)$  the family of circles passing through  $x$  and  $y$ . The center of all the circles lie on the perpendicular bisector of  $\overline{seg}(x, y)$ . If  $C_1, C_2 \in \mathcal{F}(x, y)$  and  $c_1, c_2$  be their centers respectively, then  $(C_1, C_2)_{\mathcal{F}(x, y)}$  and  $[C_1, C_2]_{\mathcal{F}(x, y)}$  will denote respectively the family of circles  $\{C \in \mathcal{F}(x, y) \mid c \in seg(c_1, c_2)\}$ , where  $c$  is the center of  $C$  and  $\{C \in \mathcal{F}(x, y) \mid c \in \overline{seg}(c_1, c_2)\}$ , where  $c$  is the center of  $C$ .



■ **Figure 2**  $Cone(x, B(y, l))$  is defined as the shaded open region enclosed by  $B(y, l)$ ,  $\overline{seg}(x, a)$  and  $\overline{seg}(x, b)$ .

For a set  $P$  of points in the plane,  $C(P)$  and  $c(P)$  will respectively denote the minimum enclosing circle of  $P$  (i.e., the smallest circle  $C$  such that  $P \subset \overline{encl}(C)$ ) and its center. The smallest enclosing circle  $C(P)$  is unique and can be computed in linear time. For  $P$ , with  $2 \leq |P| \leq 3$ ,  $CC(P)$  denotes the circumcircle of  $P$  defined as the following. If  $P = \{p_1, p_2\}$ ,  $CC(P)$  is the circle having  $\overline{seg}(p_1, p_2)$  as the diameter and if  $P = \{p_1, p_2, p_3\}$  and the three points are not collinear,  $CC(P)$  is the unique circle passing through  $p_1, p_2$  and  $p_3$ .

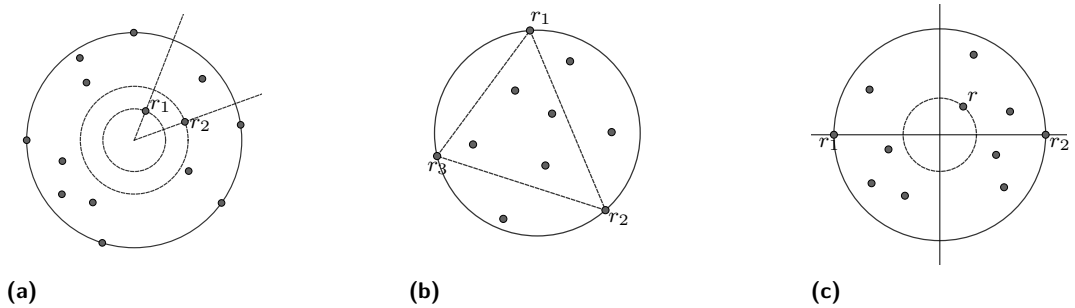
► **Property 1.** *If  $P' \subseteq P$  such that 1)  $P'$  consists of two points or  $P'$  consists of three points that form an acute angled triangle, and 2)  $P \subset \overline{encl}(CC(P'))$ , then  $C(P) = CC(P')$ . Conversely, for any  $P$ ,  $\exists P' \subseteq P$  so that 1)  $P'$  consists of two points or  $P'$  consists of three points that form an acute angled triangle and 2)  $C(P) = CC(P')$ .*

From Property 1 it follows that  $C(P)$  passes either through two points of  $P$  that are on the same diameter (antipodal points), or through at least three points so that some three of them form an acute angled or right angled triangle. A point  $p \in P$  is said to be *critical* if  $C(P) \neq C(P \setminus \{p\})$ . Note that  $p \in P$  is critical only if  $p \in C(P)$ .

► **Property 2.** *If  $|P \cap C(P)| \geq 4$  then there exists at least one point from  $P \cap C(P)$  which is not critical.*

Consider all concentric circles that are centered at  $c(P)$  and passes through at least one point of  $P$ . Let  $C_{\downarrow}^i(P)$  ( $C_{\uparrow}^i(P)$ ) denote the  $i$ th ( $i \geq 1$ ) of these circles so that  $C_{\downarrow}^{i+1}(P) \subset \text{encl}(C_{\downarrow}^i(P))$  (resp.  $C_{\uparrow}^i(P) \subset \text{encl}(C_{\uparrow}^{i+1}(P))$ ). We shall denote  $c(P)$  by  $C_{\uparrow}^0(P)$ . So we have  $C_{\downarrow}^1(P) = C(P)$  and if there is a point at  $c(P)$ , then  $C_{\uparrow}^1(P) = c(P) = C_{\uparrow}^0(P)$ . We say that a configuration of robots  $R$  is *symmetry safe* if one of the following three conditions hold (see Fig. 3).

1. i) there is some non-critical robot on  $C(R)$ , hence  $|R \cap C(R)| \geq 3$ , ii) there is no robot at  $c(R)$ , iii)  $|R \cap C_{\uparrow}^1(R)| = 1$  and  $|R \cap C_{\uparrow}^2(R)| = 1$ , iv) if  $R \cap C_{\uparrow}^1(R) = \{r_1\}$  and  $R \cap C_{\uparrow}^2(R) = \{r_2\}$ , then  $r_1, r_2, c(R)$  are not collinear.
2. i) all robots on  $C(R)$  are critical and  $R \cap C(R) = \{r_1, r_2, r_3\}$ , ii)  $\Delta r_1 r_2 r_3$  is scalene, i.e., all three sides have different lengths.
3. i) all robots on  $C(R)$  are critical and  $R \cap C(R) = \{r_1, r_2\}$ , ii)  $|R \cap C_{\uparrow}^1(R)| = 1$ , iii) if  $R \cap C_{\uparrow}^1(R) = \{r\}$ ,  $r \notin \text{line}(r_1, r_2) \cup \ell$ , where  $\ell$  is the line passing through  $c(R)$  and perpendicular to  $\text{line}(r_1, r_2)$ .



■ **Figure 3** Illustrations of symmetry safe configurations.

We shall say that a configuration  $R$  of robots has an *unbreakable symmetry* if one of the following is true: i)  $R$  has rotational symmetry with no robot at  $c(R)$ , ii)  $R$  has reflectional symmetry with respect to a line  $\ell$  with no robot on  $\ell$ .

#### 4 Approximate Arbitrary Pattern Formation

The ARBITRARY PATTERN FORMATION problem in its standard form is the following. Each robot of a team of  $n$  robots is given a pattern  $F$  as input which is a list of  $n$  distinct elements from  $\mathbb{R}^2$ . The given input  $F$  is exactly same for each robot. The problem asks for a distributed algorithm that guides the robots to a configuration that is similar to  $F$  with respect to translation, reflection, rotation and uniform scaling. We refer to this version of the problem as the EXACT ARBITRARY PATTERN FORMATION problem, highlighting the fact that the configuration of the robots is required to be exactly similar to the input pattern. However, it is not difficult to see that EXACT ARBITRARY PATTERN FORMATION is unsolvable in our model where the robot movements are inaccurate.

► **Theorem 1.** *EXACT ARBITRARY PATTERN FORMATION is unsolvable by robots with inaccurate movements.*

Therefore, we introduce a relaxed version of the problem called the APPROXIMATE ARBITRARY PATTERN FORMATION. Intuitively, we want the robots to form a pattern that is close to the given pattern, but may not be exactly similar to it. Formally, the robots are given as input a pattern  $F$  and a number  $0 < \epsilon < 1$ . The number  $\epsilon$  is small enough so

that the distance between no two pattern points is less than  $2\epsilon D$  where  $D$  is the diameter of  $C(F)$ . Given the input  $(F, \epsilon)$ , the problem requires the robots to form a configuration  $R = \{r_1, \dots, r_n\}$  such that there exists an embedding (subject to translation, reflection, rotation and uniform scaling) of the pattern  $F$  on the plane, say  $P = \{p_1, \dots, p_n\}$ , such that  $d(p_i, r_i) \leq \epsilon \bar{D}$  for all  $i = 1, \dots, n$ , where  $\bar{D}$  is the diameter of  $C(P)$ . In this case, we say that *the configuration  $R$  is  $\epsilon$ -close to the pattern  $F$* . Recall that the number  $\epsilon$  is such that the disks  $B(f_i, \epsilon D)$  are disjoint. Since  $P$  is similar to  $F$ , disks  $B(p_i, \epsilon \bar{D})$  are also disjoint. The problem requires that exactly one robot is placed inside each disk. Furthermore, the movements should be collisionless.

It is well known that ARBITRARY PATTERN FORMATION is unsolvable if the initial configuration has an unbreakable symmetry. It can be shown that this also holds for APPROXIMATE ARBITRARY PATTERN FORMATION (See Appendix A for proof).

► **Theorem 2.** *APPROXIMATE ARBITRARY PATTERN FORMATION is deterministically unsolvable, even with RIGID movements, if the initial configuration has unbreakable symmetries.*

## 5 The Algorithm for Semi-Synchronous Robots

In this section, we present an algorithm that solves APPROXIMATE ARBITRARY PATTERN FORMATION in *OBLLOT + SSYNC* from any initial configuration that does not have any unbreakable symmetries. The algorithm works in three phases which we shall describe in the next three subsections. For each phase, we shall first present the idea behind the approach and then give a brief description of the algorithm. More detailed description along with formal proofs can be found in the full version [3] of the paper.

### 5.1 Phase 1

#### Motive and Overview

The goal of Phase 1 is to create a configuration which is asymmetric and in which all robots on its minimum enclosing circle are critical. Phase 1 consists of three subphases, namely Subphase 1.1, Subphase 1.2 and Subphase 1.3. If the configuration is symmetric, our first step would be to get rid of the symmetry. Since the initial configuration cannot have any unbreakable symmetries, it will be possible to choose some unique robot from the configuration. We can remove the symmetry by appropriately moving this robot. This is done in Subphase 1.1. Once we have an asymmetric configuration, the next objective is to bring inside some non-critical robots from the minimum enclosing circle so that all the remaining robots on the minimum enclosing circle are critical. However, we have to make sure that these moves do not create new symmetries in the configuration. For this, we first make the configuration symmetry safe, i.e., have unique robots  $r_1$  and  $r_2$  respectively closest and second closest from the center of the minimum enclosing circle such that  $r_1$  and  $r_2$  are not on the same diameter. This is done in Subphase 1.2. After this, in Subphase 1.3, we start bringing inside the robots from the circumference. The movements of the robots should be such that  $r_1$  and  $r_2$  remain the unique closest and second closest robot from the center. This ensures that these movements do not create any symmetries. The two properties that we achieved in Phase 1, namely, having an asymmetric configuration and not having any non-critical robot on the minimum enclosing circle, will play crucial role in our approach and hence, will be preserved during the rest of the algorithm. This will be the case even if the target pattern  $F$  is symmetric or has non-critical robots on its minimum enclosing circle. This is not a problem as we are not required to exactly form the pattern  $F$ . Any pattern  $F$  can be approximated by a pattern that is asymmetric and has no non-critical points on its minimum enclosing circle.

### Brief Description of the Algorithm

We shall now briefly describe the algorithm. The algorithm is in Phase 1 if  $\neg u \wedge (\neg a \vee \neg c)$  holds, where  $a$  = “ $R$  is an asymmetric configuration”,  $u$  = “ $R$  has an unbreakable symmetry”,  $c$  = “all robots on  $C(R)$  are critical”. The objective is to create a configuration with  $a \wedge c$ . The algorithm is in Subphase 1.1 if  $\neg u \wedge \neg a$  holds, in Subphase 1.2 if  $a \wedge \neg s \wedge \neg c$  holds ( $s$  = “ $R$  is symmetry safe”) and in Subphase 1.3 if  $s \wedge \neg c$  holds.

First we describe Subphase 1.1. We have  $\neg u \wedge \neg a$ . Our objective is to create an asymmetric configuration, i.e., have  $a$ . As mentioned earlier, we will remove the symmetry by moving exactly one robot of the configuration, while all other robots will remain stationary. The fact that we have  $\neg u$ , will allow us to select one such robot from the configuration. To describe the algorithm, we have to consider the following four cases. Case 1 consists of the configurations in Subphase 1.1 where there is a robot at  $c(R)$ . Now consider the cases where there is no robot at  $c(R)$ . Notice that in this case,  $R$  cannot have a rotational symmetry because  $\neg u$  holds. So  $R$  has a reflectional symmetry with respect to a unique line  $\ell$  as reflectional symmetry with respect to two different lines imply rotational symmetry. Since  $\neg u$  holds, there are robots on  $\ell$ . If there is a non-critical robot on  $\ell$  then we call it Case 2. For the remaining cases where there is no non-critical robot on  $\ell$ , we call it Case 3 if there are more than 2 robots on  $C(R)$  and Case 4 if there are exactly 2 robots on  $C(R)$ .

In Case 1, we have a robot  $r$  at  $x = c(R)$ . In this case,  $r$  will move away from the center and all other robots will remain static. The destination  $y$  chosen by the robot  $r$  should satisfy the following conditions: (1)  $\mathcal{Z}(x, y) \subset \text{encl}(C_{\uparrow}^2(R)) \setminus \{c(R)\}$ , (2)  $\mathcal{Z}(x, y) \cap \ell = \emptyset$  for any reflection axis  $\ell$  of  $R \setminus \{r\}$ . It is easy to see that such an  $y$  exists. Furthermore,  $r$  can easily compute such an  $y$ .

In Case 2, there is no robot at  $c(R)$ ,  $R$  has reflectional symmetry with respect to a unique line  $\ell$  and there is at least one non-critical robot on  $\ell$ . If there are more than one non-critical robots on  $\ell$ , we can single out one of them using the concept of *view* of a robot (see Appendix A for details). In particular, all robots on  $\ell$  will have distinct views (because otherwise  $R$  will have rotational symmetry) and hence we have a unique non-critical robot  $r$  with minimum view. Only  $r$  will move in this case. Suppose that  $r$  is at point  $x$ . The destination  $y$  chosen by  $r$  should satisfy the following conditions: (1) if  $x \in C_{\uparrow}^i(R)$ , then  $\text{Cone}(x, \mathcal{Z}(x, y)) \subset \text{encl}(C_{\uparrow}^i(R)) \setminus \overline{\text{encl}(C_{\uparrow}^{i-1}(R))}$ , (2)  $\mathcal{Z}(x, y) \cap \ell' = \emptyset$  for any reflection axis  $\ell'$  of  $R \setminus \{r\}$ . Such points clearly exist and  $r$  can easily compute one.

In Case 3, we have no robot at  $c(R)$ ,  $R$  has reflectional symmetry with respect to a unique line  $\ell$ , there is no non-critical robot on  $\ell$  and  $C(R)$  has at least 3 robots on it. In this case, it can be shown that there is exactly one robot on  $\ell$  and it is on  $C(R)$ . Call this robot  $r$ . Let  $x$  denote its position. Let  $r_1, r_2$  be the two robots (specular with respect to  $\ell$ ) on  $C(R)$  such that  $\angle rc(R)r_1 = \angle rc(R)r_2 = \max\{\angle rc(R)r'' \mid r'' \in R \cap C(R)\}$ . It can be shown that  $\frac{\pi}{2} < \angle rc(R)r_1 = \angle rc(R)r_2 < \pi$ . Only  $r$  will move in this case and the rest will remain static. Here the robot will move outside of the current minimum enclosing circle. The chosen destination  $y$  should satisfy the following conditions: (1)  $\mathcal{Z}(x, y) \cap \ell = \emptyset$ , (2)  $\text{Cone}(x, \mathcal{Z}(x, y)) \subset \text{ext}(C(R)) \cap \text{encl}(C') \cap \mathcal{H}$  where  $C'$  is the largest circle from  $\{C \in \mathcal{F}(r_1, r_2) \mid R \subset \overline{\text{encl}(C)}\}$  and  $\mathcal{H}$  is the open half-plane delimited by  $\text{line}(r_1, r_2)$  that contains  $x$ , (3)  $\mathcal{Z}(x, y) \cap C_i = \emptyset$ , where  $C_i = C(r_i, d(r_1, r_2))$ ,  $i = 1, 2$ , (4)  $\mathcal{Z}(x, y) \subset \mathcal{S}(L_1, L_2)$ , where  $L_i$  is the line parallel to  $\ell$  and passing through  $r_i$ ,  $i = 1, 2$ . Again, it is straightforward to see that such an  $y$  should exist and  $r$  can easily compute one.

In Case 4, we have no robot at  $c(R)$ ,  $R$  has reflectional symmetry with respect to a unique line  $\ell$ , there is no non-critical robot on  $\ell$  and  $C(R)$  has exactly 2 robots on it. In this case, it can be shown that there is no robot on  $\ell \cap \text{encl}(C(R))$  and there are two

antipodal robots on  $\ell$ , say  $r$  and  $r'$ . Also, the views of  $r$  and  $r'$  must be different. So let  $r$  be the robot with minimum view. Only  $r$  will move in this case. Let  $\ell'$  be the line perpendicular to  $\ell$  and passing through  $r$ . For each  $r'' \in R \setminus \{r, r'\}$ , consider the line passing through  $r''$  and perpendicular to  $seg(r'', r')$ . Consider the points of intersection of these lines with  $\ell'$ . Let  $P_1, P_2$  (specular with respect to  $\ell$ ) be the two of these points that are closest to  $\ell$ . Let  $L_1, L_2$  be the lines parallel to  $\ell$  and passing through  $P_1, P_2$  respectively. Assuming that  $r$  is at point  $x$ , the destination  $y$  chosen by  $r$  should satisfy the following conditions: (1)  $Cone(x, \mathcal{Z}(x, y)) \subset ext(C(R))$ , (2)  $\mathcal{Z}(x, y) \cap \ell = \emptyset$ , (3)  $\mathcal{Z}(x, y) \subset \mathcal{S}(L_1, L_2)$ , (4)  $\mathcal{Z}(x, y) \cap C(c, d(c, r')) = \emptyset$ , where  $c = c(R \setminus \{r, r'\})$ .

It can be shown that the movements described for Subphase 1.1 will lead to an asymmetric configuration. The algorithm will be in Subphase 1.2 if  $\mathbf{a} \wedge \neg \mathbf{s} \wedge \neg \mathbf{c}$  holds. Then our goal would be to make the configuration symmetry safe. This can be easily done. When  $\mathbf{s} \wedge \neg \mathbf{c}$  holds, we are in Subphase 1.3. Then our objective would be to have  $\mathbf{a} \wedge \mathbf{c}$ . As the configuration is asymmetric (as  $\mathbf{s} \implies \mathbf{a}$ ), there is a robot with minimum view among all the non-critical robots lying on  $C(R)$ . This robot will move inside. Continuing in this manner, non-critical robots on  $C(R)$  will sequentially move inside until we obtain  $\mathbf{a} \wedge \mathbf{c}$ .

## 5.2 Phase 2

### Motive and Overview

Phase 1 was a preprocessing step where a configuration was prepared in which there is no symmetry and all robots on the minimum enclosing circle are critical. Actual formation of the pattern will be done in two steps, in Phase 2 and Phase 3. In Phase 2, the robots on the minimum enclosing circle will reposition themselves according to the target pattern and then in Phase 3, the robots inside the minimum enclosing circle will move to complete the pattern. The standard approach to solve the ARBITRARY PATTERN FORMATION problem, however, is exactly the opposite. Usually, the part of the pattern inside the minimum enclosing circle is first formed and then the pattern points on the minimum enclosing circle are occupied by robots. In this approach, the minimum enclosing circle is kept invariant throughout the algorithm. Keeping the minimum enclosing circle fixed is important because it helps to fix the coordinate system with respect to which the pattern is formed. During the second step, a robot on the minimum enclosing circle may have to move to another point on the circle. In order to keep the minimum enclosing circle unchanged, it has to move exactly along the circumference. However, it is not possible to execute such movement in our model. An error in movement in this step will change the minimum enclosing circle and the progress made by the algorithm will be lost. Placing the robots at the correct positions on the minimum enclosing circle is a difficult issue in our model. In fact, it can be proved that it is impossible to deterministically obtain a configuration with  $\geq 4$  robots on the minimum enclosing circle if the initial configuration has  $< 4$  robots on the minimum enclosing circle. For this reason, we shall work with 2 or 3 (critical) robots on the minimum enclosing circle as obtained from Phase 1 (or may be from the beginning). So in Phase 2, we start with an asymmetric configuration where all robots on the minimum enclosing circle are critical. The objective of this phase is to move these critical robots so that their relative positions on the minimum enclosing circle is consistent with the target pattern. For this, we shall choose a set of two or three pattern points from the minimum enclosing circle of the target pattern. We shall call this set the bounding structure of the target pattern. Essentially, the objective of Phase 2 is to approximate this structure by the critical robots.

### The Bounding Structure

If Algorithm 1 is applied on the target pattern  $F$ , then we obtain a set  $B_F \subseteq C(F) \cap F$  of pattern points such that  $B_F$  is a minimal set of points of  $C(F) \cap F$  such that  $CC(B_F) = C(F)$ . By minimal set we mean that no proper subset of  $B_F$  has this property. By Property 1,  $B_F$  either consists of two antipodal points or three points that form an acute angled triangle. We call  $B_F$  the *bounding structure* of  $F$  (see Fig. 4a). Recall that each robot computes the same bounding structure since the input  $F = \{f_1, f_2, \dots, f_n\}$  is same for all robots. We say that the bounding structure of  $F$  is formed by the robots if one of the following holds.

1.  $B_F$  has exactly two points,  $C(R)$  has exactly two robots on it and  $R$  is symmetry safe.
2.  $B_F$  has exactly three points and  $C(R)$  has exactly three robots on it (see Fig. 4). Let  $B_F = \{f_{i_1}, f_{i_2}, f_{i_3}\}$  and  $C(R) \cap R = \{r_1, r_2, r_3\}$ .  $R$  is symmetry safe (i.e.  $\Delta r_1 r_2 r_3$  is scalene) and furthermore, if  $seg(r_1, r_2)$  is the largest side of the triangle formed by  $r_1, r_2, r_3$  and  $seg(f_{i_1}, f_{i_2})$  is a largest side of the triangle formed by  $f_{i_1}, f_{i_2}, f_{i_3}$ , then  $\exists$  an embedding  $f_i \mapsto P_i$  of  $F$  on the plane identifying  $seg(f_{i_1}, f_{i_2})$  with  $seg(r_1, r_2)$  so that i)  $r_3 \in B(P_{i_3}, \epsilon D)$ , ( $D = \text{diameter of } C(P_1, \dots, P_n)$ ) and ii)  $B(P_i, \epsilon D) \cap \text{encl}(CC(r_1, r_2, r_3)) \neq \emptyset$  for all  $i \in \{1, \dots, n\}$

■ **Algorithm 1** Algorithm producing the bounding structure of a pattern.

---

**Input** : A pattern  $F = \{f_1, \dots, f_n\}$

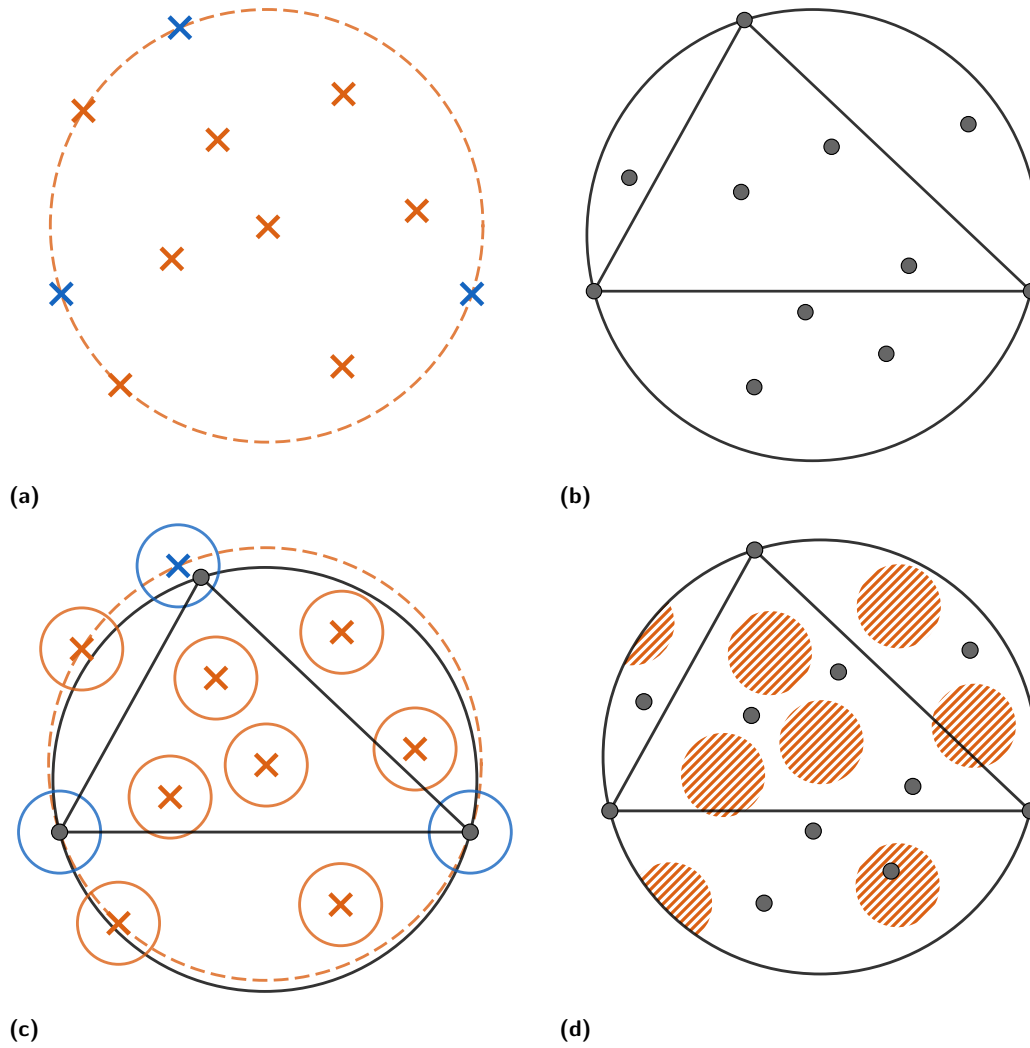
- 1 Let  $C(F) \cap F = \{f_{j_1}, \dots, f_{j_k}\}$ , where  $j_1 < \dots < j_k$
- 2  $B_F \leftarrow \{f_{j_1}, \dots, f_{j_k}\}$
- 3 **for**  $l \in 1, \dots, k$  **do**
- 4     **if**  $f_{j_l}$  is non-critical in  $F$  **then**
- 5          $F \leftarrow F \setminus \{f_{j_l}\}$
- 6          $B_F \leftarrow B_F \setminus \{f_{j_l}\}$
- 7 **Return**  $B_F$

---

### Brief Description of the Algorithm

The algorithm is in Phase 2 if  $\mathbf{a} \wedge \mathbf{c} \wedge \neg \mathbf{b}$  holds ( $\mathbf{b}$  = “the bounding structure is formed”). The objective is to have  $\mathbf{b}$ . We describe the algorithm for the following cases:  $C(R)$  has three robots and the bounding structure also has three points (Case 1),  $C(R)$  has three robots and the bounding structure has two points (Case 2),  $C(R)$  has two robots and the bounding structure has three points (Case 3) and  $C(R)$  has two robots and the bounding structure also has two points (Case 4).

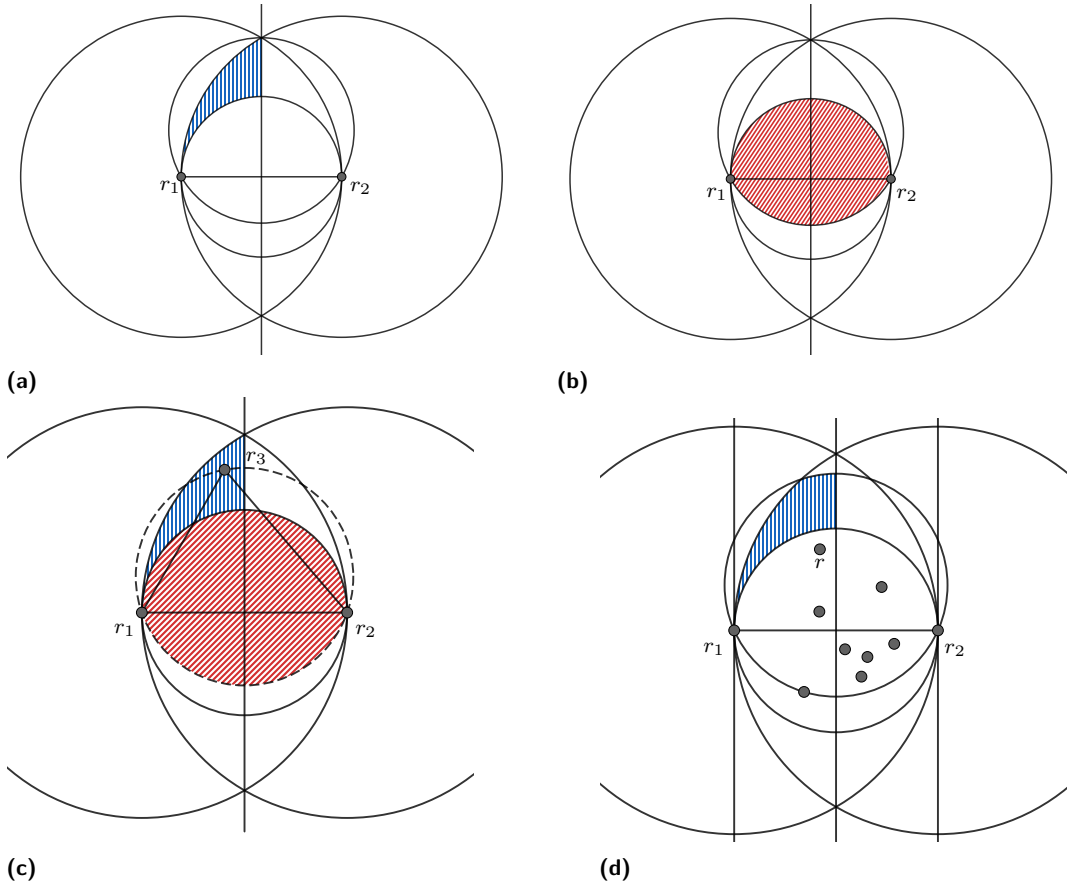
First consider Case 1. Here the goal is to transform the triangle of the robots on  $C(R)$  so that the bounding structure of  $F$  is formed. Let  $C(R) \cap R = \{r_1, r_2, r_3\}$ . If  $\Delta r_1 r_2 r_3$  is not scalene, then we shall make it so by using similar techniques from Subphase 1.1, Case 3. So now assume that  $\Delta r_1 r_2 r_3$  is scalene. Let  $\overline{seg}(r_1, r_2)$  be the largest side of  $\Delta r_1 r_2 r_3$ . In that case,  $r_3$  will be called the *transformer robot*. This robot will move to form the bounding structure of  $F$ . Let  $L$  be the perpendicular bisector of  $\overline{seg}(r_1, r_2)$ . Since no two sides of the triangle are of equal length,  $r_3 \notin L$ . Let  $\mathcal{H}$  be the open half-plane delimited by  $L$  that contains  $r_3$ . Without loss of generality, assume that  $r_1 \in \mathcal{H}$ . Let  $L_1$  be the line parallel to  $L$  and passing through  $r_1$ . Let  $\mathcal{H}'$  be the open half-plane delimited by  $L_1$  that contains  $L$ . Since  $\Delta r_1 r_2 r_3$  is acute angled,  $r_3 \in \mathcal{H}'$ . Let  $\mathcal{H}''$  be the open half-plane delimited by  $\text{line}(r_1, r_2)$  that contains  $r_3$ . Let  $C_1 = C(r_1, d(r_1, r_2))$  and  $C_2 = C(r_2, d(r_2, r_1))$ . Since  $\overline{seg}(r_1, r_2)$  is (strictly) the largest side of  $\Delta r_1 r_2 r_3$ ,  $r_3 \in \text{encl}(C_1) \cap \text{encl}(C_2)$ . If  $C_3 = CC(r_1, r_2)$ , then  $r_3 \in \text{ext}(C_3)$  as  $\Delta r_1 r_2 r_3$  is acute angled. Now take the largest side of the bounding structure  $B_F$ . In case of a tie, use the ordering of the points in the input  $F$  to choose one of them. Embed



■ **Figure 4** a) The input pattern  $F$ . The bounding structure  $b_F$  consists of the blue pattern points. b)-c) The bounding structure is formed by the robots. d) To obtain a final configuration, each shaded region must have a robot inside it.

the bounding structure  $B_F$  on the plane identifying this side with  $\overline{seg}(r_1, r_2)$  so that the third point of the bounding structure is mapped to a point  $P \in \overline{\mathcal{H}} \cap \mathcal{H}''$ . Since the bounding structure is acute angled,  $P \in \mathcal{H}'$ . Also,  $P \in ext(C_3)$  for the same reason. Furthermore, since a largest side of the bounding structure is identified with  $\overline{seg}(r_1, r_2)$ ,  $P \in \overline{encl(C_1)} \cap \overline{encl(C_2)}$ . So we have  $r_3 \in \mathcal{H} \cap \mathcal{H}' \cap \mathcal{H}'' \cap \overline{encl(C_1)} \cap \overline{encl(C_2)} \cap ext(C_3) = \mathcal{U}_{blue}$  (the blue open region in Fig. 5a) and  $P \in \overline{\mathcal{H}} \cap \mathcal{H}' \cap \mathcal{H}'' \cap \overline{encl(C_1)} \cap \overline{encl(C_2)} \cap ext(C_3) = \mathcal{U}'_{blue}$ . Notice that  $\mathcal{U}'_{blue}$  consists of the open region  $\mathcal{U}_{blue}$  and some parts of its boundary. Our objective is to move the robot  $r_3$  to a point near  $P$ . The entire trajectory of the movement should lie inside the region  $\mathcal{U}_{blue}$ . However, before this movement, we have to make sure that the configuration satisfies some desirable properties described in the following. Let  $C_4$  be the circle passing through  $r_1, r_2$  and the point in  $\mathcal{H}''$  where  $C_1$  and  $C_2$  intersect each other. We shall say that the transformer robot is *eligible to move* if  $R \cap \overline{encl(C(R))} \subset \overline{encl(C_3)} \cap \overline{encl(C_4)} = \mathcal{U}_{red}$  (the red region in Fig. 5b). The transformer robot will not move until it becomes eligible.

## 10:12 Pattern Formation by Robots with Inaccurate Movements



■ **Figure 5** a)-b) Illustrations for Phase 2, Case 1. c) Illustrations for Phase 2, Case 2. d) Illustrations for Phase 2, Case 3.

So the robots in  $encl(C(R))$  that are not in  $\mathcal{U}_{red}$ , should sequentially move inside this region first. Notice that during these movements, the configuration remains asymmetric (as  $\Delta r_1 r_2 r_3$  remains scalene) and also  $r_3$  remains the transformer robot. So when we have  $R \cap encl(C(R)) \subset \mathcal{U}_{red}$ ,  $r_3$  will become eligible to move. Now it has to move inside the region  $B(P, \epsilon D) \cap \mathcal{U}_{blue}$ . However, it is important that its trajectory lies inside  $\mathcal{U}_{blue}$ . This is because it can be shown that as long as  $r_3$  stays inside the “safe region”  $\mathcal{U}_{blue}$ , it remains as the transformer robot. This can be done by a movement scheme described in Appendix B that allows a robot to move close to a destination point through a safe region.

In Case 2,  $C(R)$  has exactly three robots and the bounding structure has exactly two points. Let  $C(R) \cap R = \{r_1, r_2, r_3\}$ . As before,  $\Delta r_1 r_2 r_3$  will be made scalene. Let  $\overline{seg}(r_1, r_2)$  be the largest side. Then  $r_3$  is the transformer robot. The plan is to move  $r_3$  inward so that it is no longer on the minimum enclosing circle. Let  $C_1, C_2, C_3, \mathcal{H}, \mathcal{H}', \mathcal{H}''$  denote the same as in Case 1. As before, we have  $r_3 \in \mathcal{H} \cap \mathcal{H}' \cap \mathcal{H}'' \cap encl(C_1) \cap encl(C_2) \cap ext(C_3) = \mathcal{U}_{blue}$  (blue region in Fig. 5c). We shall say that the transformer robot is *eligible to move* if 1)  $R \cap encl(C(R)) \subset encl(C_3) \cap encl(C(R))$  (red region in Fig. 5c) and 2)  $R \setminus \{r_3\}$  is a symmetry safe configuration. The robots in  $encl(C(R))$  that are not already in  $encl(C_3)$  will move inside it. Then we have  $C(R \setminus \{r_3\}) = C_3$  and it passes through only  $r_1$  and  $r_2$ . So  $R \setminus \{r_3\}$  will be symmetry safe if there is a unique robot closest to  $O$ , the midpoint of  $seg(r_1, r_2)$ , and it is not on  $seg(r_1, r_2)$  or its perpendicular bisector. This can be achieved easily. When  $r_3$  becomes eligible to move, it will move inside  $encl(C_3)$ . During its movement, when it has not



entered  $encl(C_3)$ , its trajectory should remain inside  $U_{blue}$ . Also, when it enters  $encl(C_3)$ , it should remain in  $ext(C)$  where  $C = C_{\uparrow}^1(R \setminus \{r_3\})$ . So its entire trajectory should be inside the region  $\mathcal{H} \cap \mathcal{H}' \cap \mathcal{H}'' \cap encl(C_1) \cap encl(C_2) \cap ext(C)$  and it should not collide with any robot upon entering  $encl(C_3)$ . This can be done by the scheme from Appendix B.

In Case 3,  $C(R)$  has exactly two robots and the bounding structure consists of exactly three points. Let  $C(R) \cap R = \{r_1, r_2\}$ . Here the strategy is to move outward one of the robots from  $encl(C(R))$ , say  $r$ , so that the minimum enclosing circle becomes the circumcircle of  $r, r_1$  and  $r_2$ . We shall call  $r$  the transformer robot. The robot farthest from  $c(R)$  will be chosen as the transformer robot. In case of a tie, it is broken using the asymmetry of the configuration. Let  $\mathcal{H}$  be the open half plane delimited by  $line(r_1, r_2)$  that contains  $r$ . Let  $L_1$  and  $L_2$  be the lines perpendicular to  $line(r_1, r_2)$  and passing through respectively  $r_1$  and  $r_2$ . Let  $L$  be the perpendicular bisector of  $seg(r_1, r_2)$ . Without loss of generality, assume that  $r \in \mathcal{S}(L_1, L) \cup L$ . Let  $C_1 = C(r_1, d(r_1, r_2))$ ,  $C_2 = C(r_2, d(r_2, r_1))$  and  $C_3 = CC(r_1, r_2)$ . Let  $C_4$  be the largest circle from the family  $\{C \in \mathcal{F}(r_1, r_2) \mid \text{center of } C \text{ lies in } \mathcal{H} \text{ and } R \subset \overline{encl}(C)\}$ . The algorithm asks  $r$  to move into the region  $encl(C_1) \cap encl(C_2) \cap ext(C_3) \cap encl(C_4) \cap \mathcal{H} \cap \mathcal{S}(L_1, L)$  (the blue region in Fig. 5d). Again, this can be done by the scheme described in Appendix B.

In Case 4,  $C(R)$  has two exactly robots and the bounding structure also has exactly two points. The only time  $\neg \mathbf{b}$  may hold is when the configuration is not symmetry safe. So we have to make the configuration symmetry safe by previously discussed techniques.

### 5.3 Phase 3

#### Motive and Overview

The algorithm is in Phase 3 if  $\mathbf{b}$  holds. The objective of this phase is to form the pattern approximately. Notice that when  $\mathbf{b}$  holds, the configuration is symmetry safe and hence asymmetric. This will allow the robots to agree on a coordinate system in which the target will be formed (approximately). During this process,  $\mathbf{b}$  has to be preserved because otherwise the agreement in coordinate system will be lost.

The termination condition of the algorithm is that both  $\mathbf{b}$  holds (i.e., it is a Phase 3 configuration) and the configuration is  $\epsilon$ -close to  $F$ . Therefore, even if the initial configuration is  $\epsilon$ -close to  $F$  (i.e., the pattern  $F$  is already formed approximately), the algorithm will still go through the earlier phases to have  $\mathbf{b}$  and then approximately form the pattern while preserving  $\mathbf{b}$ . The reason why we take this approach is because in general, even if the configuration is  $\epsilon$ -close to  $F$ , the robots may not be able to efficiently identify this. This is a basic difficulty of the problem. However, when  $\mathbf{b}$  holds there is a way to fix a particular embedding of  $F$  in the plane and then the only thing to check is whether there are robots close to each point of the embedding. For Phase 3, there are two cases to consider:  $B_F$  has exactly two points (Case 1) and  $B_F$  has exactly three points (Case 2).

#### Brief Description of the Algorithm

We shall only discuss Case 1 because its techniques can be used to solve Case 2 as well. Case 2 and all the omitted details of Case 1 can be found in the full version [3] of the paper. So for Case 1, let us first describe how we shall fix a common coordinate system. Let  $\{r_1, r_2\} = C(R) \cap R$ . Let  $\ell = line(r_1, r_2)$  and  $\ell'$  be the line passing through  $c(R)$  and perpendicular to  $\ell$ . Let  $r_i$  be the unique robot closest to  $c(R)$ . Also it is in  $encl(C(R)) \setminus (\ell \cup \ell')$ . Such a robot exists because  $\mathbf{b}$  holds. We set a *global coordinate system* whose center is at  $c(R)$ ,  $X$  axis along  $\ell$ ,  $Y$  axis along  $\ell'$ . The positive directions of  $X$  and  $Y$  axis are such that  $r_i$  lies in the positive quadrant. Now we choose an embedding of the pattern  $F$  that will be approximated. Perform a coordinate transformation (rotation) on the target

pattern  $F$  so that the bounding structure is along the  $X$  axis. Let  $F'$  denote the input after this transformation. Consider the pattern points on  $C_{\uparrow}^1(F')$  except the points of the bounding structure (notice that  $C_{\uparrow}^1(F')$  may have points from the bounding structure when  $C_{\uparrow}^1(F') = C_{\downarrow}^1(F')$ ). Reflect the pattern with respect to  $X$  axis or  $Y$  axis or both, if required, so that at least one of them is in the closed positive quadrant ( $X \geq 0, Y \geq 0$ ). Let  $F''$  denote the pattern thus obtained. Therefore, if  $\{f_i, f_j\}$  be the bounding structure, then we have 1)  $f_i, f_j$  on the  $X$  axis and 2) at least one point from  $C_{\uparrow}^1(F'') \cap (F'' \setminus \{f_i, f_j\})$  in the closed positive quadrant. Each robot applies coordinate transformations on  $F$  and obtains the same pattern  $F''$ . Let  $f_l$  denote the first pattern point from  $C_{\uparrow}^1(F'') \cap (F'' \setminus \{f_i, f_j\})$  that is in the closed positive quadrant. The pattern  $F''$  is mapped in the plane in the global coordinate system and scaled so that the bounding structure is mapped onto  $\overline{seg}(r_1, r_2)$ . These points are called the *target points*.  $T$  denotes the set of target points. Notice that the robot  $r_l$ , being the unique robot on  $C_{\uparrow}^1(R)$  and also being in an open quadrant (defined by  $\ell \cup \ell'$ ), plays crucial role in fixing the common coordinate system. This will be preserved throughout the algorithm. In particular,  $r_l$  will remain in such a position even in the final configuration. The target point that  $r_l$  will approximate in the final configuration will be the target point corresponding to  $f_l$ . Let us call it  $t_l$ . Now  $t_l$  is on  $C_{\uparrow}^1(T)$  and in the closed positive quadrant. As  $r_l$  is in the open quadrant, it does not need to move out of it to approximate  $t_l$ . Now as  $r_l$  needs to remain the closest robot from the center, we will define a circle  $C_l$ , that depends only on the position of  $t_l$ , and require that in the final configuration we have  $r_l$  inside this circle and all robots are outside the circle. If  $D$  is the diameter of  $C(T)$ , i.e.,  $D = d(r_1, r_2)$ , then define the circle  $C_l$  as (see Fig. 6) i) if  $t_l \in C_{\uparrow}^1(T) = c(T)$ , then  $C_l = C(c(T), \epsilon D)$ , ii) if  $t_l \in C_{\uparrow}^1(T) = C(T)$ , then  $C_l = C(c(T), (1 - \epsilon)\frac{D}{2})$ , iii) otherwise,  $C_l = C_{\uparrow}^1(T)$ .

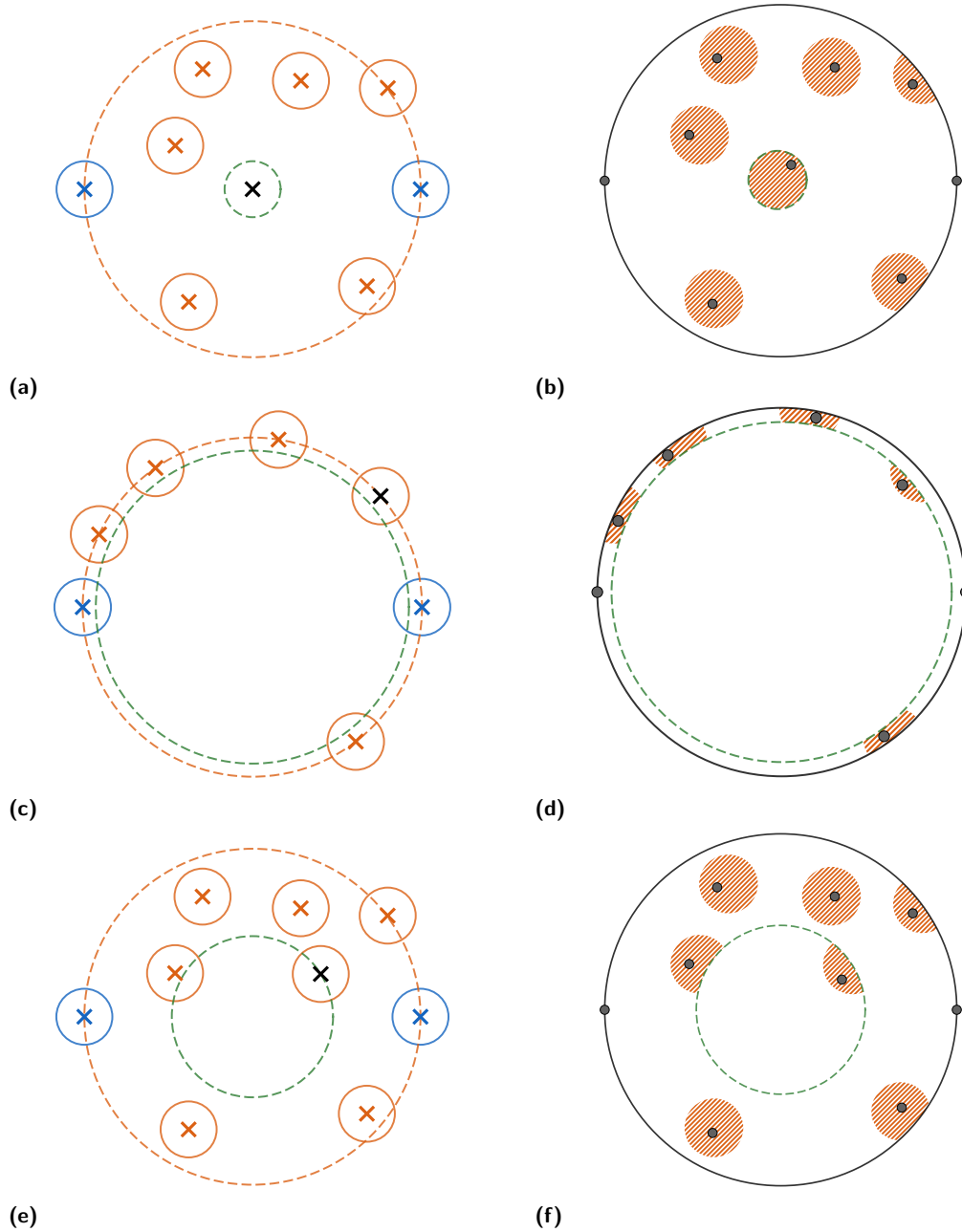
We shall say that a target point  $t \neq t_l$  is *realized* by a robot  $r$ , if  $r$  is the unique closest robot to  $t$  and  $r \in B(t, \epsilon D) \cap \overline{ext}(C_l) \cap \overline{encl}(C(R))$ . We shall say that  $t_l$  is *realized* by a robot  $r$  if all target points  $t \neq t_l$  are realized,  $r$  is the robot closest to  $t_l$  and  $r \in B(t, \epsilon D) \cap \overline{encl}(C_l)$ . Hence, if  $t_l$  is realized then it implies that all target points are realized, i.e., the given pattern is formed. We call this the final configuration (see also Fig. 6). Now the objective is to realize all the target points. This will be done in the following way. First the robot  $r_l$  moves inside  $\overline{encl}(C_l)$ , if not already there. The movement should be such that  $\mathbf{s}$  remains true. Then the robots from  $R \setminus \{r_l\}$  will sequentially realize all the target points of  $T \setminus \{t_l\}$  preserving  $\mathbf{s}$ . These movements are complicated and are described in the full version [3]. When the target points of  $T \setminus \{t_l\}$  are realized, the robot  $r_l$  will then realize  $t_l$ . Again,  $\mathbf{s}$  should remain true and  $r_l$  should remain as the unique robot closest to  $c(R)$ .

## 5.4 The Main Result

Recall that a configuration with  $\neg u \wedge (\neg a \vee \neg c)$  is in Phase 1, a configuration with  $\mathbf{a} \wedge \mathbf{c} \wedge \neg \mathbf{b}$  is in Phase 2, and a configuration with  $\mathbf{b}$  is in Phase 3. It is easy to see that any configuration with  $\neg u$  belongs to one of the three phases. Phase 1 terminates with  $\mathbf{a} \wedge \mathbf{c}$  which is either a Phase 2 or Phase 3 configuration. Phase 2 terminates with  $\mathbf{b}$  which is a Phase 3 configuration. A final configuration is formed in Phase 3. Hence the algorithm solves the problem in  $\mathcal{OBLOT} + \mathcal{SSYNC}$  from any configuration which is  $\neg u$ .

## 6 The Algorithm for Asynchronous Robots

Let us denote the algorithm presented in Section 5 as **A**. It works in  $\mathcal{OBLOT} + \mathcal{SSYNC}$ . Notice that a feature of this algorithm is that it is *sequential* in the following sense. At any round during the execution of the algorithm, at most one robot decides to move.



■ **Figure 6** Illustrations for Phase 3, Case 1. In each row, the input pattern  $F$  is shown on the left and a final configuration approximating  $F$  is shown on the right. In each case, points of the bounding structure are shown in blue,  $t_l$  is shown in black and the green circle represents  $C_l$ .

This immediately gives an algorithm that works in  $\mathcal{FCOM} + \mathcal{ASYN}\mathcal{C}$  using two colors  $\{\text{busy}, \text{idle}\}$ . The algorithm  $\mathbf{A}$  can be seen as a function that maps the snapshot taken by a robot to a movement instruction. We now construct an algorithm  $\mathbf{A}'$  from  $\mathbf{A}$  with two colors  $\{\text{busy}, \text{idle}\}$  in the following way. Initially the colors of all robots are set to `idle`. If any robot finds some robot with light set to `busy`, then it does nothing. Otherwise, it applies  $\mathbf{A}$  on its snapshot (ignoring colors). If  $\mathbf{A}$  returns a non-null move, it sets its light to `busy` and moves accordingly. If  $\mathbf{A}$  returns a null move, it sets its light to `idle` (recall that it does not know what its present color is) and does not make any move. It is easy to see that  $\mathbf{A}'$  solves the problem in  $\mathcal{FCOM} + \mathcal{ASYN}\mathcal{C}$ .

## 7 Concluding Remarks

We have introduced a model for robots with inaccurate movements. In this model, we have presented algorithms for APPROXIMATE ARBITRARY PATTERN FORMATION in  $\mathcal{OBL}\mathcal{O}\mathcal{T} + \mathcal{SSYN}\mathcal{C}$  and  $\mathcal{FCOM} + \mathcal{ASYN}\mathcal{C}$ . Solving the problem in  $\mathcal{OBL}\mathcal{O}\mathcal{T} + \mathcal{ASYN}\mathcal{C}$  is an interesting open problem. The main difficulty of the  $\mathcal{ASYN}\mathcal{C}$  setting is that a robot can see another robot while the later is moving. How will a robot identify whether a robot in its snapshot is static or moving? In  $\mathcal{FCOM}$ , a robot used the color `busy` to inform others that it is moving. But this is not possible in  $\mathcal{OBL}\mathcal{O}\mathcal{T}$ . Usually such difficulties are handled in a different way in  $\mathcal{OBL}\mathcal{O}\mathcal{T} + \mathcal{ASYN}\mathcal{C}$ . Suppose that a robot  $r$  has to move to a point  $P$ . Other robots also know this and conclude that  $r$  has completed its movement by simply observing that  $r$  has moved to  $P$ . But notice that in our case, moving exactly to  $P$  is impossible with erroneous movements. Even when  $r$  is close to  $P$ , it can not be decided whether it is still moving or not. Consider a particular situation in our algorithm where  $r$  is moving outside the smallest enclosing circle (as in Phase 1 and Phase 2), i.e., the smallest enclosing circle is changing as  $r$  is moving. If we cannot ascertain if  $r$  is moving or not, then we cannot ascertain if the smallest enclosing circle is stable or changing. Recall that the center of the smallest enclosing circle is the origin of the coordinate system with respect to which the pattern will be formed. So with a changing smallest enclosing circle, the coordinate system is also changing. So it is crucial to distinguish between moving and static robots. A possible approach in this setting could be that the robots may predict a bound on how much the coordinate system can perturb and act accordingly.

We did not consider multiplicities (points with multiple robots) in the input pattern. Since two robots cannot be brought to the same point in our model, a multiplicity can be interpreted in this case as multiple robots very close to each other. Our algorithm can be adapted to handle inputs with multiplicities. In this work, we modeled the robots as dimensionless points. Another interesting direction for future research would be to consider robots with physical extent.

---

## References

- 1 Kaustav Bose, Ranendu Adhikary, Manash Kumar Kundu, and Buddhadeb Sau. Positional encoding by robots with non-rigid movements. In *Proc. of 26th International Colloquium on Structural Information and Communication Complexity, SIROCCO 2019, L'Aquila, Italy*, volume 11639 of *LNCS*, pages 94–108. Springer, 2019. doi:10.1007/978-3-030-24922-9\_7.
- 2 Kaustav Bose, Ranendu Adhikary, Manash Kumar Kundu, and Buddhadeb Sau. Arbitrary pattern formation on infinite grid by asynchronous oblivious robots. *Theor. Comput. Sci.*, 815:213–227, 2020. doi:10.1016/j.tcs.2020.02.016.
- 3 Kaustav Bose, Archak Das, and Buddhadeb Sau. Pattern formation by robots with inaccurate movements. *arXiv*, abs/2010.09667, 2020. arXiv:2010.09667.

- 4 Kaustav Bose, Manash Kumar Kundu, Ranendu Adhikary, and Buddhadeb Sau. Arbitrary pattern formation by asynchronous opaque robots with lights. *Theor. Comput. Sci.*, 849:138–158, 2021. doi:10.1016/j.tcs.2020.10.015.
- 5 Quentin Bramas and Sébastien Tixeuil. Brief announcement: Probabilistic asynchronous arbitrary pattern formation. In *Proc. of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA*, pages 443–445, 2016. doi:10.1145/2933057.2933074.
- 6 Serafino Cicerone, Gabriele Di Stefano, and Alfredo Navarra. Asynchronous arbitrary pattern formation: the effects of a rigorous approach. *Distributed Computing*, pages 1–42, 2018. doi:10.1007/s00446-018-0325-7.
- 7 Reuven Cohen and David Peleg. Convergence of autonomous mobile robots with inaccurate sensors and movements. *SIAM J. Comput.*, 38(1):276–302, 2008. doi:10.1137/060665257.
- 8 Yoann Dieudonné, Franck Petit, and Vincent Villain. Leader election problem versus pattern formation problem. In *Proc. of 24th International Symposium on Distributed Computing, DISC 2010, Cambridge, MA, USA*, pages 267–281, 2010. doi:10.1007/978-3-642-15763-9\_26.
- 9 Paola Flocchini, Giuseppe Prencipe, Nicola Santoro, and Peter Widmayer. Arbitrary pattern formation by asynchronous, anonymous, oblivious robots. *Theor. Comput. Sci.*, 407(1-3):412–447, 2008. doi:10.1016/j.tcs.2008.07.026.
- 10 Nao Fujinaga, Yukiko Yamauchi, Hirotaka Ono, Shuji Kijima, and Masafumi Yamashita. Pattern formation by oblivious asynchronous mobile robots. *SIAM J. Comput.*, 44(3):740–785, 2015. doi:10.1137/140958682.
- 11 Giuseppe Antonio Di Luna, Paola Flocchini, Nicola Santoro, Giovanni Viglietta, and Masafumi Yamashita. Meeting in a polygon by anonymous oblivious robots. *Distributed Comput.*, 33(5):445–469, 2020. doi:10.1007/s00446-019-00362-2.
- 12 Ichiro Suzuki and Masafumi Yamashita. Distributed anonymous mobile robots: Formation of geometric patterns. *SIAM J. Comput.*, 28(4):1347–1363, 1999. doi:10.1137/S009753979628292X.
- 13 Ramachandran Vaidyanathan, Gokarna Sharma, and Jerry L. Trahan. On fast pattern formation by autonomous robots. In *Proc. of 20th International Symposium on Stabilization, Safety, and Security of Distributed Systems, SSS 2018, Tokyo, Japan*, pages 203–220, 2018. doi:10.1007/978-3-030-03232-6\_14.
- 14 Masafumi Yamashita and Ichiro Suzuki. Characterizing geometric patterns formable by oblivious anonymous mobile robots. *Theor. Comput. Sci.*, 411(26-28):2433–2453, 2010. doi:10.1016/j.tcs.2010.01.037.
- 15 Yukiko Yamauchi and Masafumi Yamashita. Randomized pattern formation algorithm for asynchronous oblivious mobile robots. In *Proc. of 28th International Symposium on Distributed Computing, DISC 2014, Austin, TX, USA*, pages 137–151, 2014. doi:10.1007/978-3-662-45174-8\_10.

## A Symmetries and Basic Impossibilities

We first present the concept of *view* (defined similarly as in [6]) of a point in a pattern or a robot in a configuration. The view of a point/robot can be used to determine whether the pattern/configuration is symmetric or asymmetric. Let  $R = \{r_1, \dots, r_n\}$  be a configuration of robots or a pattern of points. A map  $\varphi : R \rightarrow R$  is called an *isometry* or *distance preserving* if  $d(\varphi(r_i), \varphi(r_j)) = d(r_i, r_j)$  for any  $r_i, r_j \in R$ .  $R$  is said to be *asymmetric* if  $R$  admits only the identity isometry, and otherwise it is called *symmetric*. The possible symmetries that a symmetric pattern/configuration can admit are reflections and rotations. For any  $r \in R$ , its *clockwise view*, denoted by  $\mathcal{V}^\circ(r)$ , is a string of  $n + 1$  elements from  $\mathbb{R}^2$  defined as the following. For  $r \neq c(R)$ , consider the polar coordinates of the points/robots in the coordinate system with origin at  $c(R)$ ,  $\overrightarrow{c(R)r}$  as the reference axis and the angles measured in clockwise

direction. The first element of the string  $\mathcal{V}^\circ(r)$  is the coordinates of  $r$  and next  $n$  elements are the coordinates of the  $n$  points/robots ordered lexicographically. For  $r = c(R)$ , all  $n + 1$  elements are taken  $(0, 0)$ . The *counterclockwise view*  $\mathcal{V}^\circ(r)$  is defined analogously. Among  $\mathcal{V}^\circ(r)$  and  $\mathcal{V}^\circ(r)$ , the one that is lexicographically smaller is called the *view* of  $r$  and is denoted as  $\mathcal{V}(r)$ . In a configuration, each robot can compute its view as well as the views of all other robots. Hence, the following properties can be used by the robots to detect whether the configuration is symmetric or not [6]: 1)  $R$  admits a reflectional symmetry if and only if there exist two points  $r_i, r_j \in R$ ,  $r_i, r_j \neq c(R)$ , not necessarily distinct, such that  $\mathcal{V}^\circ(r_i) = \mathcal{V}^\circ(r_j)$ , 2)  $R$  admits a rotational symmetry if and only if there exist two points  $r_i, r_j \in R$ ,  $r_i \neq r_j$ ,  $r_i, r_j \neq c(R)$ , such that  $\mathcal{V}^\circ(r_i) = \mathcal{V}^\circ(r_j)$ .

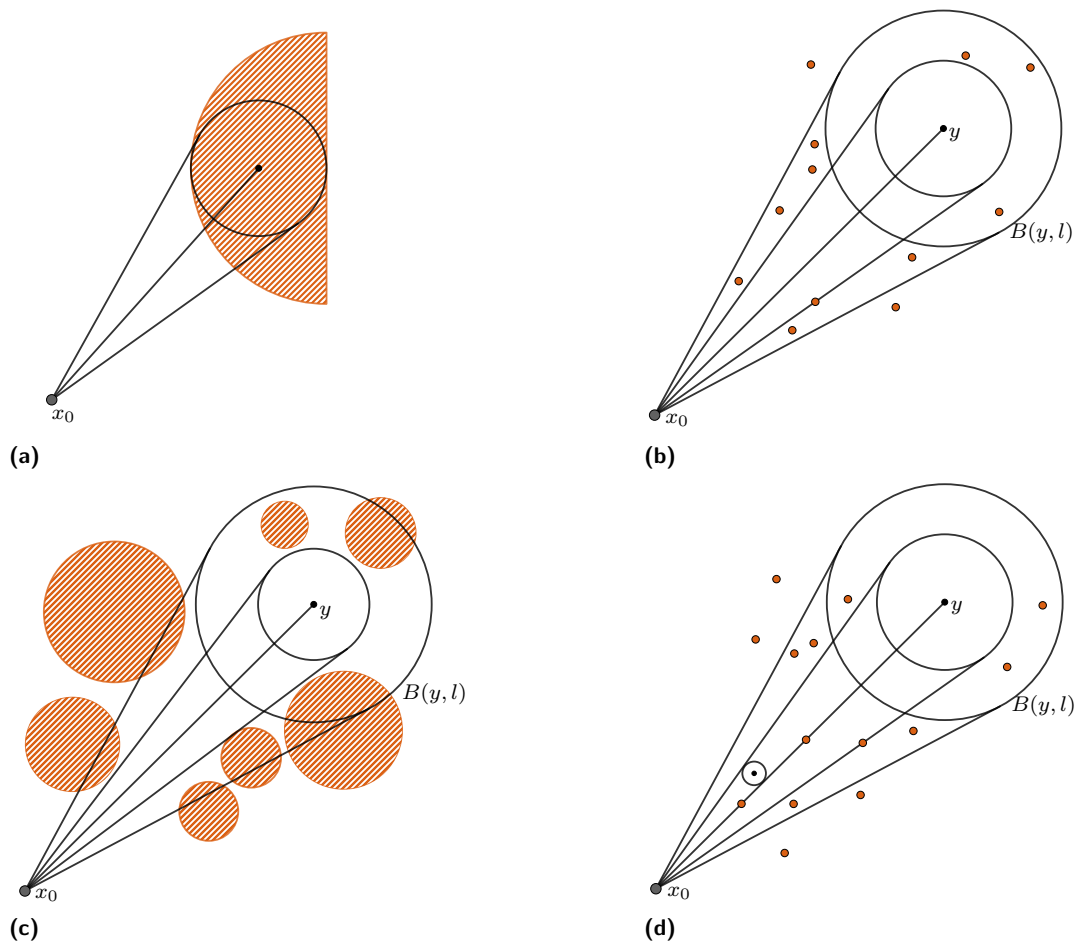
A problem that is closely related is the LEADER ELECTION problem where a unique robot from the team is to be elected as the leader. It is a well-known result [6] that LEADER ELECTION is deterministically solvable if and only if the initial configuration  $R$  does not have i) rotational symmetry with no robot at  $c(R)$  or ii) reflectional symmetry with respect to a line  $\ell$  with no robot on  $\ell$ . We call the symmetries i) and ii) *unbreakable symmetries*. If a configuration does not have such symmetries, then it can be shown that the robots can use the views to elect a unique leader. It is well-known [6] that EXACT ARBITRARY PATTERN FORMATION is deterministically unsolvable, even with RIGID movements, if the initial configuration has unbreakable symmetries. The same result holds for APPROXIMATE ARBITRARY PATTERN FORMATION as stated in Theorem 2

### Proof of Theorem 2

**Proof.** For any configuration of robots  $R$ , define  $\gamma(r)$  for any  $r \in R$  as  $\gamma(r) = \sum_{r' \in R \setminus \{r\}} d(r, r')$ . Let  $R_0$  be an initial configuration of  $n$  robots that has an unbreakable symmetry. For the sake of contradiction, assume that there is a distributed algorithm  $\mathcal{A}$  that solves APPROXIMATE ARBITRARY PATTERN FORMATION for any input  $(F, \epsilon)$  from this configuration, i.e., it forms a configuration that is  $\epsilon$ -close to  $F$ . Consider the following input pattern  $F = \{f_1, f_2, \dots, f_n\}$ , where  $f_1, f_2, f_3$  form an isosceles triangle with  $d(f_1, f_2) = d(f_1, f_3) > d(f_2, f_3)$  and  $f_4, \dots, f_n$  are arranged on the smaller side of the triangle. If  $d(f_1, f_2) = d(f_1, f_3)$  is sufficiently large compared to  $d(f_2, f_3)$  and  $\epsilon$  is sufficiently small, then for any configuration  $R'$  of robots that is  $\epsilon$ -close to  $F$ , we have  $\gamma(r_1) > \gamma(r)$  for all  $r \in R' \setminus \{r_1\}$ , where  $r_1$  is the robot approximating  $f_1$ . This property can be used to elect  $r_1$  as the leader. Hence, APPROXIMATE ARBITRARY PATTERN FORMATION can be used to solve LEADER ELECTION from the initial configuration  $R_0$ . This is a contradiction to the fact that LEADER ELECTION is deterministically unsolvable from a configuration with unbreakable symmetries [6]. ◀

## B Moving Through Safe Zone

In this section, we present some movement strategies that will be used several times in the main algorithm. Suppose that a robot needs to move to or close to some point in the plane. If the point is far away from the robot and it attempts to reach it in one step, the error would be very large and it will miss the target by a large distance. As a result, it may reach a point which causes the configuration to lose some desired property. Also, due to the large deviation from the intended trajectory, it may collide with other robots. So the robot needs to move towards its target in multiple steps and move through a “safe” region where it does not collide with any robot and the desired properties of the configuration are preserved. We first discuss the following problem. Let  $x_0$  and  $y$  be two points in the plane so that  $d(x_0, y) > l$ . Suppose that a robot  $r$  is initially at  $x_0$  and the objective is that it has to move



■ **Figure 7** Some variants of Algorithm 2. a) Starting from  $x_0$ , the robot has to move inside the shaded region. b) Starting from  $x_0$ , the robot has to move inside  $B(y, l)$  avoiding point obstacles. There are no obstacles on the line segment joining  $x_0$  and  $y$ . c) Starting from  $x_0$ , the robot has to move inside  $B(y, l)$  avoiding disk shaped obstacles. The line segment joining  $x_0$  and  $y$  does not intersect any obstacle. d) Starting from  $x_0$ , the robot has to move inside  $B(y, l)$  avoiding point obstacles. There are some obstacles on the line segment joining  $x_0$  and  $y$ .

to a point inside  $B(y, l)$  via a trajectory which lies inside  $Cone(x_0, B(y, l))$ . A pseudocode description of an algorithm that solves the problem is presented in Algorithm 2. Proof of correctness of the algorithm can be found in the full version [3] of the paper.

■ **Algorithm 2** Algorithm for moving through a safe zone.

---

```

Input : A point  $y$  on the plane and a distance  $l$ 
1  $r \leftarrow$  myself
2 if  $d(r, y) \geq l$  then
3   if  $d(r, y) = l$  or  $\frac{l}{d(r, y)} \geq \sin(\text{error}_a(r, y))$  then
4     Move to  $y$ 
5   else
6      $p \leftarrow$  point on  $\text{seg}(r, y)$  so that  $\frac{l}{d(r, y)} = \sin(\text{error}_a(r, p))$ 
7     Move to  $p$ 

```

---

We now discuss some variants of the problem. They can be solved using the movement strategy of Algorithm 2 subject to some modifications.

## 10:20 Pattern Formation by Robots with Inaccurate Movements

1. Suppose that the robot  $r$  is required to move inside some region other than a disk. Assume that the region is enclosed by some line segments and circular arcs. We can easily solve this problem using the same movement strategy, e.g., by fixing some disk  $B(y, l)$  inside the region and following Algorithm 2. See Fig. 7a.
2. Now consider the situation where the robot  $r$ , starting from  $x_0$ , have to get inside a disk  $B(y, l)$ , but there are some point obstacles that it needs to avoid. Let  $\mathcal{O} \subset \mathbb{R}^2$  be the set of obstacles. However, there are no obstacles on  $\overline{\text{seg}}(x_0, y)$ . Again a similar approach will work. Instead of  $B(y, l)$ , the robot  $r$  just needs to consider  $B(y, l')$  where  $l' \in (0, l]$  is the largest possible length such that  $\text{Cone}(r, B(y, l')) \cap \mathcal{O} = \emptyset$ . See Fig. 7b.
3. Instead of point obstacles, now consider disk shaped obstacles. Assume that none of the obstacles intersect  $\overline{\text{seg}}(x_0, y)$ . The same approach as in the previous problem would work here too. See Fig. 7c.
4. Now again consider point obstacles, but this time there might be some obstacles lying on  $\overline{\text{seg}}(x_0, y)$ . Let  $\mathcal{O}' = \mathcal{O} \cap \overline{\text{seg}}(x_0, y)$ . The robot will move to a point  $x' \in \text{Cone}(x_0, B(y, l))$  so that there is no obstacle on  $\overline{\text{seg}}(x', y)$ . For this, it will move so that it reaches a point in  $\text{Cone}(x_0, B(y, l')) \setminus \overline{\text{seg}}(x_0, y)$  where  $l' \in (0, l]$  is the largest possible length such that  $\text{Cone}(x_0, B(y, l')) \cap (\mathcal{O} \setminus \mathcal{O}') = \emptyset$ . See Fig. 7d.



# Near-Shortest Path Routing in Hybrid Communication Networks

Sam Coy ✉

University of Warwick, Coventry, UK

Michael Feldmann ✉

Paderborn University, Germany

Fabian Kuhn ✉

University of Freiburg, Germany

Philipp Schneider ✉

University of Freiburg, Germany

Artur Czumaj ✉

University of Warwick, Coventry, UK

Kristian Hinnenthal ✉

Paderborn University, Germany

Christian Scheideler ✉

Paderborn University, Germany

Martijn Struijs ✉

TU Eindhoven, The Netherlands

---

## Abstract

Hybrid networks, i.e., networks that leverage different means of communication, become ever more widespread. To allow theoretical study of such networks, [Augustine et al., SODA'20] introduced the HYBRID model, which is based on the concept of synchronous message passing and uses two fundamentally different principles of communication: a *local* mode, which allows every node to exchange one message per round with each neighbor in a *local communication graph*; and a *global* mode where *any pair* of nodes can exchange messages, but only *few such exchanges* can take place per round. A sizable portion of the previous research for the HYBRID model revolves around basic communication primitives and computing distances or shortest paths in networks. In this paper, we extend this study to a related fundamental problem of *computing compact routing schemes for near-shortest paths* in the local communication graph. We demonstrate that, for the case where the local communication graph is a *unit-disc graph* with  $n$  nodes that is realized in the plane and has *no radio holes*, we can deterministically compute a routing scheme that has constant stretch and uses labels and local routing tables of size  $O(\log n)$  bits in only  $O(\log n)$  rounds.

**2012 ACM Subject Classification** Theory of computation → Distributed algorithms

**Keywords and phrases** Hybrid networks, overlay networks

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2021.11

**Related Version** *Full Version*: <https://arxiv.org/abs/2202.08008>

**Funding** *Sam Coy*: Supported by the Centre for Discrete Mathematics and its Applications (DIMAP) and by an EPSRC studentship.

*Artur Czumaj*: Supported by the Centre for Discrete Mathematics and its Applications (DIMAP), by EPSRC award EP/V01305X/1, and by an IBM Award.

*Christian Scheideler*: Supported by the German Research Foundation (DFG) within the Collaborative Research Center 901 “On-The-Fly Computing” under the project number 160364472-SFB901.

## 1 Introduction

Humans naturally communicate in a hybrid fashion by making use of broadcast services, emails, phones, or simply face-to-face communication. Thus, it seems natural to study hybrid communication also in distributed systems. But fundamental research in this area is still in its infancy, even though there are several examples where hybrid communication is already exploited in practice. For instance, in modern data centers, wired communication networks are combined with high-speed wireless communication to reduce wire length or increase bandwidth without adding congestion to the wired network [12]. This paper focuses on hybrid *wireless* networks: networks that combine ad-hoc, WLAN-based connections (the



© Sam Coy, Artur Czumaj, Michael Feldmann, Kristian Hinnenthal, Fabian Kuhn, Christian Scheideler, Philipp Schneider, and Martijn Struijs;

licensed under Creative Commons License CC-BY 4.0

25th International Conference on Principles of Distributed Systems (OPODIS 2021).

Editors: Quentin Bramas, Vincent Gramoli, and Alessia Milani; Article No. 11; pp. 11:1–11:23



Leibniz International Proceedings in Informatics

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

*local* network) with connections via a cellular or satellite infrastructure (the *global* network). These can be realized, for instance, by smartphones, since they support both communication modes and solutions for smartphone ad hoc networks have been around for almost a decade. Connections in the local network can transfer large amounts of data cheaply, but have limited range, while global connections can transmit data between any pair of devices, but typically with bandwidth restrictions and additional costs. So ideally, global communication should be reserved for exchanging control messages while the data should be sent via the local edges, which *necessitates the computation of a routing scheme for the local network*.

The simplest solution to compute a routing scheme would be to use the global mode to collect all local device connections and/or positions in a centralized server and do the computation there. However, a centralized solution would represent a bottleneck and single point of failure, or it would not be for free when making use of a cloud service. We avoid these problems by only relying on the devices themselves. Interestingly, even without any central service, we vastly improve the results over what is possible with just the local network. More specifically, we demonstrate that with a hybrid wireless network one can significantly speed up the computation of compact routing schemes under certain natural circumstances, thereby opening up a new research direction for wireless networks.

## 1.1 Model and Problem Definition

We assume a set  $V$  of  $n$  nodes with unique IDs. Each node is associated with a fixed, distinct point in the 2-dimensional Euclidean plane, (i.e.,  $V \subseteq \mathbb{R}^2$ ), and every node  $v \in V$  knows the global coordinates of its point. We assume the standard *synchronous message passing model*: time proceeds in synchronous time slots called *rounds*. In each round, every node can perform an arbitrary amount of local computation and then communicate with other nodes.

In the HYBRID model, communication occurs in one of two modes: the *local mode* and the *global mode*. The connections for the local mode are given by a fixed graph. In our case, this graph is represented by a *unit-disk graph*  $\text{UDG}(V)$ : for any  $v, w \in V$ ,  $\{v, w\} \in \text{UDG}(V)$  if and only if  $v$  and  $w$  are at distance at most 1. For the local mode, we use the CONGEST model for simplicity: in each round, for all edges  $\{v, w\} \in \text{UDG}(V)$ , node  $v$  can send a message of  $O(\log n)$  bits to node  $w$ . However, our algorithms still work if instead the more restrictive (and more natural) Broadcast-CONGEST model is used (see the full version). We assume that each message can carry a constant number of node locations (this is analogous to the *Real RAM model*, a standard model of sequential computation).

For the global mode, we are using a variant of the *node-capacitated clique (NCC)* model called  $\text{NCC}_0$  [2] that captures key aspects of overlay networks. In this model, any node  $u$  can send messages to any other node  $v \in V$  provided  $u$  knows  $v$ 's ID. Initially, the set of IDs known to each node is just limited to its neighbors in  $\text{UDG}(V)$ . Each node is limited to sending  $O(\log n)$  messages of  $O(\log n)$  bits via the global mode in each round. W.l.o.g., we assume that whenever a node  $v$  knows the ID of some node  $w$ , it also knows  $w$ 's location (since this can be sent together with its ID).

**Model Motivation.** The assumption that the nodes know their global coordinates is motivated by the fact that smartphones can nowadays accurately determine their location using GPS or wireless access point or base station information. However, it would also be sufficient if the nodes can determine the distance and relative angles to their neighbors in the UDG (this can be obtained via known localization methods [18]), though with some precision loss.

The use of the  $\text{NCC}_0$  model in the global communication mode is motivated by the fact that nodes can communicate with any other node in the world via the cellular infrastructure given its ID (e.g., its phone number). Note that  $\text{NCC}_0$  is weaker than  $\text{NCC}$ , which assumes a clique from the start, but it is known that once the right topology has been set up in  $\text{NCC}_0$  (which can be done in  $O(\log n)$  rounds [16]), any communication round in  $\text{NCC}$  can be simulated by  $O(\log n)$  communication rounds in  $\text{NCC}_0$  [25].

**Problem Definition.** Our goal is to compute a *compact routing scheme* for  $\text{UDG}(V)$ , in hybrid networks where  $\text{UDG}(V)$  is connected and does not contain radio holes.  $\text{UDG}(V)$  is said to contain a *radio hole* if, roughly speaking, there is an internal cycle in  $\text{UDG}(V)$  that cannot be triangulated. A precise definition will be given in Section 2.

Let  $\mathcal{G}$  be a class of graphs. A stateless<sup>1</sup> routing scheme  $\mathcal{R}$  for  $\mathcal{G}$  is a family of labeling functions  $\ell_G : V(G) \rightarrow \{0, 1\}^+$  for each  $G \in \mathcal{G}$ , which assigns a bit string to every node  $v$  in  $G$ . The label  $\ell_G(v)$  serves as the address of  $v$  for routing in  $G$ : it contains the identifier of  $v$ , and may also contain information about the topology of  $G$ .

While the identifier is given as part of the input, the label is determined in the *preprocessing*. Additionally, the preprocessing has to set up a routing function  $\rho_G : V(G) \times \{0, 1\}^+ \rightarrow V(G)$  for the given graph  $G$  that, given the current node of a message and the label of the destination, determines the neighbor of  $v$  in  $G$  to forward the message to<sup>2</sup>. A routing scheme must satisfy various properties.

First of all, it must be *correct*, i.e., for every source-destination pair  $(s, t)$ ,  $\rho_G$  determines a path in  $G$  leading from  $s$  to  $t$ . Second, it must be *local*, in a sense that every node  $v$  can evaluate  $\rho_G(v, \ell)$  locally. Third, the routing should be *efficient*, i.e., the ratio of the length of the routing path and the shortest path – also known as the *stretch factor* – should be as close to 1 as possible. In our case, the length of a routing path is simply determined by the number of edges used by it. Note that whenever we have a constant stretch w.r.t. the number of edges in  $\text{UDG}(V)$ , we also have a constant stretch w.r.t. the sum of the Euclidean lengths of its edges, so we achieve a constant stretch for *both* types of metrics (see Section 2). Finally, the routing scheme should be *compact*, i.e., the labels  $\ell_G(v)$  of the nodes  $v$  and the amount of space needed at each node  $v$  to evaluate  $\rho_G(v, \ell)$  should be as small as possible.

**Problem Motivation.** There are various reasons for developing fast distributed algorithms for compact routing schemes in hybrid wireless networks. First of all, computing routing schemes for the local ad-hoc network is useful even in the presence of a cellular infrastructure since ad-hoc connections are comparatively cheap to use and typically offer a much larger bandwidth. Also, the ability to quickly compute compact routing schemes allows for frequent adaption in case of topological changes in the wireless ad-hoc network with low overhead.

## 1.2 Our Contributions

In Appendix A we show that it is impossible to set up a compact routing scheme with constant stretch in time  $o(\sqrt{n})$  when just relying on the  $\text{UDG}$  for communication even if the geometric location of all nodes is known and the  $\text{UDG}$  is hole-free. This poses the question of whether limited global communication can overcome this. We answer this question by

<sup>1</sup> In a stateless routing scheme a packet can not accumulate information along the routing path and is thus oblivious to the routing path that the packet took so far (as opposed to stateful routing).

<sup>2</sup> More general definitions of routing functions exist, but we do not require the additional power afforded by stateful routing (for instance), to compute near-constant routing schemes in logarithmic time.

showing the following result, which demonstrates the impact that a modest amount of global communication has when applied to problems which are challenging to solve locally.

► **Theorem 1.** *For a HYBRID network with a hole-free  $\text{UDG}(V)$ , a compact, stateless routing scheme can be deterministically computed for  $\text{UDG}(V)$  in  $O(\log n)$  rounds. The scheme uses node labels of  $O(\log n)$  bits and a mapping  $\rho$  that (i) can be evaluated locally with  $O(\log n)$  bits of information in each node and (ii) such that for every source-destination pair  $s, t \in V$ ,  $\rho$  determines a routing path of constant stretch from  $s$  to  $t$  in  $\text{UDG}(V)$ .*

Technical novelties of this work include a *grid graph abstraction* of any  $\text{UDG}$  which serves to sparsify the  $\text{UDG}$  while preserving its geometric structure. Computations on the grid graph can be simulated efficiently in  $\text{UDG}(V)$ . Furthermore, we can transform a routing scheme on the grid graph to one in the  $\text{UDG}$ , increasing the stretch only by a constant.

We show how to construct this abstraction in a distributed setting based entirely on local communication. This could potentially make it of interest when studying routing or distance approximation problems on  $\text{UDGs}$  in the CONGEST or Broadcast-CONGEST models or for simplification of existing algorithms. We also believe that the grid graph abstraction and its properties will be useful for future work in the HYBRID setting, making it a springboard for the case of  $\text{UDGs}$  with radio-holes.

### 1.3 Overview

The first step is the computation of a simple, yet surprisingly useful abstract graph structure on  $\text{UDG}(V)$ , which we call a *grid graph*  $\Gamma$ . The vertices of  $\Gamma$  are the centers of the cells of a regular square grid which intersect with an edge of  $\text{UDG}(V)$ . Two vertices of  $\Gamma$  share an edge iff their cells are vertically or horizontally adjacent (see Section 2.2, Figure 1). Subsequently, in Section 2.3, we tie the graphs  $\text{UDG}(V)$  and  $\Gamma$  together by defining a *representative* in  $V$  for each vertex of  $\Gamma$  that fulfills two main properties. First, two representatives of *adjacent* grid vertices are connected with a path of at most 3 hops in  $\text{UDG}(V)$  (see Figure 2). Second, each node in  $V$  has such a representative within 1 hop in  $\text{UDG}(V)$ .

We then turn to the algorithmic aspects of  $\Gamma$ . In Section 2.4 we define the *representation*  $R$  of  $\Gamma$ , where grid vertices correspond to their aforementioned representatives and grid edges correspond to paths of 3 hops in  $\text{UDG}(V)$ , and we show that  $R$  can be efficiently computed in  $\text{UDG}(V)$ . Furthermore, the representation  $R$  can be used to efficiently *simulate* the HYBRID model on  $\Gamma$ , which is summarized in Theorem 8. In Section 3, we show that an optimal path in  $\Gamma$  implies a path in  $R$  with a constant approximation ratio (Theorem 10).

The final step of the first part is to construct a constant stretch routing scheme  $\mathcal{R}$  for  $\text{UDG}(V)$  assuming that we have an optimal one for  $\Gamma$  (Section 3.2), which is encapsulated by Theorem 16. Since we can efficiently simulate the HYBRID model on  $\Gamma$  (Theorem 8), the second part can be considered in isolation from the first part. Note that so far we did not exploit the fact that  $\text{UDG}(V)$  is hole-free. In fact, the construction, simulation, and properties of  $\Gamma$  hold without that assumption, which is only needed for the second part.

Requiring the  $\text{UDG}$  to be hole-free is a strong assumption. However, we believe that at least bounding the number of holes is necessary in order to compute a compact, constant-stretch labelling scheme in  $O(\log n)$  rounds. Doing this in time and space polylogarithmic in  $n$  that also *scales well* in the number of radio holes in the  $\text{UDG}$  seems to be highly non-trivial, as these holes may intertwine in arbitrary ways, while there are exponentially many

possibilities of navigating around them.<sup>3</sup> While in our setting there still can be exponentially many simple paths between two points, we are able to exploit the lack of large holes between them to deal with arbitrarily complex boundaries of UDGs in a hybrid network setting.

To compute the routing scheme  $\mathcal{R}_\Gamma$  on  $\Gamma$ , the first step (Section 4) is to arrange the grid nodes into maximal vertical lines, called *portals* (see Figure 3a). All portals with two horizontally adjacent nodes will add one such edge, resulting in a *portal-tree*  $T_\Gamma$ , which is cycle-free because  $\text{UDG}(V)$  is hole-free (see Figure 3b). In order to compute a labelling scheme we first perform a distributed depth-first traversal on  $T_\Gamma$  (where the root is the node with min ID). This allows us to compute intervals  $I_v$  for each node  $v$  of  $T_\Gamma$  that fulfill the parenthesis theorem: it is  $I_w \supset I_v$  ( $I_w \subset I_v$ ) for each ancestor (descendant) node  $w$  of  $v$  in  $T_\Gamma$ , or else  $I_w \cap I_v = \emptyset$  when  $v, w$  are in different branches of  $T_\Gamma$  (see Figure 3d). Then all nodes of a portal will agree on interval  $I_r$  of node  $r$  that is closest to the root as their *portal label*. The challenge here is to carefully line up techniques for the more restrictive  $\text{NCC}_0$  model to obtain such a labelling in  $O(\log n)$  rounds.

Finally, in Section 5, we use  $T_\Gamma$  to route a packet from source  $s$  to target node  $t$  in  $T_\Gamma$ . Since the shortest path in  $T_\Gamma$  may not necessarily follow the tree, we have to define a routing strategy that jumps over branches when needed, for which we can use the “tree information” encoded in the labels. We use the portal labels to prioritize jumping horizontally as soon as the next portal on a path is reachable via any edge in  $\Gamma$ . Vertical routing within portals is done as a second priority for which node labels  $I_v$  are used. We prove that this strategy yields an exact routing scheme  $\mathcal{R}_\Gamma$  for  $\Gamma$  formalized in Theorem 23. Consequently, Theorem 1 is a corollary from the fact that we can emulate  $\Gamma$  on  $\text{UDG}(V)$  (Theorem 8) and that  $\mathcal{R}_\Gamma$  can be transformed into a constant stretch routing scheme  $\mathcal{R}$  for  $\text{UDG}(V)$  (Theorem 16).

## 1.4 Related Work

An early effort to formalize hybrid communication networks by [1], combined the LOCAL model with a global communication mode that essentially allows a single node to broadcast a message to all others per round. Note that this conception of the global network is fundamentally different to ours, which manifests in the fact that solving a aggregations problem (e.g., computing the sum of inputs of each node) can take  $\Omega(n)$  rounds (by contrast, it takes  $O(\log n)$  rounds in the NCC model).

Recently, shortest path problems in general **hybrid networks** have been studied by various authors [3, 9, 21, 11], which provide approximate and exact solutions for the all-pairs shortest paths problem (APSP) and the single-source shortest paths problem (SSSP). These solutions all require  $O(n^\varepsilon)$  rounds (for constant  $\varepsilon > 0$ ) to achieve a constant approximation ratio, and this is tight in the case of APSP.  $O(\log n)$ -time algorithms to solve SSSP for some classes of sparse graphs (not including UDGs) are given in [11]. Shortest path problems have also been studied for hybrid wireless networks [8]. They show that for a bounded-degree  $\text{UDG}(V)$  with a convex outer boundary, where the bounding boxes of the radio holes do not overlap, one can compute an abstraction of  $\text{UDG}(V)$  in  $O(\log^2 n)$  time so that paths of constant stretch between all source-destination pairs *outside of* the bounding boxes can be found (a simple extension of their approach to outer boundaries of *arbitrary* shape seems unlikely).

Numerous **online routing strategies** have been proposed for general UDGs, including FACE-I, FACE-II, AFR, OAFR, GOAFR and GOAFR+ [5, 24, 22, 23]. In [24, 22] it is proven that GOAFR and GOAFR+ are asymptotically optimal w.r.t. path length compared

<sup>3</sup> The number of simple  $st$ -paths that cannot be continuously deformed into each other without crossing a hole (i.e., non-homotopic paths) is  $2^h$ , where  $h$  is the number of radio holes.

to any geometric routing strategy. However, the achieved stretch is linear in the length of a shortest path. When a UDG contains the Delaunay graph of its nodes, one can exploit the fact that the Delaunay graph is a 2-spanner of the Euclidean metric [29], and MixedChordArc has been shown to be a constant-competitive routing strategy for Delaunay graphs [4]. This is only applicable in UDGs where the line segment connecting two nodes of the UDG does not intersect a boundary, which is the case if it has a convex outer boundary *and* is hole-free.

**Centralized constructions**<sup>4</sup> for compact routing schemes have been heavily investigated for general graphs (see, e.g., [28]) as well as UDGs. Here, we just focus on UDGs. Bruck et al. [6] present a medial axis based naming and routing protocol that does not require geographical locations, makes routing decisions locally, and achieves good load balancing. The routing paths seem near-optimal in simulations, but no rigorous results are given. Gao and Goswami [13] propose a routing algorithm that achieves a constant approximation ratio for load balanced routing in a UDG of arbitrary shape, but the question of near-optimal routing paths is not addressed. Based on work by Gupta et al. [17] for planar graphs, Yan et al. [30] show how to assign a label of  $O(\log^2 n)$  bits to each node of the graph such that given the labels of a source  $s$  and of a target  $t$ , one can locally compute a path from  $s$  to  $t$  with constant stretch. Using the well-separated pair decomposition (WSPD) for UDGs [14], Kaplan et al. [19] present a local routing scheme with stretch  $1+\varepsilon$  with node labels, routing tables and headers of size polynomial in  $\log n$ ,  $\log D$ , and  $1/\varepsilon$ , where  $D$  is the diameter of  $\text{UDG}(V)$ . Later, [26] shows how to achieve a stretch of  $1+\varepsilon$  without using dynamic headers.

Our routing scheme for the grid graph abstraction extends the routing scheme proposed by Santoro and Khatib [27], who presented a labelling along with an optimal routing scheme for trees by computing a minimum-distance spanning tree and labelling of that tree via a depth-first search.<sup>5</sup> In our scheme, we provide optimal paths between any source-target pair in the grid graph, because we allow using edges that are not part of the spanning tree for routing in order to jump between the branches of the spanning tree.

Our study is also related to routing problems in sparse graphs in **parallel models** [20, 10]. For example, the algorithm of Kavvadias et al. [20] can be used to compute routing tables in planar graphs in time  $\tilde{O}(1)$  and work  $\tilde{O}(n)$ . Together with the simulation framework of Feldmann et al. [11], the algorithm could in principle be used to solve our problem. However, for the simulation to work, one would need to construct a suitable global network, sparsify the graph, and, together with the simulation overhead, one would obtain a polylogarithmic runtime much higher than  $O(\log n)$ . Further, the size of the routing tables may be  $\Theta(n^2)$ .

## 2 Grid Graph

Let  $G := \text{UDG}(V)$ . The goal of this section is to construct a grid abstraction of  $G$  which makes finding routing protocols in the subsequent section manageable. In particular (but still suppressing some details), we want to simulate a bounded degree *grid graph* on  $G$  such that shortest paths in the grid graph represents only a constant factor detour in  $G$ . The way we obtain such a grid representation of  $G$  in a distributed fashion is by simulating grid nodes

<sup>4</sup> Note that in this paper, we allow ourselves just  $O(\log n)$  rounds for pre-computation and each node can learn only  $\text{polylog } n$  bits per round given that it has small ( $\text{polylog } n$ ) degree, which can be true for every node. The local network has size  $\Omega(n)$ , meaning no single node can learn it completely. This inhibits solving the problem locally at some node, i.e., by *direct* use of some centralized algorithm.

<sup>5</sup> While the routing scheme in [27] guarantees a 2-approximation for general graphs regarding the worst-case optimal cost when routing over all possible source-target-pairs, their scheme does not guarantee constant stretch when routing a message between two specific nodes  $s, t$  in the grid graph.

with real nodes of  $V$  that are close by, where edges between grid nodes correspond to paths of constant length in  $G$ . We start by introducing some notations we require in the following.

## 2.1 Preliminaries

**Graphs and Polygons in  $\mathbb{R}^2$ .** Since each node in  $V$  is associated with a point in  $\mathbb{R}^2$ , we can associate each edge  $\{u, v\} \in \text{UDG}(V)$  with the line segment with endpoints  $u$  and  $v$ , i.e., the set  $\{x \cdot u + (1-x) \cdot v \mid x \in [0, 1]\}$ . We use the names of vertices and edges to refer to their associated subsets of  $\mathbb{R}^2$  when no ambiguity arises.

A *polygonal chain* is a finite sequence of points where consecutive points are connected by segments. A polygonal chain is *closed* if the first point in the sequence is equal to the last. A *polygon* is a closed, connected, and bounded region in  $\mathbb{R}^2$  where the boundary consists of a finite number of (not necessarily disjoint) closed polygonal chains (this implies the edges in these polygonal chains have no proper intersections).

A *hole* of a polygon  $P$  is an open region in  $\mathbb{R}^2$  that is a maximal bounded and connected component of  $\mathbb{R}^2 \setminus P$ . Note that the boundary of each hole of  $P$  is equal to one of the polygonal chains bounding  $P$ . A polygon is *simple* if it has no holes.

**Distance Metrics.** We use the notation  $\|\cdot\|$  for the Euclidean metric on  $\mathbb{R}^2$ . Consequently, for  $p, q \in \mathbb{R}^2$ ,  $\|p - q\|$  denotes the Euclidean distance from  $p$  to  $q$ . For sets of points  $A, B \subseteq \mathbb{R}^2$  we define the distance between those sets as  $\text{dist}(A, B) := \min_{a \in A, b \in B} \|a - b\|$ .

Let  $P \subseteq \mathbb{R}^2$  be a polygon in the Euclidean plane and let  $p, q \in P$ . We define the *geometric distance* between  $p$  and  $q$  in  $P$ ,  $\text{dist}_P(p, q)$ , to be the length of the shortest path between  $p, q$  in  $P$ . Note that because  $P$  is a polygon, there is a polygonal chain  $\Pi = (v_1, \dots, v_k)$  from  $p$  to  $q$  inside  $P$  such that  $\text{dist}_P(p, q) = \sum_{i=1}^{k-1} \|v_{i+1} - v_i\|$ .

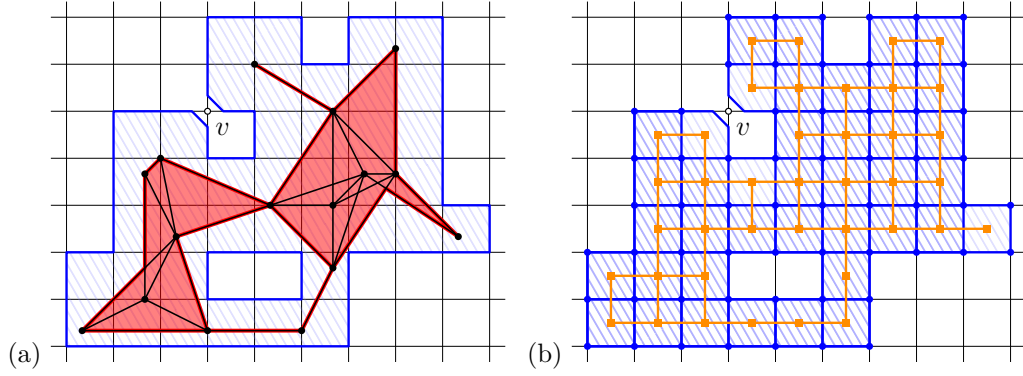
Let  $G = (V, E)$  be an embedded graph. Let  $\Pi \subseteq E$  be a path, i.e., a sequence of incident edges of  $G$ . Then we define  $\text{dist}_G(\Pi) = \sum_{(u,v) \in \Pi} \|u - v\|$ . Let  $|\Pi|$  be the number of edges (or *hops*) of a path  $\Pi$  in  $G$ . The *hop-distance* between two nodes  $u, v \in V$  is defined as  $\text{hop}_G(u, v) := \min_{u-v\text{-path } \Pi} |\Pi|$ .

## 2.2 Grid Graph Definition

We first give some definitions to formalize the notion of an UDG having radio-holes. A *triangle of UDG(V)* is a region in  $\mathbb{R}^2$  that is bounded by the edges of a 3-cycle in  $\text{UDG}(V)$  (including both the boundary and interior of the triangle). We define the *contour polygon P of UDG(V)* as the union of all triangles and edges of  $\text{UDG}(V)$ . Since  $\text{UDG}(V)$  is connected,  $P$  is indeed a polygon. We call the holes in  $P$  *radio-holes* of  $\text{UDG}(V)$ . We say an UDG has *no radio-holes* if the contour polygon of that UDG has no holes, i.e., the polygon  $P$  is simple.

Next we partition the plane into an axis-parallel square grid with side-length  $c = \frac{1}{10}\sqrt{15}$  and a fixed origin corresponding to origin of the coordinate system. Note that due to knowledge of coordinates, all nodes are aware of their position relative to the grid.

Define a square grid-cell to be *active* if it has a non-empty intersection with  $P$ . Based on this grid, we define the grid graph  $\Gamma = (V_\Gamma, E_\Gamma)$ , where  $V_\Gamma$  has a node positioned at the center of each active cell in our grid, and we have an edge in  $E_\Gamma$  between every pair of nodes of  $V_\Gamma$  that lie in adjacent cells in the grid (i.e., the square cells share an edge). The grid graph  $\Gamma$  will be simulated in the routing protocol. We will also define the cell graph  $\Gamma' = (V_{\Gamma'}, E_{\Gamma'})$  in the analysis of our protocol, but do not simulate it. We call a vertex of the grid *loose* if it is a corner of exactly 2 active cells that are not adjacent.  $\Gamma'$  is composed of the boundaries of the square grid, with  $V_{\Gamma'}$  the set of all corners of each active grid-cell that are not loose, with a pair of vertices in  $V_{\Gamma'}$  having an edge in  $E_{\Gamma'}$  if they are ends of an edge of a grid-cell. To define the *cell polygon P'*, first take the union of all active grid-cells. Then, for every



■ **Figure 1** (a):  $G := \text{UDG}(V)$  (black), polygon  $P$  (red), and cell polygon  $P'$  (blue). (b): grid graph  $\Gamma$  (orange), active grid-cells and cell graph  $\Gamma'$  (blue). The grid vertex  $v$  is loose. Note that  $G$  and  $\Gamma$  have a hole. Our routing algorithm on  $\Gamma$  would not work for this UDG, but we can still construct  $\Gamma$ .

loose vertex  $v$  in the grid, remove a triangle from  $P'$  at every active grid-cell incident to  $v$  that is small enough to be disjoint from  $P$ , such that  $P'$  no longer contains  $v$ . Note that since a loose vertex does not lie in  $P$  (otherwise, all 4 cells incident to it would have been active), such a triangle exists. See Figure 1 for an example of these definitions. Next, we define a *representative*  $r$  for each grid node  $g \in V_\Gamma$ , which simulates  $g$  throughout the rest of the protocol. We apply one of the following rules to assign a grid node to a node  $u \in V$ .

► **Definition 2.** Let  $g \in V_\Gamma$ , and let  $C$  be the grid cell of which  $g$  is the center. We define  $\mathcal{C}_1(g)$  as the set of vertices of all triangles of  $\text{UDG}(V)$  that contain the point  $g$ . We define  $\mathcal{C}_2(g)$  as the set of vertices incident to an edge that intersects  $C$ .

We define the set of candidate representatives  $\mathcal{C}(g)$  as  $\mathcal{C}_1(g) \cup \mathcal{C}_2(g)$ .

The representative of  $g$  is defined as  $r = \arg \min_{v \in \mathcal{C}_1(g)} \|v - g\|$  if  $\mathcal{C}_1(g)$  is non-empty, and  $r = \arg \min_{v \in \mathcal{C}_2(g)} \|v - g\|$  otherwise. In either case, we break ties by smallest node ID.

### 2.3 Properties of the Grid Graph

The next step is to show that the grid abstraction introduced in Definition 2 represents the UDG well. In this section we prove several properties to this effect: we show that nodes are adjacent to the representative of the cell which they are in (Lemma 3); that representatives for adjacent grid cells are close (Lemma 4); and that the cell polygon  $P'$  is simple (Lemma 5).

For brevity, all proofs in this section are delegated to the full version .

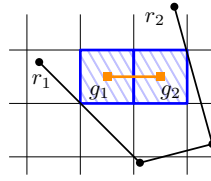
► **Lemma 3.** Let  $u, r \in V$ . If  $r$  is the representative of the cell  $C$  containing  $u$ , then  $\text{hop}_G(u, r) \leq 1$

Intuitively, this is true because  $u$  must be close to the centre of  $C$ , as must  $r$ : even if these nodes are different they cannot be too far apart.

► **Lemma 4.** Let  $(g_1, g_2) \in E_\Gamma$  be an edge in  $\Gamma$ . Let  $u, v \in V$  be representatives of  $g_1, g_2$  respectively. Then  $\text{hop}_G(u, v) \leq 3$ .

We show that the edges which define  $\mathcal{C}(g_1)$  and  $\mathcal{C}(g_2)$  are at most the diagonal of a  $2 \times 1$  block of grid cells apart. We conclude that this distance is small enough that an edge connects an endpoint of one edge with an endpoint from the other, and so the representatives of adjacent cells have distance at most 3 from each other.





■ **Figure 2** Representatives  $r_1, r_2$  of adjacent grid nodes  $g_1, g_2$  are connected by a path of 3 hops.

Finally, we show that  $P'$  is simple, i.e., it has no holes. We show this by observing that if there is a hole in  $P'$ , there is a cycle of active cells with an inactive cell in its interior. We show this cycle of cells contains a cycle of  $G$ , which implies  $G$  contains a radio-hole.

► **Lemma 5.** *If  $G$  has no holes, then  $P'$  is simple.*

## 2.4 Grid Graph Representation, Computation and Simulation

Building on the previous subsections, we show that we can efficiently simulate the grid graph  $\Gamma$  with a sub-graph  $R = (V_R, E_R)$  of the UDG  $G$  which we call a *representation* of  $\Gamma$  in  $G$  which closely approximates the structure of  $\Gamma$ . In a nutshell: the set of nodes  $V_R$  contains the set of representatives of all grid nodes  $V_\Gamma$ . On top of that, for each grid edge in  $E_\Gamma$ , we add a path in the UDG  $G$  to  $R$  between two representatives of the corresponding grid nodes (see example in Figure 2). Note that in the previous subsection we have shown the existence of such paths that have at most 3 hops.

The first goal of this subsection is to thoroughly define  $R$  and to show that we can compute  $R$  in  $G$  according to that definition in  $O(1)$  rounds. The second goal is to give an interfacing theorem for later sections that purely work with  $\Gamma$ , showing that a round of HYBRID in the grid graph  $\Gamma$  can be simulated in  $O(1)$  rounds by the nodes in  $R$  (the proof is also given in the full version). By *simulation*, we mean that one round of local communication between adjacent grid nodes in  $\Gamma$  can be performed using  $O(1)$  rounds of local communication in  $G$  to route messages between the representatives of adjacent grid nodes. An analogous property holds for the global communication.

► **Definition 6.** *Let  $\Gamma = (V_\Gamma, E_\Gamma)$  be the grid graph as defined in Section 2. A representation  $R = (V_R, E_R)$  of  $\Gamma$  in  $G$  is a sub-graph of  $G$  defined as follows. For every grid node  $g \in V_\Gamma$  with representative  $r \in V$  we define:  $r \in V_R$ . For each edge  $\{g_1, g_2\} \in E_\Gamma$  let  $r_1, r_2 \in V$  be the corresponding representatives. Then  $R$  contains all nodes and edges of one  $r_1$ - $r_2$ -path  $\Pi_{r_1, r_2}$  in  $G$  such that  $|\Pi_{r_1, r_2}| \leq 3$ . We call  $\Pi_{r_1, r_2}$  the representation of the edge  $\{g_1, g_2\}$ . Note that such a path always exists due to Lemma 4.*

► **Lemma 7.** *A representation  $R = (V_R, E_R)$  of  $\Gamma$  can be computed in  $O(1)$  rounds.*

► **Theorem 8.** *A round of the HYBRID model in  $\Gamma$  can be simulated in  $O(1)$  rounds.*

## 3 Constant Stretch Routing Scheme for the UDG

It remains to show how to leverage the grid graph constructed in the previous section for the computation of routing schemes for the UDG assuming that an exact routing scheme for the grid graph is known. We start with the analysis of the approximation factor.

### 3.1 From Shortest Paths in $\Gamma$ to Approximate Paths in $G$

The goal of this subsection is to show that shortest paths in the simulated grid graph  $\Gamma$  represent good paths in the UDG  $G$ . In particular, paths in  $G$  that are obtained via the representation  $R$  of  $\Gamma$  are constant approximations of *optimal* paths in  $G$ , both in terms of hop-length and Euclidean distance. We start by defining a *representative path*.

► **Definition 9.** Let  $s, t \in V$ . Let  $g_s, g_t \in V_\Gamma$  be the two grid nodes which are located in the same grid cell as  $s, t$  respectively. Let  $r_s, r_t$  be the representatives of  $g_s$  and  $g_t$ . Note that  $\{s, r_s\}, \{r_t, t\} \in E_G$  due to Lemma 3. Consider an optimal  $g_s$ - $g_t$ -path  $\Pi^*$ . For  $e \in \Pi^*$  let  $\Pi_e$  be the representation of the grid edge  $e \in E_\Gamma$  (see Definition 6). Let  $\Pi_{r_s, r_t} := \bigcup_{e \in \Pi^*} \Pi_e$ . We define the representative  $s$ - $t$ -path as  $\Pi_{s,t} := \{\{s, r_s\}\} \cup \Pi_{r_s, r_t} \cup \{\{r_t, t\}\}$ .

We will show that our routing scheme routes packets from  $s$  to  $t$  along the representative path  $s$ - $t$ -path  $\Pi_{s,t}$ . First, we show that these paths achieve constant stretch in  $G$ .

► **Theorem 10.** Let  $s, t \in V$ . Let  $\Pi_{s,t}$  be the  $s$ - $t$ -path given in Def. 9. If  $\{s, t\} \notin E_G$  Then  $\text{dist}(\Pi_{s,t}) \leq |\Pi_{s,t}| \leq 36 \cdot \text{dist}_G(s, t)$ .

Note that if  $\{s, t\} \in E_G$  then we can send the packet directly along this edge and the distance and number of hops is guaranteed to be optimal. If  $\{s, t\} \notin E_G$  then  $\text{dist}_G(s, t) > 1$ , a fact which we use in the proof of Theorem 10. We prove this theorem in stages represented by the subsequent lemmas. In the first stage we upper bound the number of hops of the representative path  $\Pi_{s,t}$  with the distance of a corresponding  $g_s, g_t$ -path in  $\Gamma$ .

► **Lemma 11.** Let  $s, t \in V$ . Then  $|\Pi_{s,t}| \leq \frac{3}{c} \cdot \text{dist}_\Gamma(g_s, g_t) + 2$ .

**Proof.** We exploit the fact that edges in  $\Gamma$  have distance at least  $c$  and that  $\Pi_{s,t} = \{\{s, r_s\}\} \cup \Pi_{r_s, r_t} \cup \{\{r_t, t\}\}$  is constructed from an optimal path in  $\Gamma$  (see Definition 9). We combine this with Lemmas 3, 4 to obtain the following  $|\Pi_{s,t}| \stackrel{\text{Lem. 3}}{\leq} |\Pi_{r_s, r_t}| + 2 \stackrel{\text{Lem. 4}}{\leq} 3 \cdot \text{hop}_\Gamma(g_s, g_t) + 2 \leq \frac{3}{c} \text{dist}_\Gamma(g_s, g_t) + 2$ . ◀

Since the cell-polygon  $P'$  completely covers  $P$  (the smallest polygon containing all edges of  $\Gamma$  does not, in general), we relate paths in the grid graph  $\Gamma$  to paths in the cell-graph  $\Gamma'$ . This allows us to relate paths in  $P$  to  $\Gamma$ . Note that comparisons of hop-distance in  $\Gamma$  and  $\Gamma'$  correspond to equal comparisons of distances, since both graphs have the same granularity  $c$ .

► **Lemma 12.** Let  $g_1, g_2 \in V_\Gamma$  be located in cells  $C_1, C_2$ , respectively. There exist nodes  $g'_1, g'_2 \in V_{\Gamma'}$  that are corners of  $C_1, C_2$  respectively, such that  $\text{dist}_\Gamma(g_1, g_2) \leq 2 \cdot \text{dist}_{\Gamma'}(g'_1, g'_2)$ .

**Proof.** Choose  $g'_1, g'_2$  such that  $\text{hop}_G(g'_1, g'_2) \geq 1$ . Let  $\Pi'$  be a shortest  $g'_1 g'_2$ -path in  $\Gamma'$ . Since all edges and vertices of  $\Gamma'$  are part of the boundary of an active grid cell and  $\Gamma'$  contains no loose vertices, there is a sequence  $A$  of active grid-cells from  $C_1$  to  $C_2$ , where consecutive cells share a side and each cell has an edge or vertex of  $\Pi'$  on its boundary. There are two kinds of cells in  $A$ : the first kind has an edge of  $\Pi'$  on its boundary, the second kind does not have an edge of  $\Pi'$  on its boundary, but has a vertex of  $\Pi'$  on its boundary. The number of cells of the first kind is at most  $|\Pi'|$ , because each edge in  $\Pi'$  is adjacent to at most one cell of  $A$ . The number of cells of the second kind is at most  $|\Pi'| + 1$ , because each vertex of  $\Pi'$  is adjacent to at most one cell of this type (since  $\Pi'$  has at least one edge.). So,  $|A| \leq 2|\Pi'| + 1$ .

We obtain a  $g_1 g_2$ -path  $\Pi$  of length  $|A| - 1$  in  $\Gamma$  from the chain  $A$  by taking the vertex centered at each cell in  $A$ . So, we have  $\text{hop}_\Gamma(g_1, g_2) \leq |\Pi| \leq |A| - 1 \leq 2|\Pi'| = 2 \text{hop}_{\Gamma'}(g'_1, g'_2)$ . Since all edges in  $\Gamma$  and  $\Gamma'$  have length  $c$ , we have  $\text{dist}_\Gamma(g_1, g_2) \leq 2 \text{dist}_{\Gamma'}(g'_1, g'_2)$ . ◀

We follow up on the previous stage, and bound the distance of an optimal path in the graph  $\Gamma'$  with that of an optimal *geometric* path in the *polygon*  $P'$ . The resulting approximation factor of  $\sqrt{2}$  stems from a segment-wise comparison of Euclidean distance of a shortest polygonal chain in  $P'$  to the Manhattan distance in the graph  $\Gamma'$ .

► **Lemma 13.** *Let  $g_1, g_2 \in V_{\Gamma'}$ . Then  $\text{dist}_{\Gamma'}(g_1, g_2) \leq \sqrt{2} \cdot \text{dist}_{P'}(g_1, g_2)$ .*

**Proof.** Let  $\Pi$  be the shortest *geometric* path from  $g_1$  to  $g_2$  in  $P'$ . Since  $P'$  is a polygon,  $\Pi$  is a polygonal chain connecting vertices  $g_1 =: v_1, \dots, v_n := g_2$ , where  $v_i$  are reflex vertices (i.e., vertices with an internal angle of at least  $\pi$ ) of  $P'$ . Note that by construction of  $P'$ , all reflex vertices of  $P'$  are vertices of  $\Gamma'$ , so we have  $v_1, \dots, v_n \in V_{\Gamma'}$ .

Consider one such segment  $s_i$ . Each point of  $s_i$  lies in some gridcell belonging to  $P'$ , because the path  $\Pi$  lies in  $P'$ . Therefore, there is a monotone chain of gridcells connecting  $v_i$  and  $v_{i+1}$ . Consider the axis aligned bounding rectangle  $R_i$  defined by the two opposite corners  $v_i, v_{i+1} \in V_{\Gamma'}$ . The width and length of  $R_i$  sum up to  $\|v_i - v_{i+1}\|_1$  (where  $\|(x, y)\|_1 = x + y$  for some  $(x, y) \in \mathbb{R}^2$  denotes the  $L_1$ -norm).

Traversing the boundary of the monotone chain of gridcells between  $v_i$  and  $v_{i+1}$  in the shortest possible way represents a shortest path between  $v_i$  and  $v_{i+1}$  in  $\Gamma'$ . On one hand, the length of this path equals the sum of side-lengths of  $R_i$ , i.e.,  $\text{dist}_{\Gamma'}(v_i, v_{i+1}) = \|v_i - v_{i+1}\|_1$ . On the other hand the geometric distance equals the length of  $s_i$  which is  $\text{dist}_{P'}(v_i, v_{i+1}) = \|v_i - v_{i+1}\|$ . We have

$$\text{dist}_{\Gamma'}(v_i, v_{i+1}) = \|v_i - v_{i+1}\|_1 \leq \sqrt{2} \cdot \|v_i - v_{i+1}\|_2 = \sqrt{2} \cdot \text{dist}_{P'}(v_i, v_{i+1}),$$

using the equivalence property of  $L_1$  and  $L_2$ -norms:  $\|x\|_1 \leq \sqrt{2}\|x\|_2$  for any  $x \in \mathbb{R}^2$ . So, for each segment  $S_i$  of  $\Pi$ , there exists a path in  $\Gamma'$  with stretch at most  $\sqrt{2}$  connecting the endpoints. Concatenating these paths gives the required  $g_1$ - $g_2$ -path in  $\Gamma'$ . ◀

Next we observe that an optimal path between two nodes in the UDG  $G$  can not be any shorter than a corresponding shortest geometric path in  $P'$ .

► **Lemma 14.** *Let  $s, t \in V$ . Then  $\text{dist}_{P'}(s, t) \leq \text{dist}_G(s, t)$ .*

**Proof.** Let  $\Pi$  be a shortest  $st$ -path in  $G$ . By definition, each cell that is intersected by an edge of  $\Pi$  is active and therefore this edge lies in  $P'$ . So,  $\Pi$  is an  $st$ -path in  $P'$ . ◀

We now use the inequalities proven in the lemmas above to prove Theorem 10.

**Proof of Theorem 10.** Let  $s, t \in V$  and let  $g_s, g_t \in \Gamma$  be their cell representatives. Let  $g'_s, g'_t \in V_{\Gamma'}$  be two corner-nodes of  $g_s, g_t$  for which Lemma 12 holds. Then we get

$$\begin{aligned} |\Pi_{s,t}| &\leq \frac{3}{c} \cdot \text{dist}_{\Gamma}(g_s, g_t) + 2 && \text{Lemma 11} \\ &\leq \frac{6}{c} \cdot \text{dist}_{\Gamma'}(g'_s, g'_t) + 2 && \text{Lemma 12} \\ &\leq \frac{6\sqrt{2}}{c} \cdot \text{dist}_{P'}(g'_s, g'_t) + 2 && \text{Lemma 13} \\ &\leq \frac{6\sqrt{2}}{c} \cdot (\text{dist}_{P'}(g'_s, s) + \text{dist}_{P'}(s, t) + \text{dist}_{P'}(t, g'_t)) + 2 && \text{triangle ineq.} \\ &= \frac{6\sqrt{2}}{c} \cdot (\|g'_s - s\| + \text{dist}_{P'}(s, t) + \|g'_t - t\|) + 2 && sg'_s \text{ and } tg'_t \text{ in same cell} \\ &\leq \frac{6\sqrt{2}}{c} \cdot (\|g'_s - s\| + \text{dist}_G(s, t) + \|g'_t - t\|) + 2 && \text{Lemma 14} \\ &= \frac{6\sqrt{2}}{c} \cdot (\text{dist}_G(s, t) + \sqrt{2} \cdot c) + 2 = \frac{6\sqrt{2}}{c} \cdot \text{dist}_G(s, t) + 14 \end{aligned}$$

In the equality in the fourth step we use that the segments  $sg'_s$  and  $tg'_t$  are both contained in a single grid cell, hence the distance in the cell-polygon equals the Euclidean distance.

Since a grid cell has side length  $c$ , we have  $\|g'_s - s\|, \|g'_t - t\| \leq \frac{1}{2}\sqrt{2} \cdot c$  in the second last step. As  $\Pi_{s,t}$  is a path in a UDG each edge has distance at most 1, thus

$$\text{dist}_G(\Pi_{s,t}) \leq |\Pi_{s,t}| \leq \frac{6\sqrt{2}}{c} \cdot \text{dist}_G(s, t) + 14 \leq 22 \cdot \text{dist}_G(s, t) + 14.$$

Since we have a direct edge to targets with distance at most 1, the additive error can be accounted for by increasing the multiplicative stretch by the additive error for targets at distance more than 1. Consequentially, we obtain  $\text{dist}(\Pi_{s,t}) \leq |\Pi_{s,t}| \leq 36 \cdot \text{dist}_G(s, t)$ . ◀

### 3.2 Transforming Routing Schemes for the Grid Graph to the UDG

We provide an interface to transform a routing scheme  $\mathcal{R}_\Gamma$  for the grid graph  $\Gamma$  (for which an exact routing scheme is provided in the subsequent section) into a routing scheme  $\mathcal{R}$  for the UDG  $G$  with constant stretch. The idea is to construct  $\mathcal{R}$  from  $\mathcal{R}_\Gamma$  using the representation  $R$  of  $\Gamma$  (see Definition 6). Theorem 16 provides approximation guarantees by leveraging the insights on representative paths from the previous subsection (for a proof see the full version).

► **Definition 15** (UDG Routing Scheme). *Let  $\mathcal{R}_\Gamma$  be an exact routing scheme for  $\Gamma$  consisting of  $\ell_\Gamma : V_\Gamma \rightarrow \{0, 1\}^+$  and  $\rho_\Gamma : V_\Gamma \times \{0, 1\}^+ \rightarrow V_\Gamma$ . Let  $R = (V_R, E_R)$  be the representation of  $\Gamma$  (see Def. 6). The routing scheme  $\mathcal{R}$  for  $G$  is defined on the basis of grid cells. Let  $C$  be a cell with grid node  $g \in V_\Gamma$  and let  $r \in V_R$  be the representative of  $g$ . For each  $v \in C$  we set  $\ell_G(v) := \ell_\Gamma(g) \circ ID(v)$  (where “ $\circ$ ” represents the concatenation of bit strings). The routing function  $\rho_G$  is defined as follows. Let  $v \in V_G$  be the current node and let  $\ell_t := \ell_{\Gamma,t} \circ ID(t)$  be the label of the target node  $t \in V$ , where  $\ell_{\Gamma,t}$  is the label of the representative in  $t$ 's cell w.r.t.  $\mathcal{R}_\Gamma$ . We assume  $t \neq v$ , as otherwise the packet has already arrived.*

1. *If  $\{v, t\} \in E_G$ , then we can directly deliver to  $t$ :  $\rho_G(v, \ell_t) := t$ .*
2. *Else, if  $v \in C \setminus V_R$  is the source we directly route to the representative of  $C$ :  $\rho_G(v, \ell) := r$ .*
3. *Else, if  $v = r$  is the representative of this grid cell  $C$ , let  $g' := \rho_\Gamma(g, \ell_{\Gamma,t})$  be the next grid node suggested by  $\mathcal{R}_\Gamma$ . Let  $u$  be the first node on the path  $\Pi_{\{g, g'\}} \subseteq E_R$  that represents the edge  $\{g, g'\} \in E_\Gamma$ . Then  $\rho_G(v, \ell_t) := u$ .*
4. *Else, if  $v \in V_R$  but  $v$  is not the representative of  $C$ , then  $v$  must be a “transitional node” on  $\Pi_{\{g, g'\}} \in E_R$  that represents  $\{g, g'\} \in E_\Gamma$ . W.l.o.g. let  $g' := \rho_\Gamma(g, \ell_{\Gamma,t})$  be the next grid node suggested by  $\mathcal{R}_\Gamma$  and  $u$  be the next node on  $\Pi_{\{g, g'\}}$  towards  $g'$ . Then  $\rho_G(v, \ell_t) := u$ .*

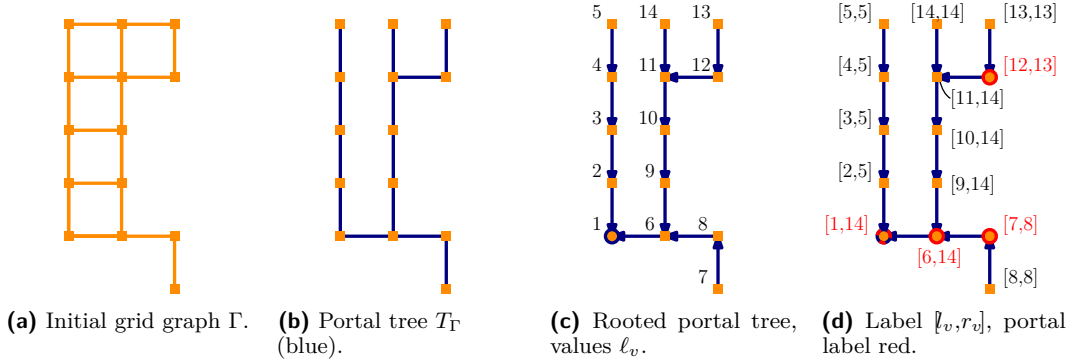
► **Theorem 16.** *Let  $\mathcal{R}_\Gamma$  be a local, correct, exact routing scheme for  $\Gamma$  with labels and local routing information of  $O(\log n)$  bits. Then the routing scheme  $\mathcal{R}$  from Definition 15 is local, correct, has constant stretch, labels and local routing information of size  $O(\log n)$  bits and can be computed in  $O(1)$  rounds.*

## 4 Computing a Labelling for the Grid Graph

This section is dedicated to computing the labelling  $\ell_\Gamma : V_\Gamma \rightarrow \{0, 1\}^+$  for the grid graph by first constructing a particular tree structure  $T_\Gamma$  and then computing a labelling on it in  $O(\log n)$  rounds leveraging various HYBRID (and in particular NCC<sub>0</sub>) model techniques. For the tree-labelling we use a similar approach as presented in [27], but slightly adapt the labelling which later allows jumping over branches of our specifically constructed tree, facilitating an optimal routing scheme in grid graphs. Afterwards, Section 5 will deal with computing the routing function  $\rho_\Gamma : V_\Gamma \times \{0, 1\}^+ \rightarrow V_\Gamma$  leading to the routing scheme  $\mathcal{R}_\Gamma$ .

We assume the HYBRID model on the grid graph  $\Gamma$  that represents the network which we constructed and simulated in the previous sections (Theorem 8). The goal is to divide the

grid nodes into sets of vertically connected grid nodes called *portals*. Connecting neighboring portals with a single edge gives us a spanning tree of  $\Gamma$ , which we call *portal tree*. We then root the portal tree at the node with minimum identifier and compute a label for each grid node, leading to a well-defined labelling function  $\ell_\Gamma$ . Note that we require that the cell polygon  $P'$  does not contain holes (Lemma 5), as otherwise there be a cycle after connecting neighboring portals.



■ **Figure 3** Example for creation and labelling of the portal tree.

We first define the set of portals as follows:

► **Definition 17 (Portals).** Let  $\Gamma = (V_\Gamma, E_\Gamma)$  be the grid graph as constructed in the last section. The set of portals are the connected components of  $(V_\Gamma, E_{vert})$ , where  $E_{vert} \subset E_\Gamma$  are the vertical edges of the grid graph.

For convenience, assume that the grid nodes  $v_1, \dots, v_k$  within a portal  $\mathcal{P}$  are sorted by their  $y$ -coordinates in descending order, i.e.,  $v_1$  is the northernmost node.

To construct the portal tree  $T_\Gamma$  of the grid graph  $\Gamma$  we connect neighboring portals via a single edge. Each grid node  $v$  checks whether it has an edge to the left and communicates this to its northern and southern neighbors  $v_N$  and  $v_S$ . Assume that  $v$  has an edge  $\{v, v_W\}$  to the left. Then  $v$  checks if  $v_S$  also has a horizontal edge to the left. If that is not the case,  $v$  adds the edge  $\{v, v_W\}$  to the portal tree. We refer to Figures 3a and 3b for an example. This gives us the following lemma, the proof of which is delegated to Appendix C.1:

► **Lemma 18.** The portal tree  $T_\Gamma$  of a grid graph  $\Gamma$  can be computed in  $O(1)$  rounds.

Given the portal tree  $T_\Gamma$ , we want to compute a unique label for each grid node that reflects its structure as portal tree. First, we root  $T_\Gamma$  at the grid node  $r$  whose representative is the UDG node  $u$  with minimal identifier, using pointer jumping (Appendix C.2) on the cycle of all grid nodes that corresponds to an Euler tour (Appendix C.3).

Now we compute the labelling for the (rooted) portal tree. For each grid node  $v$  in  $T_\Gamma$ , we aim to assign an interval  $I_v = [l_v, r_v] \in \mathbb{N}^2$  to  $v$ , such that  $I_v \supset I_w$  for any child node  $w$  of  $v$  in  $T_\Gamma$ . To obtain the left interval border  $l_v$  for each grid node  $v$  in the portal tree, we perform a depth-first traversal (DFS) on  $T_\Gamma$  in  $O(\log n)$  rounds, using Lemma 28 (see Appendix C.4). The value  $l_v$  is then the preorder number of  $v$  according to the DFS. Note that  $l_v < l_u$  for any node  $u$  lying in the subtree of  $v$ . We then compute the number  $r_v$ , corresponding to the maximum left interval border among all nodes in  $v$ 's subtree. In a nutshell, we first let all nodes compute some value  $d \in O(\log n)$ ,  $d \geq \log D(T_\Gamma)$ , where  $D(T_\Gamma)$  is the depth of the portal tree. Then we generate additional edges in  $T_\Gamma$  for  $d$  iterations, by performing pointer-jumping on the paths from the leaf nodes of  $T_\Gamma$  to the root. We perform the pointer-jumping technique in a condensed way to ensure that the node degrees do not exceed  $O(\log n)$ . With the help of these additional edges, we let each node  $v \in T_\Gamma$  compute the value  $r_v$  as an aggregate of the  $l_u$ -values of all nodes  $u$  that are contained in the subtree  $T_\Gamma(v)$  of  $T_\Gamma$  with root  $v$ . We elaborate on this approach in Appendix C.5 (see Lemma 29).

After the algorithm has terminated, each node  $v$  knows the correct value  $r_v$  and thus its interval  $I_v = [l_v, r_v]$ . Observe that grid nodes which are in different branches of the portal tree have incomparable labels. We obtain the following lemma:

► **Lemma 19.** *Given a rooted portal tree  $T_\Gamma$ , each node  $v \in T_\Gamma$  can compute an interval  $I_v = [l_v, r_v]$  in  $O(\log n)$  rounds, such that  $I_v \supset I_w$  for any child node  $w$  of  $v$  in  $T_\Gamma$ .*

Now that each grid node  $v$  knows its interval in  $T_\Gamma$  we need to perform one final step. In addition to its own (unique) interval, a grid node  $v$  needs to know the interval that has been assigned to the node  $v_i$  which is closest to the root within its own portal. We call this label the *portal label* of  $v$ . The node on a portal which is closest to the root can determine this locally. Each portal label can then be broadcasted to all nodes within the respective portal in  $O(\log |\mathcal{P}|)$  rounds (see Lemma 26), so we obtain the following lemma (cf. Figures 3c and 3d).

► **Lemma 20.** *After  $O(\log n)$  rounds, each grid node  $v$  in the portal  $\mathcal{P} = (v_1, \dots, v_k)$  knows the interval  $I_{v_i}$  of the node  $i \in \mathcal{P}$  closest to the root of the portal tree.*

Observe that, the way we defined the portal labels we obtain the property that for portal labels of two neighboring portals, one portal's label is always a subset of the other. Combining Lemma 19 and Lemma 20 yields the main result of this section.

► **Theorem 21.** *Computing the labelling  $\ell_\Gamma : V_\Gamma \rightarrow \{0, 1\}^+$  for the grid graph  $\Gamma$  as part of the routing scheme  $\mathcal{R}_\Gamma$  can be done within  $O(\log n)$  rounds.*

## 5 Compact Routing Scheme for the Grid Graph

Finally, we explain our routing strategy for transmitting a packet between two nodes  $s, t \in V_\Gamma$  in the grid graph, leading to the routing function  $\rho_\Gamma : V_\Gamma \times \{0, 1\}^+ \rightarrow V_\Gamma$ . At the start of the routing protocol, the node  $s$  generates a message  $m$  that contains the identifier of the target node  $t$ , as well as  $t$ 's label and portal label. The goal of our routing strategy is to route  $m$  to  $t$  along grid edges via an optimal path in the grid graph. To do so, each grid node receiving the message  $m$  has to decide which of its grid neighbors to forward  $m$  to, using only the information stored in  $m$ , and the information stored in its own local memory. Briefly, the strategy works as follows. While we are not at the portal containing  $t$ , we always try going left (west) or right (east) first by going to a portal whose label is closest to the portal label of the target node  $t$ . If going east or west is not possible, we go up (north) or down (south) instead by comparing  $g$ 's own label with the *actual* label of the target node  $t$ . Once we are at the portal that contains the target node, we only consider going up or down until we reach  $t$ .

**Detailed Description.** We describe the routing strategy in more detail now (see the full version for pseudocode). Assume we are at a grid node  $g$  and want to route a message  $m$  to a grid node  $t$ . We introduce the following notation for the information known to  $g$ . Note that grid nodes obtain this information in one communication round with their neighbors.

► **Definition 22.** *The information required to be stored by a grid node  $g \in V_\Gamma$  are denoted by the following variables.*

- (i)  $g.L \in \mathbb{N}^2$ :  $g$ 's own interval given to it by labelling of the portal tree.
- (ii)  $g.P \in \mathbb{N}^2$ : The portal label of the portal containing  $g$ .
- (iii)  $g_N, g_S, g_E, g_W \in V_\Gamma \cup \{\perp\}$ :  $g$ 's grid neighbors in north, south, east and west direction ( $\perp$  denotes that there is no such neighbor). For each of these grid neighbors  $g$  also knows the label of the grid node and the portal label of the grid node.

Additionally, we store the label  $t.L$  of  $t$  and the portal label  $t.P$  of  $t$  in the message  $m$ , so  $g$  knows these as well upon receipt of  $m$ . Note that storing this information at  $g$  requires only  $O(\log n)$  bits. Assuming that  $g \neq t$ ,  $g$  must decide which of its grid neighbors  $g_W, g_E, g_N, g_S$

to forward  $m$  to. Node  $g$  first checks if it is in the same portal as  $t$  by comparing  $g.P$  and  $t.P$ . Assume that this is not the case. Then  $g$  has to consider the following cases. We use the notation  $a \approx b$  to denote that label  $a$  is *incomparable* to label  $b$ , i.e.,  $a \not\subseteq b \wedge b \not\subseteq a$ .

We start by explaining how a message  $m$  is routed in horizontal direction.

- (i)  $g.P \subset t.P$  or  $g.P \supset t.P$ . In case  $g.P \subset t.P$  then  $g$  checks if either  $g.P \subset g_W.P \subset t.P$  or  $g.P \subset g_E.P \subset t.P$  holds (only one of these conditions can be true). In the first case,  $g$  forwards  $m$  to  $g_W$ , in the second case  $g$  forwards  $m$  to  $g_E$ . If none of the conditions hold (for example, if  $g_W = \perp$  or  $g_E = \perp$ ), then  $g$  routes  $m$  vertically (see the description below). The case  $g.P \supset t.P$  works analogously.
- (ii)  $g.P \approx t.P$ . In this case  $g$  tries to forward  $m$  horizontally to a node, whose portal label is a superset of  $g.P$ . By doing so,  $m$  eventually reaches a node  $g'$  whose portal label is also a superset of  $t.P$  (at the closest “common ancestor portal”), and case (i) is considered. If neither  $g_W$  nor  $g_E$  satisfies this condition or does not exist,  $g$  routes vertically.

We now explain how  $m$  is routed in vertical direction. We do this if  $g$  has not been able to route  $m$  horizontally (either because its horizontal neighbors are not appropriate, or because they do not exist) or if it is already contained in the same portal as the target node  $t$ . Again,  $g$  considers the following cases, this time for its own label  $g.L$  instead for  $g.P$  and for the actual label  $t.L$  instead of the portal label  $t.P$ .

- (i)  $g.L \subset t.L$  or  $g.L \supset t.L$ . In the case  $g.L \subset t.L$  node  $g$  checks if either  $g.L \subset g_N.L \subset t.L$  or  $g.L \subset g_S.L \subset t.L$  holds. In the first case,  $g$  forwards  $m$  to  $g_N$ , in the second case  $g$  forwards  $m$  to  $g_S$ . The case  $g.L \supset t.L$  works analogously.
- (ii)  $g.L \approx t.L$ . If the labels  $g.L$  and  $t.L$  are incomparable,  $g$  tries to forward  $m$  vertically to a node, whose label is a superset of  $g.L$ . This is the case for either  $g_N$  or  $g_S$ , depending on the location of the root of the labeled tree.

**Analysis of the Routing Strategy.** We show that our routing strategy is local, efficient, and correct, so it fulfills all requirements for a routing scheme. Our routing strategy is local, as each node  $v$  can determine the next node to forward the message  $m$  to based solely on the  $O(\log n)$  bits of local information, and the labels  $t.L$  and  $t.P$  given to  $v$  upon receipt of  $m$ .

Regarding efficiency of our routing strategy, we prove in the full version that it is optimal. The idea is to show that in case the message is routed in a specific direction, there exists at least one optimal path that moves in the same direction. We conclude the following theorem.

► **Theorem 23.** *A local, correct and exact routing scheme  $\mathcal{R}_\Gamma$  for  $\Gamma$  using node labels and local space of  $O(\log n)$  bits can be computed in  $O(\log n)$  rounds in the HYBRID model.*

## 6 Conclusion

We showed that for any HYBRID network with a hole-free  $\text{UDG}(V)$ , a compact routing scheme can be computed for  $\text{UDG}(V)$  in just  $O(\log n)$  rounds. There are various interesting directions for follow-up research. For example, we suspect that our approach can be generalized to 3 dimensions (potentially more) where the corresponding “unit ball graph” implies a polyhedron of genus 0. In particular, some approach akin to multidimensional range trees might work: define  $\Gamma$  analogously in a three dimensional grid; dissect  $\Gamma$  along 2d-hyperplanes to obtain 2d-portals in  $\Gamma$  – if one then comes up with a routing scheme to find the correct 2d-portal, then this can be applied alongside the 2d-routing algorithm presented here to find the correct node in that 2d-portal. There are unresolved issues, however. Another interesting direction is to efficiently compute compact routing schemes for *arbitrary* connected UDGs, or ideally,



to find efficient solutions for arbitrary planar graphs. This seems to be a daunting task; a simpler setting might be to consider UDGs with a small number of holes where our grid construction could be of help. Finally, it would be interesting to think about adaptations of our routing scheme to also minimize congestion, which should be possible in the special case of hole-free UDGs (see for example the case where the contour polygon is a square [7]).

---

## References

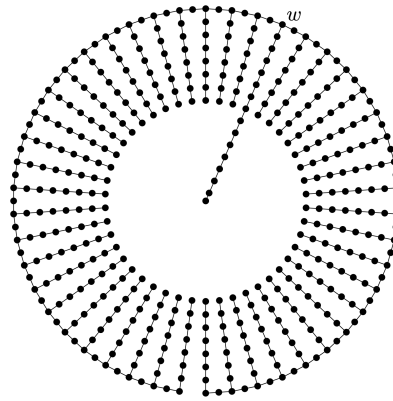
- 1 Yehuda Afek, Gad M. Landau, Baruch Schieber, and Moti Yung. The power of multimedia: Combining point-to-point and multiaccess networks. *Information and Computation*, 84(1):97–118, January 1990. doi:10.1016/0890-5401(90)90035-G.
- 2 John Augustine, Keerti Choudhary, Avi Cohen, David Peleg, Sumathi Sivasubramaniam, and Suman Sourav. Distributed graph realizations. In *Proc. of the 34th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2020)*, pages 158–167, 2020. doi:10.1109/IPDPS47924.2020.00026.
- 3 John Augustine, Kristian Hinnenthal, Fabian Kuhn, Christian Scheideler, and Philipp Schneider. Shortest paths in a hybrid network model. In *Proc. of the 31st ACM-SIAM Symposium on Discrete Algorithms (SODA 2020)*, pages 1280–1299, 2020. doi:10.1137/1.9781611975994.78.
- 4 Nicolas Bonichon, Prosenjit Bose, Jean-Lou De Carufel, Vincent Despré, Darryl Hill, and Michiel H. M. Smid. Improved routing on the Delaunay triangulation. In *Proc. of the 26th Annual European Symposium on Algorithms (ESA 2018)*, pages 22:1–22:13, 2018. doi:10.4230/LIPIcs.ESA.2018.22.
- 5 Prosenjit Bose, Pat Morin, Ivan Stojmenovic, and Jorge Urrutia. Routing with guaranteed delivery in ad hoc wireless networks. *Wireless Networks*, 7(6):609–616, 2001. doi:10.1023/A:1012319418150.
- 6 Jehoshua Bruck, Jie Gao, and Anxiao Jiang. MAP: medial axis based geometric routing in sensor networks. *Wireless Networks*, 13(6):835–853, 2007. doi:10.1007/s11276-006-9857-z.
- 7 Antonio Carzaniga, Koorosh Khazaei, and Fabian Kuhn. Oblivious low-congestion multicast routing in wireless networks. In *Proc. of the 13th ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc 2012)*, pages 155–164, 2012. doi:10.1145/2248371.2248395.
- 8 Jannik Castenow, Christina Kolb, and Christian Scheideler. A bounding box overlay for competitive routing in hybrid communication networks. In *Proc. of the 21st International Conference on Distributed Computing and Networking (ICDCN 2020)*, pages 14:1–14:10, 2020. doi:10.1145/3369740.3369777.
- 9 Keren Censor-Hillel, Dean Leitersdorf, and Volodymyr Polosukhin. Distance computations in the hybrid network model via oracle simulations. *CoRR*, abs/2010.13831, 2020. arXiv:2010.13831.
- 10 Hristo N. Djidjev, Grammati E. Pantziou, and Christos D. Zaroliagis. Computing shortest paths and distances in planar graphs. In *Proc. of the 18th International Colloquium on Automata, Languages and Programming (ICALP 1991)*, pages 327–338, 1991. doi:10.1007/3-540-54233-7\_145.
- 11 Michael Feldmann, Kristian Hinnenthal, and Christian Scheideler. Fast hybrid network algorithms for shortest paths in sparse graphs. In *Proc. of the 24th International Conference on Principles of Distributed Systems (OPODIS 2020)*, pages 31:1–31:16, 2020. doi:10.4230/LIPIcs.OPODIS.2020.31.
- 12 Klaus-Tycho Foerster and Stefan Schmid. Survey of reconfigurable data center networks: Enablers, algorithms, complexity. *SIGACT News*, 50(2):62–79, 2019. doi:10.1145/3351452.3351464.
- 13 Jie Gao and Mayank Goswami. Medial axis based routing has constant load balancing factor. In *Proc. of the 23rd Annual European Symposium on Algorithms (ESA 2015)*, pages 557–569, 2015. doi:10.1007/978-3-662-48350-3\_47.

- 14 Jie Gao and Li Zhang. Well-separated pair decomposition for the unit-disk graph metric and its applications. *SIAM Journal on Computing*, 35(1):151–169, 2005. doi:10.1137/S0097539703436357.
- 15 Robert Gmyr, Kristian Hinnenthal, Christian Scheideler, and Christian Sohler. Distributed monitoring of network properties: The power of hybrid networks. In *Proc. of the 44th International Colloquium on Automata, Languages and Programming (ICALP 2017)*, pages 137:1–137:15, 2017. doi:10.4230/LIPIcs.ICALP.2017.137.
- 16 Thorsten Götte, Kristian Hinnenthal, Christian Scheideler, and Julian Werthmann. Time-optimal construction of overlay networks. *CoRR*, abs/2009.03987, 2020. arXiv:2009.03987.
- 17 Anupam Gupta, Amit Kumar, and Rajeev Rastogi. Traveling with a pez dispenser (or, routing issues in MPLS). *SIAM Journal on Computing*, 34(2):453–474, 2004. doi:10.1137/S0097539702409927.
- 18 Fabian Höflinger, Joan Bordoy, Rui Zhang, Amir Bannoura, Nikolas Simon, Leonhard M. Reindl, and Christian Schindelhauer. Localization system based on ultra low-power radio landmarks. In *Proc. of the 7th International Conference on Sensor Networks (SENSORNETS 2018)*, pages 51–59, 2018. doi:10.5220/0006608800510059.
- 19 Haim Kaplan, Wolfgang Mulzer, Liam Roditty, and Paul Seiferth. Routing in unit disk graphs. *Algorithmica*, 80(3):830–848, 2018. doi:10.1007/s00453-017-0308-2.
- 20 Dimitris J Kavvadias, Grammati E Pantziou, Paul G Spirakis, and Christos D Zaroliagis. Hammock-on-ears decomposition: A technique for the efficient parallel solution of shortest paths and other problems. *Theoretical Computer Science*, 168(1):121–154, 1996. doi:10.1016/S0304-3975(96)00065-5.
- 21 Fabian Kuhn and Philipp Schneider. Computing shortest paths and diameter in the hybrid network model. In *Proc. of the 39th Annual ACM Symposium on Principles of Distributed Computing (PODC 2020)*, pages 109–118, 2020. doi:10.1145/3382734.3405719.
- 22 Fabian Kuhn, Roger Wattenhofer, Yan Zhang, and Aaron Zollinger. Geometric ad-hoc routing: of theory and practice. In *Proc. of the 22nd ACM Symposium on Principles of Distributed Computing (PODC 2003)*, pages 63–72, 2003. doi:10.1145/872035.872044.
- 23 Fabian Kuhn, Roger Wattenhofer, and Aaron Zollinger. Asymptotically optimal geometric mobile ad-hoc routing. In *Proc. of the 6th International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications (DIAL-M 2002)*, pages 24–33, 2002. doi:10.1145/570810.570814.
- 24 Fabian Kuhn, Roger Wattenhofer, and Aaron Zollinger. Worst-case optimal and average-case efficient geometric ad-hoc routing. In *Proc. of the 4th ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc 2003)*, pages 267–278, 2003. doi:10.1145/778415.778447.
- 25 Frank Thomson Leighton, Bruce M. Maggs, Abhiram G. Ranade, and Satish Rao. Randomized routing and sorting on fixed-connection networks. *Journal of Algorithms*, 17(1):157–205, 1994. doi:10.1006/jagm.1994.1030.
- 26 Wolfgang Mulzer and Max Willert. Compact routing in unit disk graphs. In *Proc. of the 31st International Symposium on Algorithms and Computation (ISAAC 2020)*, pages 16:1–16:14, 2020. doi:10.4230/LIPIcs.ISAAC.2020.16.
- 27 Nicola Santoro and Ramez Khatib. Labelling and implicit routing in networks. *The Computer Journal*, 28(1):5–8, 1985. doi:10.1093/comjnl/28.1.5.
- 28 Mikkel Thorup and Uri Zwick. Compact routing schemes. In *Proc. of the 13th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA 2001)*, pages 1–10, 2001. doi:10.1145/378580.378581.
- 29 Ge Xia. The stretch factor of the Delaunay triangulation is less than 1.998. *SIAM Journal on Computing*, 42(4):1620–1659, 2013. doi:10.1137/110832458.
- 30 Chenyu Yan, Yang Xiang, and Feodor F. Dragan. Compact and low delay routing labeling scheme for unit disk graphs. *Computational Geometry: Theory and Applications*, 45(7):305–325, 2012. doi:10.1016/j.comgeo.2012.01.015.

## A Lower Bound Without Global Communication

The counterexample in Figure 4 (which follows the arguments of [23]) demonstrates that it is impossible to set up a compact routing scheme with constant stretch in polylogarithmic time when just relying on the unit-disk graph, even if it does not have radio holes and the geometric location of the destination is known: Suppose the destination is in the center. With a wheel of  $\Theta(\sqrt{n})$  spikes of  $\Theta(\sqrt{n})$  length each, no node on the wheel can guess the right spike with probability better than  $\Theta(1/\sqrt{n})$ , so routing information is required for a constant stretch, but in order to compute the information needed for a constant stretch, the starting point  $w$  of the spike leading to the destination in the center needs to be identified, which requires  $\Omega(\sqrt{n})$  communication rounds.

► **Theorem 24.** *There is no deterministic (randomized) distributed algorithm that can, within  $o(\sqrt{n})$  rounds, compute a compact routing scheme that achieves (expected)  $o(\sqrt{n})$  stretch when only communicating over the unit-disk graph. This claim holds even when the algorithm can use geometric information<sup>6</sup> and the unit-disk graph has no radio-holes.*



■ **Figure 4** Lower bound graph (slightly adapted from Figure 8 in [23]).

## B Proof of Theorem 16

**Proof of Theorem 16.** Given some  $t \in V$ , the label  $\ell_t$  of  $t$  is the concatenation of the label  $\ell_{\Gamma,t}$  of the grid node which is in the same cell as  $t$  and  $\text{ID}(t)$ . Hence, the labelling  $\ell_G$  requires  $O(\log n)$  bits, given that the same is true for  $\ell_{\Gamma}$ . The information required to compute  $\rho_G(v, \ell_t)$  is composed of the knowledge of neighbors of  $v$  in  $G$  which includes the representative of the cell of  $v$  (due to Lemma 3) and the information required to evaluate  $\rho_{\Gamma}(v, \ell_{\Gamma,t})$ . Since nodes know their neighbors already as part of the problem input (local network equals the routing graph), we do not regard this as additional routing information. The information to evaluate  $\rho_{\Gamma}(v, \ell_{\Gamma,t})$  is  $O(1)$  bits by our presumption.

We continue with the correctness and the stretch of a path implied by  $\rho_G$ . Let  $s \neq t \in V$  be the current node and the target node respectively. Consider the case that  $\|s - t\| \leq 1$ , i.e., the nodes are adjacent. Then, according to Definition 15 rule (1) the packet is delivered directly to  $t$  which constitutes a correct and exact path.

<sup>6</sup> i.e., each node knows and can communicate its own location, the location of its neighbours, and each source will be given the location of its destination before routing.

Consider the case that  $\|s-t\| \geq 1$ . Let  $g_s, g_t \in V_\Gamma$  and  $r_s, r_g$  be the respective grid nodes and representatives of the cells of  $s, t$ . Let  $\Pi^*$  be the optimal  $g_s$ - $g_t$ -path in  $\Gamma$  implied by  $\rho_\Gamma$ . Then the path implied by  $\rho_G$  equals  $\Pi_{s,t} = \{\{s, r_s\}\} \cup \Pi_{r_s, r_t} \cup \{\{r_t, t\}\}$  from Definition 9, where  $\Pi_{r_s, r_t} := \bigcup_{\{g, g'\} \in \Pi^*} \Pi_{\{g, g'\}}$  and  $\Pi_{\{g, g'\}}$  is the representation of the grid edge  $\{g, g'\} \in \Pi^*$ . This is due to rules (2),(3) and (4).

By Theorem 10, we have that  $\text{dist}(\Pi_{s,t}) \leq |\Pi_{s,t}| \leq 36 \cdot \text{dist}_G(s, t)$ , implying a stretch of  $O(1)$ .

The runtime of pre-computing  $\mathcal{R}_G$  amounts to that of computing a representation  $R$  of  $\Gamma$ , which takes  $O(1)$  rounds due to Lemma 7. Note that in all four cases of the routing function  $\rho_G$  can be evaluated locally using the representation  $R$  of  $\Gamma$  from the pre-computation step, information about local neighbors in  $G$  and (local) evaluations of  $\rho_\Gamma$ . ◀

## C Additional Technical Details for Section 4

We provide the missing proof of Lemma 18 and additionally give an overview on some techniques for hybrid networks that are used by our tree labelling algorithm in Section 4. A more detailed description of these techniques can be found in [11, 16, 15].

### C.1 Proof of Lemma 18

**Proof of Lemma 18.** The runtime of  $O(1)$  rounds is clear, as each node  $v$  only needs to communicate for one round with its southern neighbor  $v_S$  in the portal. We provide arguments on why the construction is a tree. Since the cell polygon  $P'$  is simple (Lemma 5), the cells of vertices in a portal connect two points on the same polygonal boundary of  $P'$ . Thus, removing these cells disconnects  $P'$  and therefore removing the vertices of a portal from  $\Gamma$  disconnects  $\Gamma$ . This means the portal graph is acyclic, i.e., a tree. Since  $\Gamma$  is connected, the portal graph is connected as well. ◀

### C.2 Pointer Jumping

We show how to construct a network with diameter  $O(\log n)$  in time  $O(\log n)$  out of a simple line graph  $L$  with  $O(n)$  nodes. Assume each node  $v \in L$  knows its left and right neighbor in the line (except for the left- and rightmost node, who only know one neighbor). We let the nodes of  $L$  generate *shortcut edges* via *pointer jumping*: In the first round, each node  $v_i \in L$  that has two neighbors  $v_{i-1}, v_{i+1} \in L$  establishes the edge  $\{v_{i-1}, v_{i+1}\}$ . Whenever in each subsequent round, a node  $v \in L$  receives two new shortcut edges  $\{u, v\}, \{v, w\}$  in the previous round,  $v$  generates another shortcut edge  $\{u, w\}$ . It is easy to see that after  $O(\log n)$  rounds, no further shortcut edges are created and the resulting structure has diameter  $O(\log n)$ , thus implying the following lemma.

► **Lemma 25.** *Given a line  $L$  of  $O(n)$  nodes, setting up additional edges to obtain a structure  $L^+$  with diameter  $O(\log n)$  and degree  $O(\log n)$  takes  $O(\log n)$  rounds.*

Performing pointer jumping on each of our portals in the portal tree, the grid nodes within each portal  $\mathcal{P}$  are able to set up a structure on which they can quickly broadcast information to all grid nodes within  $\mathcal{P}$ . By doing so, we immediately obtain the following lemma.

► **Lemma 26.** *Any  $O(\log n)$ -bit message can be broadcast among all grid nodes within a single portal  $\mathcal{P}$  in  $O(\log |\mathcal{P}|)$  rounds.*

### C.3 Rooting Trees of Arbitrary Depth

Given a tree  $T$  of  $n$  nodes with arbitrary depth and constant node degree, we show how to root  $T$  at the node  $s$  with minimum identifier, such that every node in  $T$  is aware of its parent node. To do so, we adapt the well-known *Euler tour* technique to a distributed setting. Every node  $v \in T$  with neighbors  $v(0), \dots, v(\deg(v) - 1)$  (sorted in ascending order by their identifiers) simulates a virtual node  $v_i$  for each of its neighbor  $v(i)$ . We now connect all virtual nodes to a simple cycle  $C$  as follows. For every node  $v_i \in C$ , there is an edge  $(v_i, u_j) \in C$  such that  $u = v((i + 1) \bmod \deg(v))$  and  $v = u(j)$ . Therefore, each virtual node  $v_i$  that belongs to the node  $v$  with identifier  $id(v)$  is able to introduce itself to its predecessor in  $C$  by sending its *virtual identifier*  $\tilde{id}(v_i) := id(v) \circ i$  for all  $i \in [\deg(v)]$ , where  $\circ$  denotes the concatenation of two binary strings and  $[k] = \{0, \dots, k - 1\}$ . Since each node simulates only a constant number of virtual nodes, the number of virtual nodes in the cycle  $C$  is  $O(n)$ .

We first describe how to determine the virtual node  $s_i$  with minimal virtual identifier in  $O(\log n)$  rounds.<sup>7</sup> Note that the node  $s$  simulating  $s_0$  is then the node with minimal identifier. Consider the cycle  $C$  of virtual nodes and denote the edges of the cycle as *level-1* edges. Our algorithm works in multiple iterations. Initially, each virtual node  $v$  stores its own virtual identifier  $\tilde{id}(v)$  in some variable  $v.I$ . In the first iteration, each virtual node  $v$  does the following. In the first step,  $v$  sends  $v.I$  to its left neighbor in the cycle<sup>8</sup>. Upon receipt of a virtual identifier  $v.I$ , each node  $u$  updates its variable  $u.I$  to  $v.I$  in case that  $v.I < u.I$ . In the next step, each virtual node  $v$  introduces its left neighbor  $v_l$  to its right neighbor  $v_r$  to create the edge  $\{v_l, v_r\}$ , a level-2 edge. In each subsequent iteration, say the  $i$ -th iteration, each node  $v$  first sends  $v.I$  along with its own identifier via *all* of its level- $j$  edges (for all  $j \in \{1, \dots, i\}$ ) and then creates level- $(i + 1)$  edges, using its level- $i$  edges created in the previous iteration. Note that after the  $i$ -th iteration,  $2^i$  nodes are aware of  $v$ 's virtual identifier and thus have stored  $v$ 's virtual identifier in their variables  $u.I$ , in case  $v$ 's virtual identifier is the minimal virtual identifier among all of these nodes. We proceed in this manner until a virtual node  $v$  has received its own virtual identifier from its right neighbor in some iteration, as in this case all nodes have received  $v$ 's virtual identifier. This happens at the node  $s_0$  with minimum virtual identifier after  $O(\log n)$  rounds, because  $C$  contains  $O(n)$  virtual nodes. Thus, all that is left to do is to let  $s_0$  announce itself as the root of the tree by broadcasting a message on the cycle with the generated shortcuts, indicating the termination of the algorithm and announcing itself as the node with minimum virtual identifier. This takes another  $O(\log n)$  rounds. Then, each virtual node is now aware of the node  $s_0$  with minimum virtual identifier and therefore also of the node  $s$  with minimum identifier.

Now we want to root the tree  $T$  at  $s$ . The virtual node  $s_0$  starts broadcasting its virtual identifier via all of its outgoing edges to the left (including all of the generated shortcuts from before). During this broadcast, we keep track of the traversal distance of the message to be broadcasted, such that each virtual node is able to determine how many hops it is away from  $s_0$  in the cycle. A real node  $v$  can now determine its parent in the tree  $T$  by looking at its virtual node with minimum traversal distance to  $s_0$ . Let this node be the node  $v_i$  and let  $u_i$  be the predecessor of  $v_i$  in the cycle  $C$ . Then it is easy to see that  $u$  is the parent node of  $v$  in  $T$ , resulting in  $T$  getting rooted at  $s$  and implying the following lemma.

<sup>7</sup> The virtual node with minimal virtual identifier  $\tilde{id}(s_i) = id(s) \circ i$  is the node  $s_i$  with  $id(s) \leq id(v)$  for all  $v \in T$  and  $i = 0$ .

<sup>8</sup> The nodes may have different perceptions on which direction is left, but this is of no concern for our algorithm.

► **Lemma 27.** *Let  $T$  be a tree of  $n$  nodes with constant node degree.  $T$  can be rooted at the node  $s$  with minimal identifier within  $O(\log n)$  rounds.*

## C.4 Depth-First Search on Trees

Given a rooted tree  $T$  of  $n$  nodes with arbitrary depth and constant node degree, we compute for each node  $v \in T$  the preorder number  $l_v \in \mathbb{N}$  according to a depth-first search (DFS) of  $T$ . Let  $s$  be the root of  $T$ . As the first step, we perform the distributed Euler-Tour technique described in the previous section with the exception that the virtual node  $s_0$  refrains from introducing itself to its predecessor. It is easy to see that this results in the virtual nodes being arranged in a simple line  $L$  instead of a cycle.

Next, we apply the pointer jumping technique from Lemma 25 to transform  $L$  into a structure  $L^+$  with diameter  $O(\log n)$ . Through a single broadcast from the leftmost node  $s \in L^+$ , we are now able to compute a number  $l'_u$  for a virtual node  $u$  indicating the number of (real) nodes  $v$  for which at least one of  $v$ 's virtual nodes is left of  $u$  on the line  $L$ . Each real node  $u$  then sets  $l_u$  to the minimum value out of all  $l'_u$  values of its virtual nodes, which corresponds to  $u$ 's position in the DFS.

► **Lemma 28.** *Let  $T$  be a rooted tree of  $n$  nodes with constant node degree. A DFS on  $T$  where each node  $v \in T$  is assigned its number  $l_v \in \mathbb{N}$  in the DFS can be computed in  $O(\log n)$  rounds.*

## C.5 Computing the Maximum Preorder Number in a Rooted Tree

Assume we are given a rooted tree  $T$  of  $n$  nodes with arbitrary depth and constant node degree in which every node  $v \in T$  possesses a preorder number  $l_v \in \mathbb{N}$  according to a DFS. We compute for each node  $v \in T$  the maximum preorder number possessed by a node in  $v$ 's subtree, i.e., we compute  $r_v = \max\{l_u \in \mathbb{N} \mid u \in T(v)\}$ , where  $T(v)$  is the subtree of  $T$  with  $v$  as the root.<sup>9</sup>

Before we describe our algorithm, we let the nodes compute an upper bound of  $\log n$ , i.e., some value  $d = O(\log n)$ ,  $d \geq \log n$  as follows. We compute the line  $L$  via the Euler-tour described earlier on the rooted tree  $T$  and apply Lemma 25 on  $L$  to obtain the structure  $L^+$ . Then we perform a broadcast from the rightmost node  $u$  in  $L^+$  to the leftmost node  $s_0$  in  $L^+$ , where each message generated by the broadcast contains a counter that is incremented by 1 once the message is forwarded. The node  $s_0$  then maintains a variable  $d$  that contains the maximum counter received by  $s_0$ . Since  $L^+$  has diameter  $O(\log n)$ , the broadcast finishes after  $O(\log n)$  rounds. Once the broadcast is finished, it is easy to see that  $d = O(\log n)$ . The node  $s_0$  then broadcasts  $d$  to all nodes in  $L^+$ , such that after another  $O(\log n)$  rounds, each node knows  $d$ . Observe that  $d \geq \log D(T)$ , where  $D(T)$  is the depth of the tree  $T$ .

We are now ready to describe the algorithm for computing the values  $r_v$  for each node  $v \in T$ . Initially, each node  $v$  sets  $r_v$  to  $l_v$ . Denote the edges of  $T$  as *level-0* edges. The algorithm performs  $i = 1, \dots, d$  iterations, each iteration needing  $O(1)$  rounds. Iteration

<sup>9</sup> A more general approach to this problem is presented in [16, Lemma 4.12], where the goal is to compute the value of a distributive aggregate function for each node  $v$ 's own subtree. An aggregate function  $f$  is called *distributive* if there is an aggregate function  $g$  such that for any multiset  $S$  and any partition  $S_1, \dots, S_\ell$  of  $S$ ,  $f(S) = g(f(S_1), \dots, f(S_\ell))$ . Classical examples are MAX, MIN, and SUM. However, due to the generality of  $f$ , the authors had to make use of randomization, which results in a runtime of  $O(\log n)$ , w.h.p. for their algorithm. We present a deterministic  $O(\log n)$ -algorithm that is specifically tailored to the MAX function in this section.

$i$  works as follows at each node  $v \in T$ . First, if  $v$  has a level- $(i - 1)$  edge going up in the tree to some node  $u$ , then  $v$  sends  $r_v$  to  $u$ . Upon receipt of a value  $r_w$  from node  $w$  in the previous step,  $v$  updates  $r_v$  by setting  $r_v \leftarrow r_w$  and marks the edge  $\{v, w\}$ .<sup>10</sup> As the final step of the iteration,  $v$  checks whether it has a marked edge  $\{v, u\}$  going up the tree and a marked edge  $\{v, w\}$  going down the tree. If that is the case,  $v$  creates a level- $i$  edge  $\{u, w\}$  by introducing  $u$  to  $w$  and vice versa. If not, then  $v$  marks itself as *ready*.

Let  $T(v)$  be the subtree of  $T$  with  $v$  as the root and let  $w \in T(v)$  be the leaf node with maximum preorder number. Consider the unique path  $P$  up the tree from  $w$  to  $v$  in  $T(v)$ . It is easy to see that our algorithm transfers the preorder number  $l_w$  to all nodes on this path within  $\lceil \log k \rceil$  iterations, where  $k$  is the length of  $P$ , because in each iteration  $i$ , new level- $i$  shortcuts are added to the nodes on the path in a manner similar to the pointer-jumping approach from Section C.2. Therefore, once a node  $v$  has marked itself as ready in iteration  $i$ ,  $v$  has received the desired value for  $r_v$  in iteration  $i$ . As each node  $v \in T$  performs the algorithm in parallel, each node  $v$  has determined  $r_v$  after at most  $d = O(\log n)$  iterations (recall that  $d \geq \log D(T)$ ). Note that the node degree for each node  $v$  does not exceed  $O(\log n)$  throughout the algorithm, as in each iteration,  $v$ 's degree increases by at most 2.

We obtain the following lemma.

► **Lemma 29.** *Let  $T$  be a rooted tree of  $n$  nodes with constant node degree in which every node  $v \in T$  possesses a preorder number  $l_v \in \mathbb{N}$  according to a DFS on  $T$  starting at its root. Each node  $v \in T$  can compute the value  $r_v = \max\{l_u \in \mathbb{N} \mid u \in T(v)\}$ , where  $T(v)$  is the subtree of  $T$  with  $v$  as the root, within  $O(\log n)$  rounds.*

<sup>10</sup>Note that in the first iteration ( $i = 1$ ), a node  $v$  receives a value  $r_w$  from each of its child nodes  $w$ . It then just sets  $r_v$  to be the maximum value out of all received values  $r_w$ . It is easy to see that  $v$  receives at most one message in any subsequent iterations in this step.






# Efficient Assignment of Identities in Anonymous Populations

Leszek Gašieniec ✉ 🏠 

Department of Computer Science, University of Liverpool, UK

Jesper Jansson ✉ 🏠 

Graduate School of Informatics, Kyoto University, Japan

Christos Levcopoulos ✉ 🏠 

Department of Computer Science, Lund University, Sweden

Andrzej Lingas ✉ 🏠 

Department of Computer Science, Lund University, Sweden

---

## Abstract

We consider the fundamental problem of assigning distinct labels to agents in the probabilistic model of population protocols. Our protocols operate under the assumption that the size  $n$  of the population is embedded in the transition function. Their efficiency is expressed in terms of the number of states utilized by agents, the size of the range from which the labels are drawn, and the expected number of interactions required by our solutions. Our primary goal is to provide efficient protocols for this fundamental problem complemented with tight lower bounds in all the three aspects. W.h.p. (with high probability), our labeling protocols are silent, i.e., eventually each agent reaches its final state and remains in it forever, and they are safe, i.e., never update the label assigned to any single agent. We first present a silent w.h.p. and safe labeling protocol that draws labels from the range  $[1, 2n]$ . Both the number of interactions required and the number of states used by the protocol are asymptotically optimal, i.e.,  $O(n \log n)$  w.h.p. and  $O(n)$ , respectively. Next, we present a generalization of the protocol, where the range of assigned labels is  $[1, (1 + \varepsilon)n]$ . The generalized protocol requires  $O(n \log n / \varepsilon)$  interactions in order to complete the assignment of distinct labels from  $[1, (1 + \varepsilon)n]$  to the  $n$  agents, w.h.p. It is also silent w.h.p. and safe, and uses  $(2 + \varepsilon)n + O(n^c)$  states, for any positive  $c < 1$ . On the other hand, we consider the so-called pool labeling protocols that include our fast protocols. We show that the expected number of interactions required by any pool protocol is  $\geq \frac{n^2}{r+1}$ , when the labels range is  $1, \dots, n + r < 2n$ . Furthermore, we provide a protocol which uses only  $n + 5\sqrt{n} + O(n^c)$  states, for any  $c < 1$ , and draws labels from the range  $1, \dots, n$ . The expected number of interactions required by the protocol is  $O(n^3)$ . Once a unique leader is elected it produces a valid labeling and it is silent and safe. On the other hand, we show that (even if a unique leader is given in advance) any silent protocol that produces a valid labeling and is safe with probability  $> 1 - \frac{1}{n}$ , uses  $\geq n + \sqrt{\frac{n-1}{2}} - 1$  states. Hence, our protocol is almost state-optimal. We also present a generalization of the protocol to include a trade-off between the number of states and the expected number of interactions. Finally, we show that for any silent and safe labeling protocol utilizing  $n + t < 2n$  states, the expected number of interactions required to achieve a valid labeling is  $\geq \frac{n^2}{t+1}$ .

**2012 ACM Subject Classification** Theory of computation; Theory of computation  $\rightarrow$  Design and analysis of algorithms; Theory of computation  $\rightarrow$  Complexity theory and logic; Theory of computation  $\rightarrow$  Distributed computing models

**Keywords and phrases** population protocol, state efficiency, time efficiency, one-way epidemics, leader election, agent identities

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2021.12

**Funding** Research supported in part by VR grant 621-2017-03750 (Swedish Research Council).

**Acknowledgements** The authors are thankful to the anonymous referees for their valuable comments.



© Leszek Gašieniec, Jesper Jansson, Christos Levcopoulos, and Andrzej Lingas; licensed under Creative Commons License CC-BY 4.0

25th International Conference on Principles of Distributed Systems (OPODIS 2021).

Editors: Quentin Bramas, Vincent Gramoli, and Alessia Milani; Article No. 12; pp. 12:1–12:21

Leibniz International Proceedings in Informatics



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

## 1 Introduction

The problem of assigning and further maintaining unique identifiers for entities in distributed systems is one of the core problems related to network integrity. In addition, a solution to this problem is often an important preprocessing step for more complex distributed algorithms. The tighter the range that the identifiers are drawn from, the harder the assignment problem becomes.

In this paper we adopt the probabilistic population protocol model in which we study the problem of assigning distinct identifiers, which we refer to as *labels*, to all agents<sup>1</sup> The adopted model was originally intended to model large systems of agents with limited resources (state space) [4]. In this model the agents are prompted to interact with one another towards a solution of a shared task. The execution of a protocol in this model is a sequence of pairwise interactions between randomly chosen agents. During an interaction, each of the two agents: the *initiator* and the *responder* (the asymmetry assumed in [4]) updates its state in response to the observed state of the other agent according to the predefined (global) transition function. For more details about the population protocol model, see Appendix A.

Designing our population protocols for the problem of assigning unique labels to the agents (labeling problem), we make an assumption that the number  $n$  of agents is known in advance. Our protocols would also work if only an upper bound on the number of agents is known to agents. In fact, in such case the problem becomes easier as the range from which the labels are drawn is larger. In particular, if we do not have the limit on  $n$  we also do not have limit on the number of states to be used. More natural assumption is that such a limit is imposed. And indeed, there are plenty of population protocols which rely on the knowledge of  $n$  [12, 13].

Our labeling protocols include a preprocessing for electing a *leader*, i.e., an agent singled out from the population, which improves coordination of more complex tasks and processes. A good example is synchronization via phase clocks propelled by leaders. More examples of leader-based computation can be found in [5].

In the unique labeling problem adopted here, the number of utilized states needs to reflect the number of agents  $n$ . Also,  $\Omega(n \log n)$  is a natural lower bound on the expected number of interactions required to solve not only the labeling problem but any non-trivial problem by a population protocol. The main reason is that  $\Omega(n \log n)$  interactions are needed to achieve a positive constant probability that each agent is involved in at least one interaction [10].

Perhaps the simplest protocol for unique labeling in population networks is as follows [13] (cf. [11]). Initially, all agents hold label 1 which is equivalent with all agents being in state 1. In due course, whenever two agents with the same label  $i$  interact, the responder updates own label to  $i + 1$ . The advantage of this simple protocol is that it does not need any knowledge of the population size  $n$  and it utilizes only  $n$  states and assigns labels from the smallest possible range  $[1, n]^2$ . The severe disadvantage is that it needs at least a cubic in  $n$  number of interactions (getting rid of the last multiple label  $i$ , for all  $i = 1, \dots, n - 1$ , requires a quadratic number of interactions in expectation) to achieve the configuration in which the agents have distinct labels.

In the following two examples of protocols for unique labeling, we assume that the population size  $n$  is embedded in the transition function, such protocols are commonly used and known as non-uniform protocols [3], and one of the agents is distinguished as the leader, see leader based protocols [5].

<sup>1</sup> When the size of the label range is equal to the number of agents, the problem is also called *ranking* in the literature [12].

<sup>2</sup> We shall denote a range  $[p, \dots, q]$  by  $[p, q]$  from here on.

In the first of the two examples, we instruct the leader to pass labels  $n, n-1, \dots, 2$  to the encountered subsequently unlabeled yet agents and finally assign 1 to itself. The protocol uses only  $2n-1$  states ( $n$  states utilized by the leader and  $n-1$  states by other agents) and it assigns unique labels in the smallest possible range  $[1, n]$  to the  $n$  agents. Unfortunately, this simple protocol requires  $\Omega(n^2 \log n)$  interactions because as more agents get their labels, interactions between the leader and agents without labels become less likely. The probability of such an encounter drops from  $\frac{1}{n}$  at the beginning to  $\frac{1}{n(n-1)}$  at the end of the process.

By using randomization, we can obtain a much faster simple protocol as follows. We let the leader to broadcast the number  $n$  to all agents. It requires  $O(n \log n)$  interactions w.h.p.<sup>3</sup> [18]. When an agent gets the number  $n$ , it uniformly at random picks a number in  $[1, n^3]$  as its label. The probability that a given pair of agents gets the same label is only  $\frac{1}{n^3}$ . Hence, this protocol assigns unique labels to the agents with probability at least  $1 - \frac{1}{n}$ . It requires only  $O(n \log n)$  interactions w.h.p. The drawback is that it uses  $O(n^3)$  states and the large range  $[1, n^3]$ . This method also needs a large number of random bits independent for each agent.

Besides the efficiency and population size aspects, there are other deep differences between the three examples of labeling protocols. An agent in the first protocol never knows whether or not it shares its label with other agents. This deficiency cannot happen in the case of the second protocol but it takes place in the third protocol although with a small probability.

The labeling protocols presented in this paper are *silent* and *safe*. We say that a (non-necessarily labeling) protocol is silent if eventually each agent reaches its final state and remains in it forever. We say that a labeling protocol is safe if it never updates the label assigned to any single agent. While the concept of a silent population protocol is well established in the literature [12, 15], the concept of a safe labeling protocol is new. The latter property is useful in the situation when the protocol producing a valid labeling has to be terminated before completion due to some unexpected emergency or running out of time.

Observe that among the three examples of labeling protocols, only the second one is both silent and safe. The first example protocol is silent [12] but not safe. Finally, the third (probabilistic) one is silent and almost safe as it violates the definition only with small probability.

**Our contributions.** The primary objective of this paper is to provide efficient labeling protocols complemented with tight lower bounds in the aspects of the number of states utilized by agents, the size of the range from which the labels are drawn, and the expected number of interactions required by our solutions.

In particular, we provide positive answers to two following natural questions under the assumption that the number  $n$  of agents is known at the beginning.

1. Can one design a protocol for the labeling problem requiring an asymptotically optimal number of  $O(n \log n)$  interactions w.h.p., utilizing an asymptotically optimal number of  $O(n)$  states and an asymptotically minimal label range of size  $O(n)$  ?
2. Can one design a silent and safe protocol for the labeling problem utilizing substantially smaller number of states than  $2n$  and possibly the minimal label range  $[1, n]$  ?

We first present a population protocol that w.h.p. requires an asymptotically optimal number of  $O(n \log n)$  interactions to assign distinct labels from the range  $[1, 2n]$ . The protocol uses an asymptotically optimal number of  $O(n)$  states. We also present a more

---

<sup>3</sup> That is with the probability at least  $1 - \frac{1}{n^\alpha}$ , where  $\alpha \geq 1$  and  $n$  is the number of agents.

involved generalization of the protocol, where the range of assigned labels is  $[1, (1 + \varepsilon)n]$ . The generalized protocol requires  $O(n \log n / \varepsilon)$  interactions in order to complete the assignment of distinct labels from  $[1, (1 + \varepsilon)n]$  to the  $n$  agents, w.h.p. It uses  $(2 + \varepsilon)n + O(n^c)$  states, for any positive  $c < 1$ . Both protocols are silent w.h.p. and safe. Furthermore, we consider a natural class of population protocols for the unique labeling problem, the so-called *pool protocols*, including our fast labeling protocols. We show that for any protocol in this class that picks the labels from the range  $[1, n + r]$ , the expected number of interactions is  $\Omega(\frac{n^2}{r+1})$ .

Next, we provide a labeling protocol which uses only  $n + 5\sqrt{n} + O(n^c)$  states, for any positive  $c < 1$ , and the label range  $[1, n]$ . The expected number of interactions required by the protocol is  $O(n^3)$ . Once a unique leader is elected it produces a valid labeling and it is silent and safe. On the other hand, we show that (even if a unique leader is given in advance) any silent protocol that produces a valid labeling and is safe with probability larger than  $1 - \frac{1}{n}$ , uses at least  $n + \sqrt{\frac{n-1}{2}} - 1$  states. It follows that our protocol is almost state-optimal. In addition, we present a variant of this protocol which uses  $n(1 + \varepsilon) + O(n^c)$  states, for any positive  $c < 1$ . The expected number of interactions required by this variation is  $O(n^2/\varepsilon^2)$ , where  $\varepsilon = \Omega(n^{-1/2})$ . On the other hand, we show that for any silent and safe labeling protocol utilizing  $n + t < 2n$  states the expected number of interactions required to achieve a valid labeling is at least  $\frac{n^2}{t+1}$ .

All our labeling protocols include a preprocessing for electing a unique leader and assume the knowledge of the population size  $n$ . However, our almost state-optimal protocol (Single-Cycle protocol) can be made independent of  $n$  (see Section 4).

Our results are summarized in Tables 1 and 2.

**Main ideas of our protocols.** Our first fast labeling protocol roughly operates as follows. The leader initially has label 1 and a range of labels  $[2, n]$ . During the execution of the first phase, encountered unlabeled agents also get a label and an interval of labels that they can distribute among other agents. Upon a communication between a labeled agent that has a non-empty interval and an unlabeled agent, the latter agent gets a label from the interval and if the remaining part of the interval has length  $\geq 2$  then it is shared between the two agents. After  $O(n \log n)$  interactions, a sufficiently large fraction of agents is labeled and has no additional labels to distribute w.h.p. The leader counts its own interactions up to  $O(\log n)$  in order to trigger the second phase by broadcasting. In the latter phase, an agent with a label  $x$  and without a non-empty interval can distribute one additional label  $x + n$ . In the first phase of this protocol the labels from the range  $[1, n]$  are distributed rapidly among the agents. In the second phase the unlabeled agent still have a high chance of communicating with an agent that can distribute a label. Roughly, our second, generalized fast labeling protocol is obtained from the first one by constraining the set of agents that may distribute the labels  $x + n$  in the second phase to those having labels in the range  $[1, n\varepsilon]$ .

The main idea of the almost state-optimal labeling protocol (Single-Cycle protocol) is to use the leader and an auxiliary leader *nominated* by the leader to disperse the  $n$  labels jointly among the remaining free agents. The leader disperses the first and the auxiliary leader the second part of each individual label. When a free agent gets both partial labels, it combines them into its individual label and then informs the leaders about this. The two leaders operate in two embedded loops. For each of roughly  $\sqrt{n}$  partial labels of the leader, the auxiliary leader makes a full round of dispersing its roughly  $\sqrt{n}$  partial labels. In the generalized version of the protocol (k-Cycle protocol), the process is partially parallelized by letting the leader to form  $k$  pairs of dispensers, where each pair labels agents in a distinct range of size  $n/k$ .

■ **Table 1** Upper bounds on the number of states, the number of interactions and the range required by the labeling protocols presented in this paper. In Theorem 9,  $\varepsilon$  is  $\Omega(n^{-1})$  while in Theorem 12  $\Omega(n^{-0.5})$ .

Theorem	# states	# interactions	Range
Theorem 4	$O(n)$	$O(n \log n)$ w.h.p.	$[1, 2n]$
Theorem 9	$(2 + \varepsilon)n + O(n^c)$ , any $c < 1$	$O(n \log n / \varepsilon)$ w.h.p.	$[1, (1 + \varepsilon)n]$
Theorem 11	$n + 5 \cdot \sqrt{n} + O(n^c)$ , any $c < 1$	expected $O(n^3)$	$[1, n]$
Theorem 12	$(1 + \varepsilon)n + O(n^c)$ , any $c < 1$	expected $O(n^2 / \varepsilon^2)$	$[1, n]$

■ **Table 2** Lower bounds on the number of states or/and the number of interactions required by labeling protocols. (1) Any labeling protocol that is capable to produce a valid labeling. (2) The silent protocol in Theorem 14 (first part) is assumed to produce a valid labeling and be safe with probability greater than  $1 - \frac{1}{n}$ . (3) The silent protocol in Theorem 14 (2nd part) is assumed to produce a valid labeling and be safe with probability 1.

Protocol type	# states	# interactions	Theorem
any <sup>1</sup>	$n$	$\Omega(n \log n)$ w.h.p.	Theorem 13
silent, safe <sup>2</sup>	$n + \sqrt{\frac{n-1}{2}} - 1$	-	Theorem 14 (1st part)
silent, safe <sup>3</sup> , $n + t < 2n$ states	-	expected $\frac{n^2}{t+1}$	Theorem 14 (2nd part)
pool, range $[1, n + r]$	-	expected $\frac{n^2}{r+1}$	Theorem 17

**Related work.** There are several papers concerning labeling of processing units (also known as renaming or naming) in different communication models [14]. E.g., Berenbrink et al. [8] present efficient algorithms for the so-called loose and tight renaming in shared memory systems improving on or providing alternative algorithms to the earlier algorithms by Alistarh et al. [2, 1]. The loose renaming where the label space is larger than the number of units is shown to admit substantially faster algorithms than the tight renaming [1, 8].

The problem of assigning unique labels to agents has been studied in the model of population protocols by Beauquier et al. [7, 11]. In [11], the emphasis is on estimating the minimum number of states which are required by apparently non-safe protocols. In [7], the authors provide among other things a generalization of a leader election protocol to include a distribution of  $m$  labels among  $n$  agents, where  $m \leq n$ . In the special case of  $m = n$ , all agents will receive unique labels. No analysis on the number of interactions required by the protocol is provided in [7]. Their focus is on the feasibility of the solution, i.e., that the process eventually stabilizes in the final configuration. Their protocol seems inefficient in the state space aspect as it needs many states/bits to keep track of all the labels.

Doty et al. considered the labeling problem in [17] and presented a subroutine named “UniqueID” for it based on the technique of traversing a labeled binary tree and associating agents with nodes in the tree. The subroutine requires  $O(n \log n \log \log n)$  interactions.

The labeling problem has also been studied in the context of self-stabilizing protocols where the agents start in arbitrary (not predefined) states, see [12, 13]. In [13], Cai et al. propose a solution which coincides with our first example of labeling protocols presented in the introduction. In a very recent work [12], Burman et al. study both slow and fast labeling protocols, the latter utilizing an exponential number of states. The protocols in both papers require the exact knowledge of  $n$ . The work [12] focuses on self-stabilizing protocols which cannot be safe by definition. It is more proper to compare our protocols with the initialized version of the protocols in [12]. E.g., the leader-driven initialized (silent) ranking protocol

in [12] (see Lemma 4.1) requires  $O(n^2)$  interactions, uses  $O(n)$  states and it is safe. An analogous variant of the fast ranking protocol from [12] requiring  $O(n \log n)$  interactions and an exponential number of states is also safe but not silent. The known labeling protocols are summarized in Table 3 in Appendix B.

The most closely related problem more studied in the literature is that of counting the population size, i.e., the number of agents. It has been recently studied by Aspnes et al. in [6] and Berenbrink et al. in [10]. We assume that the population size is initially known. Alternatively, it can be computed by using the protocol counting the exact population size given in [10]. The aforementioned protocol computes the population size in  $O(n \log n)$  interactions w.h.p., using  $\tilde{O}(n)$  states. Another possibility is to use the protocol computing the approximate population size, presented in [10]. The latter protocol requires  $O(n \log^2 n)$  interactions to compute the approximate size w.h.p., using only a poly-logarithmic number of states. For references to earlier papers on protocols for counting or estimating the population size, in particular the papers that introduced the counting problem and that include the original algorithms on which the improved algorithms of Berenbrink et al. are based, see [10].

All our protocols include a preprocessing for electing a unique leader and its synchronization with the proper labeling protocol (e.g., see the proof of Theorem 4). There is a vast literature on population protocols for leader election [9, 16, 18, 19]. For our purposes, the most relevant is the protocol that elects a unique leader from a population of  $n$  agents using  $O(n \log n)$  interactions and  $O(n^c)$  many states, for any positive constant  $c < 1$ , w.h.p., described in [10, 16] (see also Fact 7). The newest results elaborate on state-optimal leader election protocols utilizing  $O(\log \log n)$  states. These include the fastest possible protocol [9] based on  $O(n \log n)$  interactions in expectation, and a slightly slower protocol [19] requiring  $O(n \log^2 n)$  interactions with high probability.

Our population protocols for unique labeling use also the known population protocol for (one-way) epidemics, or broadcasting. It completes spreading a message in  $\Theta(n \log n)$  interactions w.h.p. and it uses only two states [18] (see also Fact 4).

**Organization of the paper.** In the next section, we provide basic facts on probabilistic inequalities and population protocols for broadcasting, counting and leader election. In Section 3, we present our fast silent w.h.p. and safe protocol for unique labeling in the range  $[1, 2n]$  and its generalization to include the range  $[1, n(1 + \varepsilon)]$ . Section 4 is devoted to the almost state-optimal, roughly silent and safe protocol with the label range  $[1, n]$  and its variation. Section 5 presents lower bounds on the number of states or the number of interactions for silent, safe and the so-called pool protocols for unique labeling. We conclude with Final remarks.

## 2 Preliminaries

### Probabilistic bounds.

► **Fact 1** (The union bound). *For a sequence  $A_1, A_2, \dots, A_r$  of events,  $\text{Prob}(A_1 \cup A_2 \cup \dots \cup A_r) \leq \sum_{i=1}^r \text{Prob}(A_i)$ .*

► **Fact 2** (multiplicative Chernoff lower bound). *Suppose  $X_1, \dots, X_n$  are independent random variables taking values in  $\{0, 1\}$ . Let  $X$  denote their sum and let  $\mu = E[X]$  denote the sum's expected value. Then, for any  $\delta \in [0, 1]$ ,*

*$\text{Prob}(X \leq (1 - \delta)\mu) \leq e^{-\frac{\delta^2 \mu}{2}}$  holds. Similarly, for any  $\delta \geq 0$ ,  $\text{Prob}(X \leq (1 + \delta)\mu) \leq e^{-\frac{\delta^2 \mu}{2 + \delta}}$  holds.*

► **Fact 3** ([18]). *For all  $C > 0$  and  $0 < \delta < 1$ , during  $Cn \log n$  interactions, with probability at least  $1 - n^{-O(\delta^2 C)}$ , each agent participates in at least  $2C(1 - \delta) \log n$  and at most  $2C(1 + \delta) \log n$  interactions.*

**Broadcasting, counting and leader election.** We shall refer to the following broadcast process which can be completed during  $\Theta(n \log n)$  interactions w.h.p. Each agent is either in a state of M-type (got the message) or in a state of  $\neg$ M-type. Whenever an agent in a state of M-type interacts with an agent in a state of  $\neg$ M-type, the latter changes its state to a state of M-type (gets the message). The process starts when the first agent gets the message and completes when all agents have the message.

► **Fact 4.** *There is a constant  $c_0$ , such that for  $c \geq c_0$ , the broadcast process completes in  $cn \log n$  interactions with probability at least  $1 - n^{-\Theta(c)}$ .*

Berenbrink et al. [10] obtained among other things the following results on counting the population size, i.e., the number of agents.

► **Fact 5.** *There is a protocol for a population of an unknown number  $n$  of agents such that w.h.p., after  $O(n \log^2 n)$  interactions the protocol stabilizes and each agent holds the same estimation of the population size which is either  $\lceil \log n \rceil$  or  $\lfloor \log n \rfloor$ . The protocol uses  $O(\log^2 n \log \log n)$  states.*

► **Fact 6.** *There is a protocol for a population of an unknown number  $n$  of agents such that w.h.p., after  $O(n \log n)$  interactions the protocol stabilizes and each agent holds the exact population size. The protocol uses  $\tilde{O}(n)$  states.*

There is a vast literature on population protocols for leader election [18]. For our purposes, the following fact will be sufficient. Its idea is to start leader election with a subprotocol of [19] that elects a junta of substantially sublinear in  $n$  number of leaders. The junta is formed using  $O(n \log n)$  interactions. Then, when state space of size  $n^c$  is available,  $c < 1$ , only a constant number of rounds of leader elimination is needed, each requiring  $O(n \log n)$  interactions. For more details, see [10, 16].

► **Fact 7.** *There is a protocol that elects a unique leader from a population of  $n$  agents using  $O(n \log n)$  interactions and  $O(n^c)$  many states, for any positive constant  $c < 1$ , w.h.p. [10, 16].*

### 3 Labeling with asymptotically optimal number of interactions, nearly optimal number of states and range

In this section, we provide a silent w.h.p. and a safe labeling protocol that assigns unique labels from the range  $[1, 2n]$  to  $n$  agents in  $O(n \log n)$  interactions w.h.p. Then, we generalize the protocol to include the range  $[1, (1 + \varepsilon)n]$ , where  $\varepsilon$  does not have to be a constant; it can even be as small as  $O(n^{-1})$ . We show that the generalized protocol assigns unique labels from  $[1, (1 + \varepsilon)n]$  in  $O(n \log n / \varepsilon)$  interactions w.h.p. In the first protocol, the agents use  $O(n)$  states, in the second protocol only  $(2 + \varepsilon)n + O(n^c)$  states, for any positive  $c < 1$ .

**Range  $[1, 2n]$ .** The protocol runs in two main phases preceded by a leader election preprocessing. The idea of the first phase resembles that of load balancing [10], the difference is that tokens (in our case labels and interval sub-ranges) are distinct.

At the beginning of the first phase, the leader assigns the label 1 and also temporarily the interval  $[2, n]$  to itself. Next, whenever two agents interact, one with label and a temporarily assigned interval  $[q, r]$  where  $r > q$  and the other without label, the former agent shrinks its interval to  $[q, \lfloor \frac{q+r}{2} \rfloor]$  and it gives away the label  $\lfloor \frac{q+r}{2} \rfloor + 1$  and if  $\lfloor \frac{q+r}{2} \rfloor + 2 \leq r$  also the sub-interval  $[\lfloor \frac{q+r}{2} \rfloor + 2, r]$  to the latter agent. Furthermore, whenever an agent with label and a temporarily assigned singleton interval  $[q, q]$  interacts with an agent without label, the former agent cancels its interval and gives the label  $q$  to the latter agent. In the remaining cases, interactions have no effect. Note that during the first phase a sub-tree of the binary tree of the partition of the start interval  $[1, n]$  with  $n$  leaves determined by the protocol rules is formed; see Fig. 1 in Appendix C. Also observe that when an agent at an intermediate node of the tree interacts with an agent without label then the former agent migrates to the left child of the node while the latter agent lands at the right child of the node.

In the second phase, when an agent with a label  $i \in [1, n]$  at a leaf of the tree interacts with an agent without label for the first time then the latter agent gets the label  $i + n$ . Interactions between agents (if any) at intermediate nodes of the tree and agents without labels are defined as in the first phase.

The following lemmata are central in showing that  $O(n \log n)$  interactions are sufficient w.h.p. to implement our protocol.

► **Lemma 1.** *There is a constant  $c$  such that after  $cn \log n$  interactions in the first phase the number of agents without labels drops below  $n/4$  w.h.p.*

**Proof.** The proof is by contradiction. Suppose that a set  $F$  of at least  $n/4$  agents without labels survives at least  $cn \log n$  interactions, where the constant  $c$  will be specified later.

Consider first the leader agent starting with the interval  $[2, n]$  during the aforementioned interactions. When the agent interacts with an agent without label its interval is roughly halved. We shall call such an interaction a success. The probability of success is at least  $\frac{1}{4n}$ . The expected number of successes is at least  $\frac{c}{4} \log n$ . By using Chernoff multiplicative bound given in Fact 2, we can set  $c$  to enough large constant so the probability of at least  $\log_2 n + 1$  successes will be at least  $1 - \frac{1}{n^2}$ . This means that the leader will end up without any interval with so high probability during the  $cn \log n$  interactions. The leader chooses the leftmost path in the binary partition tree of the start interval  $[1, n]$ . Consider an arbitrary path  $P$  from the root to a leaf in the tree. Note that several agents during distinct interactions can appear on the path. Define as a success an interaction in which an agent currently on  $P$  interacts with an agent without label. The expected number of successes is again at least  $\frac{c}{4} \log n$  and again we can conclude that there are at least  $\log_2 n + 1$  successes with probability at least  $1 - \frac{1}{n^2}$ . Simply, the probabilities of interacting with an agent without label are the same for all agents with labels, i.e., on some paths in the tree. Another way to argue is that the leader could make other decisions as to which roughly half of interval to preserve and the path choice. By the union bound (Fact 1), we conclude that all the  $n$  paths from the root to the leaves in the tree could be developed during the  $cn \log n$  interactions, so all agents would get a label, with probability at least  $1 - \frac{1}{n}$ . We obtain a contradiction with the so long existence of the set  $F$ . ◀

► **Lemma 2.** *If the second phase starts after  $cn \log n$  interactions, where  $c$  is the constant from Lemma 1, then only  $O(n \log n)$  interactions are needed to assign labels in  $[1, 2n]$  to the remaining agents without labels, w.h.p.*

**Proof.** The number of agents without labels at the beginning of the second phase is at most  $n/4$  w.h.p. Hence, at the beginning of this phase the number of agents with labels is at least  $\frac{3}{4}n$  w.h.p. An agent with label  $i \leq n$  at a leaf of the tree can give the label  $i + n$  to an agent



without label only once. Since this can happen at most  $\frac{n}{4}$  times, the number of agents with labels in  $[1, n]$  that can give a label is always at least  $\frac{n}{2}$  w.h.p. We conclude that for an agent without label the probability of an interaction with an agent that can give a label is at least almost  $\frac{1}{2n}$ . Hence, after each  $O(n)$  interactions the expected number of agents without label halves. It follows that the expected number of such interactions rounds is  $O(\log n)$ . Consequently, the number of the rounds is also  $O(\log n)$  w.h.p. by Chernoff bound (Fact 2).

An alternative way to obtain the  $O(n \log n)$  bound on the number of interactions w.h.p. is to use Fact 3 with  $C = O(\frac{1}{1/2})$  and  $\delta = \frac{1}{2}$ . Then, each agent will interact with at least  $C \log n$  agents w.h.p. during  $Cn \log n$  interactions. Consequently, the probability that a given agent does not interact with any agent that can give a label during the aforementioned interactions is  $(1 - \frac{1}{2})^{O(2 \log n)}$ . Hence, by picking enough large  $C$ , we conclude that each agent (in particular without label) will interact with at least one agent that can give a label during the  $Cn \log n$  interactions w.h.p. ◀

► **Lemma 3.** *During both phases, no pair of agents gets the same label.*

**Proof.** The uniqueness of the label assignments in the first phase follows from the disjointness of the labels and intervals assigned to agents before and after each interaction. This argument also works for the labels not exceeding  $n$  assigned later in the second phase. Finally, the uniqueness of the labels of the form  $i + n$  follows from the uniqueness of the labels of the agents passing these labels. ◀

► **Theorem 4.** *There is a safe protocol for population of  $n$  agents that w.h.p. assigns unique labels in the range  $[1, 2n]$  to the agents equipped with  $O(n)$  states in  $O(n \log n)$  interactions. The protocol is also silent w.h.p.*

**Proof.** Under the assumption that the leader election preprocessing provides a unique leader, the correctness of label assignment in both phases w.h.p. and the fulfilling of the definition of a silent and safe protocol follows from Lemmata 1, 2, and 3 and the specification of the protocol, respectively.

For the purpose of the leader election preprocessing, we use the simple leader election protocol using  $O(n \log n)$  interactions and  $O(n^c)$  states, for any positive constant  $c$ , described in [10, 16] (Fact 7). The phase clock (based on junta of leaders) from [19] is also formed in  $O(n \log n)$  interactions, using  $O(\log \log n)$  states and we use this clock to count the required (by the simple leader election protocol) time  $\Omega(n \log n)$ . When this time is reached on the clock we switch from leader election to our proper labeling protocol. The two aforementioned processes can be run simultaneously, resulting in additional state usage  $O(n^c \log \log n)$  (still fine for our needs). Thus, the leader election preprocessing and its synchronization with the proper labeling protocol in two phases add  $O(n \log n)$  interactions and  $o(n)$  states w.h.p. It provides a unique leader w.h.p. It follows that w.h.p. the whole protocol provides a correct labeling, it is silent and safe. In fact, we can make it safe (with probability 1) by prohibiting agents to change or get rid of an assigned label. Note that this constraint does not affect the operation of the protocol when a unique leader is provided by the preprocessing.

Both phases require  $O(n \log n)$  interactions w.h.p. by Lemmata 1, 2.

To put the two phases described in Lemmata 1, 2 together, we let the leader agent to count its interactions. When the number of interactions of the leader in the first phase exceeds an appropriate multiplicity of  $\log n$ , the total number of interactions in the first phase achieves the required lower bound from Lemma 1 w.h.p. by Fact 3. Therefore, then the leader starts broadcasting the message on the transition to the second phase to the other agents. By Fact 4, the broadcasting increases the number of interactions only by  $O(n \log n)$  w.h.p. (The leader can also stop the second phase in a similar fashion.)

## 12:10 Efficient Assignment of Identities in Anonymous Populations

To save on the number of states, instead of having states corresponding to all possible sub-intervals of  $[1, n]$ , we consider states corresponding to the nodes of the interval partition tree (see Fig. 1 in Appendix C) whose sub-tree is formed in the first phase. More precisely, we associate two states with each intermediate node of the binary tree on  $n$  leaves and  $n - 1$  intermediate nodes. They indicate whether or not the agent at the intermediate node has already received the message about the transition to the second phase. Next, we associate four states to each leaf of the tree. They indicate similarly whether or not the agent at the leaf has already received the phase transition message and whether or not the agent has already passed a label to an agent without label in the second phase, respectively. With each label in the range  $[n + 1, 2n]$ , we associate only a single state. Additionally, there are  $O(\log n)$  states used by the leader to count interactions in order to start the second phase. Recall also that the leader election preprocessing requires  $o(n)$  additional states. Thus the total number of states does not exceed  $2n + 4n + n + o(n)$ . ◀

By combining the protocol of Theorem 4 with that of Berenbrink et al. for exact counting the population size (Fact 6), we obtain the following corollary on unique labeling when the population size is unknown to agents initially.

► **Corollary 5.** *There is a protocol for a population of  $n$  agents that assigns unique labels in the range  $[1, 2n]$  to the agents initially not knowing the number  $n$ , equipped with  $\tilde{O}(n)$  states, in  $O(n \log n)$  interactions w.h.p.*

**Proof.** We run first the protocol for exact counting (Fact 6) and then our protocol for unique labeling (Theorem 4) using the leader elected by the counting protocol. We can synchronize the three protocols in a similar fashion as we synchronized the two phases of our protocol additionally using  $O(n \log n)$  interactions and  $O(\log n)$  states. ◀

By using the method of approximate counting from [10] (Fact 5) instead of that for exact counting (Fact 6), we can decrease the number of states to  $O(n)$  at the cost of increasing the label range to  $[1, 8n]$  and the number of interactions required to  $O(n \log^2 n)$ .

**Range  $[1, (1 + \varepsilon)n]$ .** The new protocol is obtained by the following modifications in the previous one. The leader which counts the number of own interactions starts broadcasting the phase transition message when the number of agents without labels drops below  $n\varepsilon/4$  w.h.p. (see Lemma 6). The information about the transition to the second phase affects only the agents at the leaves of the interval partition tree, corresponding to labels in  $[1, n\varepsilon]$ . When they get the message about the phase transition, they know that they can pass a label which is the sum of their own label and  $n$  to the first agent without label they interact with. For this reason, only the agents at the leaves corresponding to labels in  $[1, n\varepsilon]$  as well as the agents that are at the nodes that are ancestors of the aforementioned leaves participate in the broadcasting of the phase transition message. (Observe that the number of agents at these ancestors is  $O(n\varepsilon)$  and an agent at such an ancestor also has a label in  $[1, n\varepsilon]$ .) In the second phase, besides the agents at the leaves corresponding to labels in  $[1, n\varepsilon]$  and the agents without labels, also the agents at the intermediate nodes of the tree (if any) can really interact, in fact as in the first phase.

The following generalization of Lemma 1 is straightforward; see Appendix D for the proof.

► **Lemma 6.** *Let  $c$  be the constant from the statement of Lemma 1. During  $cn \log n/\varepsilon$  interactions in the first phase the number of agents without label drops below  $n\varepsilon/4$  w.h.p.*

Having Lemma 6, we can easily generalize Lemma 2 to the following one; see Appendix D for the proof.

► **Lemma 7.** *If the second phase starts after  $cn \log n/\varepsilon$  interactions, where  $c$  is the constant from Lemmata 1, 6, then only  $O(n \log n/\varepsilon)$  interactions are needed to assign labels in  $[1, (1 + \varepsilon)n]$  to the remaining agents without labels, w.h.p.*

We also need the following auxiliary lemma on broadcasting constrained to a subset of agents; see Appendix D for the proof.

► **Lemma 8.** *The leader can inform  $\Theta(n\varepsilon)$  agents with labels not exceeding  $O(n\varepsilon)$  about the phase transition using only these agents in  $O(n \log n/\varepsilon)$  interactions.*

The proof of the following theorem is analogous to that of Theorem 4 with Lemmata 1, 2 replaced by Lemmata 6, 7.

► **Theorem 9.** *Let  $\varepsilon > 0$ . There is a silent w.h.p. and safe protocol for a population of  $n$  agents that assigns unique labels in the range  $[1, (1 + \varepsilon)n]$  to  $n$  agents equipped with  $(2 + \varepsilon)n + O(n^c)$  states, for any positive  $c < 1$ , in  $O(n \log n/\varepsilon)$  interactions w.h.p.*

**Proof.** Under the assumption that the leader election preprocessing provides a unique leader, the correctness of the label assignment in both phases w.h.p. and the fulfillment of the definition of a silent and safe protocol follow from Lemmata 6, 7, and 8 by the same arguments as in the proof of Theorem 4.

The leader election preprocessing and its synchronization with the proper labeling protocol require  $O(n \log n)$  interactions and  $o(n^c)$  states, for  $c < 1$ , w.h.p. as described in the proof of Theorem 4. Analogously, it follows that w.h.p. the whole protocol provides a valid labeling, it is silent and safe. Again, it can be transformed to a safe protocol by prohibiting agents to change or get rid of an assigned label.

By Lemmata 6, 7, both phases require  $O(n \log n/\varepsilon)$  interactions w.h.p. The broadcasting about the phase transition starts when the number of agents without labels in the first phase drops below  $n\varepsilon/4$  w.h.p. By Lemma 8, it requires  $O(n \log n/\varepsilon)$  interactions w.h.p. since only the  $\Theta(n\varepsilon)$  agents in states corresponding to labels in  $[1, n\varepsilon]$  are involved in it.

The estimation of the number of needed states is more subtle than in Theorem 4. With each intermediate node of the interval partition tree that does not correspond to a label in  $[1, n\varepsilon]$  (equivalently, that is not an ancestor of a leaf corresponding to a label in  $[1, n\varepsilon]$ ), we associate a single state. (Recall here that if an agent at an intermediate node of the tree encounters an agent without label then the former agent moves to the left child of the node.) With each intermediate node corresponding to a label in  $[1, n\varepsilon]$ , we associate two states. They indicate whether or not the agent at the node has already got the message about phase transition. Next, with each leaf of the tree corresponding to a label  $i$  in  $[1, n\varepsilon]$ , we associate four states. They indicate whether or not the agent at the leaf has already got the message about the phase transition, and whether or not the agent has already passed the label  $i + n$  to some agent without label, respectively. To each of the remaining leaves, we associate only a single state.

We also need  $O(\log n/\varepsilon)$  additional states for the leader to count the number of own interactions in order to start broadcasting the message on transition to phase two at a right time step. In fact, we can get rid of the  $O(\frac{1}{\varepsilon})$  factor here by letting the leader to count approximately each  $\Theta(1/\varepsilon)$  interaction. Simply, the leader can count only interactions with agents which have got labels not exceeding  $O(\varepsilon n)$ .

Finally, we have  $n\varepsilon$  states corresponding to the labels in  $[n + 1, (1 + \varepsilon)n]$ . Thus, totally only  $(2 + O(\varepsilon))n + O(n^c)$  states, for any positive  $c < 1$ , are sufficient. To get rid of the constant factor at  $\varepsilon$ , it is sufficient to run the protocol for a smaller  $\varepsilon' = \Omega(\varepsilon)$ . It does not change the asymptotic upper bound on the number of required interactions w.h.p. and even it decreases the range of the labels. ◀

## 12:12 Efficient Assignment of Identities in Anonymous Populations

Note that  $\varepsilon$  in Theorem 9 does not have to be a constant; it can even be as small as  $O(n^{-1})$ .

By combining the protocol of Theorem 9 with that of Berenbrink et al. for exact counting the population size (Fact 6), we obtain the following corollary on unique labeling when the population size is unknown to agents initially. The proof is analogous to that of Corollary 5.

► **Corollary 10.** *Let  $\varepsilon > 0$ . There is a protocol for a population of  $n$  agents that assigns unique labels in the range  $[1, (1 + \varepsilon)n]$  to the agents initially not knowing the number  $n$ , equipped with  $\tilde{O}(n)$  states in  $O(n \log n / \varepsilon)$  interactions w.h.p.*

### 4 State- and range-optimal labeling

In this section we propose and analyze state-optimal protocols, which are silent and safe once a unique leader is elected, and utilize labels from the smallest possible range  $[1, n]$ . We assume the number of agents  $n$  to be known. We propose such a labeling protocol *Single-Cycle* which utilizes  $n + 5\sqrt{n} + O(n^c)$  states, for any positive  $c < 1$ , and the expected number of interactions required by the protocol is  $O(n^3)$ . We show in Section 5 that any silent and safe labeling protocol requires  $n + \sqrt{\frac{n-1}{2}} - 1$  states, see Theorem 14. Thus, our protocol is almost state-optimal. Finally, we propose a partial parallelization of *Single-Cycle* protocol called *k-Cycle* protocol which utilizes  $(1 + \varepsilon)n$  states and  $O((n/\varepsilon)^2)$  interactions for  $\varepsilon = \Omega(n^{-1/2})$ .

**Labeling protocol.** The state efficient labeling protocol starts from a preprocessing electing a unique leader. Its main idea is to use two agents: the initial leader  $A$  and a nominated (by  $A$ ) agent  $B$ , as partial *label dispensers*. These two agents jointly dispense unique labels for the remaining *free* (non-labeled yet) agents in the population where agent  $A$  dispenses the first and agent  $B$  the second part of each individual label. For the simplicity of presentation, we assume that  $n$  is a square of some integer. During execution of the protocol agent  $A$  uses partial labels  $\mathbf{label}(a) \in \{0, \dots, \sqrt{n} - 1\}$  and  $B$  uses partial labels  $\mathbf{label}(b) \in \{1, \dots, \sqrt{n}\}$ . The two dispensers label every agent by a unique pair of partial labels  $(\mathbf{label}(a), \mathbf{label}(b))$  where the combination  $(i, j)$  is interpreted as the integer label  $i \cdot \sqrt{n} + j$ . The protocol first labels all *free* (different to dispensers unlabeled) agents and eventually gives labels  $(0, 2)$  to agent  $B$  and  $(0, 1)$  to agent  $A$ .

In a nutshell, the labeling process is based on a single cycle of interactions between dispensers  $A$  and  $B$  and the free agents. Agent  $A$  awaits an interaction with a free agent  $F$  when  $A$  dispenses to  $F$  its current partial label  $\mathbf{label}(a)$ . Now  $F$  awaits an interaction with  $B$  in order to receive the second part of its label. And when this happens agent  $F$  concludes with the combined label and agent  $B$  awaits an interaction with  $A$  to inform that the next free agent needs to be labeled. On the conclusion of this interaction if  $\mathbf{label}(b) > 1$  agent  $B$  adopts new partial label  $\mathbf{label}(b) - 1$ , otherwise  $B$  adopts  $\mathbf{label}(b) = \sqrt{n}$  and agent  $A$  adopts new label  $\mathbf{label}(a) - 1$ . The only exception is when  $\mathbf{label}(a) = 0$  and  $\mathbf{label}(b) = 2$  when agent  $B$  adopts label  $(0, 2)$  and agent  $A$  adopts label  $(0, 1)$  and both agents conclude the labeling process. For more details, see the definition of the transition function in Appendix E.

► **Theorem 11.** *Single-cycle utilizes  $n + 5 \cdot \sqrt{n} + O(n^c)$  states, for any positive  $c < 1$ , and the minimal label range  $[1, n]$ . The expected number of interactions required by the protocol is  $O(n^3)$ . Once a unique leader is elected, it produces a valid labeling of the  $n$  agents and it is silent and safe.*

**Proof.** Assume that the leader election preprocessing provides a unique leader. Then, the protocol is silent and safe by its definition. All  $ll$  labels are dispensed in the sequential manner and the labeling process concludes when the two dispensers finalize their own labels.

In particular, as soon as the two dispensers  $A$  and  $B$  are established they operate in a short cycle formed of steps  $C1, C2$  and  $C3$  labeling one by one all free agents in the population. One can observe that the sequence of cycles mimics the structure of two nested loops where the external loop iterates along the partial labels of  $A$  and the internal one along partial labels of  $B$ . In total, we have  $n - 2$  iterations where the expected number of interactions required by each iteration is  $O(n^2)$ . Thus one can conclude that the expected number of interactions required by the whole labeling process but for the leader election preprocessing is  $O(n^3)$ . By the definition of the protocol the range of assigned labels is  $[1, n]$ . Finally, as indicated earlier in this section the number of states utilized by the protocol but for the leader election preprocessing is equal to  $n + 5 \cdot \sqrt{n} + 4$ .

The leader election preprocessing and its synchronization with the proper labeling protocol require additional  $O(n \log n)$  interactions and additional  $o(n^c)$  states, for  $c < 1$ , w.h.p. as described in the proof of Theorem 4. ◀

Observe that when the exact value of  $n$  is embedded in the transition function on the conclusion all agents become dormant, i.e., they stop participating in the labeling process. One could redesign the protocol such that the labels are dispensed by  $A$  and  $B$  in the increasing order using a diagonal method, e.g.,  $(0, 0)$ ,  $(0, 1)$ ,  $(1, 0)$ ,  $(0, 2)$ ,  $(1, 1)$ ,  $(2, 0)$ ,  $(0, 3)$ ,  $(1, 2)$ ,  $(2, 1)$ ,  $(3, 0)$  etc., where agent  $A$  gets label  $(0, 0)$ , agent  $B$  gets label  $(0, 1)$ , the first labeled free agent gets  $(1, 0)$ , the second  $(0, 2)$ , then  $(1, 1)$  and  $(2, 0)$ , when  $A$  and  $B$  start using the next diagonal, etc. Each pair  $(i, j)$  is interpreted as  $(i + j)(i + j + 1)/2 + i$ , e.g.,  $(0, 1) = 1$ ,  $(0, 2) = 3$ ,  $(0, 3) = 6$  and in general  $(0, j) = j(j + 1)/2$ ,  $(1, j - 1) = j(j + 1)/2 + 1$ ,  $(1, j - 2) = j(j + 1)/2 + 2, \dots, (j, 0) = j(j + 1)/2 + j = (j + 1)(j + 2)/2 - 1 = (0, j + 1) - 1$ . In this case the size of the population does not need to be known in advance, however, the two dispensers will never stop searching for free agents yet to be labeled.

**Faster Labeling.** We observe that one can partially parallelize *Single-Cycle* protocol by instructing leader  $A$  to form  $k$  pairs of dispensers where each pair labels agents in a distinct range of size  $n/k$ . In such case the new  $k$ -cycle protocol requires extra  $2k$  states to allow leader  $A$  initialize the labeling process (create two dispensers) in all  $k$  cycles. Thus the total number of states is bounded by  $n + 2k + k \cdot (5\sqrt{n/k} + 4) = n + 6k + 5k \cdot \sqrt{n/k} < n + 6(k + \sqrt{nk}) < n + 12\sqrt{nk}$ , as  $k < \sqrt{nk}$ , plus the number of states required by the leader election preprocessing. We use the same method for the leader election preprocessing and its synchronization with the proper labeling protocol described in the proof of Theorem 4. Analogously, it adds  $O(n \log n)$  interactions and  $O(n^c)$  states, for any positive  $c < 1$ . As we need to pick  $k$  for which  $n + 12\sqrt{nk} \leq n + n\varepsilon$  we conclude that  $k \leq n\varepsilon^2/144$ .

One can show that for  $k = n\varepsilon^2/144$ , the expected number of interactions required by the  $k$ -cycle protocol is  $O(n^2/\varepsilon^2)$ . Note that in order to initialize  $k$  cycles the leader  $A$  has to communicate with  $2k - 1$  free agents. As  $k$  is at most a small fraction of  $n$  during the search for dispensers for each cycle the number of free agents is always greater than  $n/2$  (in fact it is very close to  $n$ ). Thus the probability of forming a new dispenser during any interaction is greater than  $1/2n$ , i.e., the product of the probability  $1/n$  that the random scheduler selects leader  $A$  as the initiator, times the probability greater than  $1/2$  that the responder is a free agent. In order to finish the initialization, we need to create new dispensers  $2k - 1$  times. Using Chernoff bound, we observe that after  $O(kn) = O(n^2/\varepsilon^2)$  interactions all  $k$  cycles have their two dispensers formed. As each cycle dispenses  $n/k = 144/\varepsilon^2$  labels and the expected number of interactions required to dispense a single label is  $O(n^2)$  with high probability, the expected number of interactions required by a specific cycle to generate all labels is  $O(n^2/\varepsilon^2)$  also with high probability. As observed earlier, the leader election preprocessing adds

only  $O(n \log n)$  interactions w.h.p. Hence, the expected number of interactions required to conclude the labeling process is  $O(n^2/\varepsilon^2)$ . Finally, note that for small values of  $\varepsilon$  approaching  $n^{-1/2}$   $k$ -cycle protocol reduces to *Single-cycle* protocol and for constant  $\varepsilon$  the number of interactions required by the protocol is  $O(n^2)$ .

► **Theorem 12.** *For  $k = n\varepsilon^2/144$ , where  $\varepsilon = \Omega(n^{-1/2})$ , and the minimal label range  $[1, n]$ , the proposed  $k$ -cycle labeling protocol provides a space-time trade-off in which utilization of  $(1 + \varepsilon)n + O(\log \log n)$  states permits the expected number of interactions  $O(n^2/\varepsilon^2)$ .*

## 5 Lower bounds

In this chapter, we derive several lower bounds on the number of states or the number of interactions required by silent, safe or the so-called pool protocols for unique labeling. Importantly, these lower bounds also hold in our model assuming that the population size is known to the agents initially and also when a unique leader is available initially.

The following general lower bound valid for any range of labels follows immediately from the definitions of a population protocol and the problem of unique labeling, respectively.

► **Theorem 13.** *The problem of assigning unique labels to  $n$  agents requires  $\Omega(n \log n)$  interactions w.h.p. and the agents have to be equipped with at least  $n$  states.*

**Proof.**  $\Omega(n \log n)$  interactions are needed w.h.p. since each agent has to interact at least once, see, e.g., the introduction in [10]. The lower bound on the number of states follows from the symmetry of agents, so any agent has to be prepared to be assigned an arbitrary label with at least a logarithmic bit representation. ◀

**A sharper lower bound on the number of states.** We obtain the following lower bound on the number of states required by a silent protocol which produces a valid labeling of the  $n$  agents and is safe w.h.p. The lower bound holds even if the protocol is provided with a unique leader and the knowledge of the number of agents. It almost matches the upper bound established in the previous section.

► **Theorem 14.** *A silent protocol which produces a valid labeling of the  $n$  agents and is safe with probability larger than  $1 - \frac{1}{n}$  requires at least  $n + \sqrt{\frac{n-1}{2}} - 1$  states. Also, if a silent protocol, which produces a valid labeling of the  $n$  agents and is safe with probability 1, uses  $n + t$  states, where  $t < n$ , then the expected number of interactions required by the protocol to provide a valid labeling is  $\frac{n^2}{t+1}$ .*

**Proof.** Let  $I$  be the set of ordered pairs of the  $n$  agents.  $I$  can be interpreted as the set of possible pairwise interactions between the agents.

Let  $Z$  be a finite run of the protocol, i.e., a finite sequence of pairs in  $I$ . Suppose that the execution of  $Z$  is successful, i.e., each agent reaches a final state with a distinct label, and no agent gets assigned two or more distinct labels during the run.

Let  $F_Z$  be the set of final states achieved by the agents after the execution of the run  $Z$ . We have  $|F_Z| = n$ . Also, let  $R_Z$  stand for the set of remaining states used in this run. Observe that if an agent is in a state in  $F_Z$  then it has a label.

For an agent  $x$ , let  $f_Z(x) \in F_Z$  be the last state achieved by the agent in the run  $Z$ , and let  $pred_Z(x)$  be the next to the last state achieved by the agent  $x$  in the run. Since for at most one agent the common initial state can be the final one,  $pred_Z(\cdot)$  is defined for at least  $n - 1$  agents. If  $pred_Z(x) \in F_Z$  and  $pred_Z(x)$  assigns a distinct label from that

assigned by  $f_Z(x)$  to  $x$  then we have a contradiction with our assumptions on  $Z$ . In turn, if  $pred_Z(x) \in F_Z$  and  $pred_Z(x)$  assign the same label as that assigned by  $f_Z(x)$  to  $x$  then we have a contradiction with the validity of the final labeling resulting from  $Z$ . We conclude that if  $pred_Z(x)$  is defined then  $pred_Z(x) \in R_Z$ .

Next, let  $A_Z$  be the set of agents  $x$  that achieved their final state in the run  $Z$  by an interaction of  $x$  in the state  $pred_Z(x)$  with an agent in a state in  $F_Z$ . For the proof of the following claim under the assumptions of the first statement in the theorem, see Appendix F.

▷ **Claim 15.** There is a finite run  $Z$  of the protocol such that after the execution of  $Z$ , each agent is in a final state with a distinct label, no single agent is assigned distinct labels during  $Z$ , and for any pair of distinct agents  $x, y \in A_Z$ ,  $pred_Z(x) \neq pred_Z(y)$ .

From here on, we assume that the run  $Z$  satisfies the claim. Consequently,  $|R_Z| \geq |A_Z|$ .

Let  $B_Z$  be the set of remaining agents that got their final state in  $F_Z$  in an interaction where both agents were in states outside  $F_Z$ , i.e., in  $R_Z$ . Since the agents in  $B$  achieved distinct final states with distinct labels in the aforementioned interactions, we infer that  $2|R_Z|^2 \geq |B_Z|$  and thus  $|R_Z| \geq \sqrt{|B_Z|/2}$ . Simply, there are  $|R_Z|^2$  ordered pairs of states in  $R_Z$ , and when agents in the states forming such a pair interact they can achieve at most two distinct states in  $F_Z$ . (Consequently, if  $2|R_Z|^2 < |B_Z|$  then there would be a pair of agents in  $B_Z$  that would achieve the same final state in the run and hence it would have the same label at the end of the considered run.)

Thus, we obtain  $|R_Z| \geq \max\{|A_Z|, \sqrt{\frac{n-1-|A_Z|}{2}}\} \geq \sqrt{\frac{n-1}{2}} - 1$  by straightforward calculations. This completes the proof of the first statement of the theorem.

To prove the second statement of the theorem, we need  $|R_Z| \geq |A_Z|$  to hold for any run  $Z$  resulting in a valid labeling of the agents without updating the label of any single agent. The existence of such a run  $Z$  implied by Claim 15 is not sufficient to obtain a lower bound on the expected number of required interactions. The stronger assumptions on the silent protocol in the second statement of the theorem requiring the protocol to provide always a valid labeling without updating the label of any single agent solves the problem. Namely, if  $pred_Z(x) = pred_Z(y)$  for  $x, y \in A_Z$  then following the notation and argumentation from the proof of Claim 15 neither  $Z_1i_1Z_2i_3$  nor any of its lengthening can provide a valid labeling without updating the label of any single agent. We obtain a contradiction with the aforementioned assumptions. Thus, the inequality  $|R_Z| \geq |A_Z|$  holds for arbitrary run  $Z$  ending with a valid labeling without updating the label of any single agent.

To prove the second statement, we may also assume w.l.o.g. that  $|A_Z| < n$  since otherwise  $t \geq |R_Z| \geq |A_Z| \geq n$ . Hence, the set  $B_Z$  of agents is non-empty. Let  $x$  be a last agent in  $B_Z$  that being in the state  $pred(x)$  gets its final state  $f(x)$  by an interaction with another agent  $y$  in a state  $s$ . If  $y$  belongs to  $B_Z$  then both  $x$  and  $y$  are the two last agents in  $B_Z$  that simultaneously get their final states in  $F_Z$  in the same interaction. The probability of the interaction between them is only  $\frac{1}{n^2}$ . Suppose in turn that  $y$  belongs to  $A_Z$ . We know that  $t \geq |R_Z| \geq |A_Z|$  from the previous part. Thus, there are at most  $t$  agents in  $B_Z$  in the state  $s$  with which the agent  $x$  in the state  $pred_Z(x)$  could interact. The probability of such an interaction is at most  $\frac{t}{n^2}$ . We conclude that the probability of an interaction between the agent  $x$  and the agent  $y$  after which  $x$  gets its final state  $f(x)$  is at most  $\frac{t+1}{n^2}$ , which proves the second statement. ◀

▶ **Corollary 16.** *If for  $\varepsilon > 0$ , a silent protocol that produces a valid labeling of the  $n$  agents and is safe with probability 1 uses only  $n + O(n^{1-\varepsilon})$  states then the expected number of interactions required by the protocol to achieve a valid labeling is  $\Omega(n^{1+\varepsilon})$ .*

**A lower bound for the range  $[1, n + r]$ .** Our fast protocols presented in Section 3 are examples of a class of natural protocols for the unique labeling problem that we term *pool protocols*.

In each step of a pool protocol, a subset of agents owns explicit or implicit pools of labels which are pairwise disjoint and whose union is included in the assumed range of labels. When two agents interact, they can repartition the union of their pools among themselves. Before the start of a pool protocol, only a single agent (the leader) owns a pool of labels. This initial pool corresponds to the assumed range of labels. An agent can be assigned a label from its own pool only. After that, the label is removed from the pool and cannot be changed. Finally, an agent without an assigned label cannot give away the whole own pool during an interaction with another agent without getting some part of the pool belonging to the other agent.

► **Theorem 17.** *The expected number of interactions required by a pool protocol to assign unique labels in the range  $[1, n + r]$ , where  $r \geq 0$ , to the population of  $n$  agents is at least  $\frac{n^2}{r+1}$ .*

**Proof.** We shall say that an agent has the  $P$  property if the agent owns a non-empty pool or a label has been assigned to the agent. Observe that if an agent accomplishes the  $P$  property during running a pool protocol then it never loses it. Also, all agents have to accomplish the  $P$  property sooner or later in order to complete the assignment task. During each interaction of a pool protocol at most one more agent can get the  $P$  property. Since at the beginning only one agent has the  $P$  property, there must exist an interaction after which only one agent lacks this property. By the disjointedness of the pools and labels, the assumed label range, and the definition of a pool protocol, there are at most  $r + 1$  agents among the remaining ones that could donate a sub-pool or label from own pool to the agent missing the  $P$  property. The expected number of interactions leading to an interaction between the agent missing the  $P$  property and one of the at most  $r + 1$  agents is  $\frac{n^2}{r+1}$ . ◀

## 6 Final remarks

Our upper bound of  $n + 5 \cdot \sqrt{n} + O(n^c)$ , for any positive  $c < 1$ , on the number of states achieved by a protocol for unique labeling that is silent and safe once a unique leader is elected almost matches our lower bound of  $n + \sqrt{\frac{n-1}{2}} - 1$ . We can combine our protocols for unique labeling with the recent protocols for counting or approximating the population size due to Berenbrink et al. [10] in order to get rid of the assumption that the population size is known to one of the agents initially. Since the aforementioned protocols from [10] either require  $\tilde{O}(n)$  states or  $O(n \log^2 n)$  interactions, the resulting combinations lose some of the near-optimality or optimality properties of our protocols (cf. Corollaries 5, 10). The related question if one can design a protocol for counting or closely approximating the population size simultaneously requiring  $O(n \log n)$  interactions w.h.p. and at most  $cn$  states, where  $c$  is a low constant, is of interest in its own right.

---

### References

- 1 D. Alistarh, H. Attiya, S. Gilbert, A. Giurgiu, and R. Guerraoui. Fast randomized test-and-set and renaming. In *Proc. of the International Symposium on Distributed Computing, DISC*, volume 6343 of *Lect. Notes in Comput. Sci.*, pages 94–108. Springer, 2010.
- 2 D. Alistarh, O. Denysyuk, L. Rodrigues, and N. Shavit. Balls-into-leaves: Sub-logarithmic renaming in synchronous message-passing systems. In *Proc. of the 2014 ACM Symposium on Principles of Distributed Computing, PODC*, pages 232–241. ACM, 2014.



- 3 D. Alistarh and R. Gelashvili. Recent algorithmic advances in population protocols. *ACM SIGACT News*, 49(3):63–73, 2018.
- 4 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.
- 5 D. Angluin, J. Aspnes, and D. Eisenstat. Fast computation by population protocols with a leader. In *Proc. of the International Symposium on Distributed Computing , DISC*, volume 4167 of *Lect. Notes in Comput. Sci.*, pages 61–75. Springer, 2006.
- 6 J. Aspnes, J. Beauquier, J. Burman, and D. Sohler. Time and space optimal counting in population protocols. In *Proc. of the 20th International Conference on Principles of Distributed Systems, OPODIS*, volume 70 of *LIPICs*, pages 13:1–13:17. Dagstuhl - LZI, 2016.
- 7 J. Beauquier, J. Burman, L. Rosaz, and B. Rozoy. Non-deterministic population protocols. In *Proc. of the 20th International Conference on Principles of Distributed Systems, OPODIS*, *Lect. Notes in Comput. Sci.*, pages 61–75. Springer, 2012.
- 8 P. Berenbrink, A. Brinkmann, R. Elsässer, T. Friedetzky, and L. Nagel. Randomized renaming in shared memory systems. In *Proc. of the IEEE Int. Parallel and Distributed Processing Symp., IPDPS*, pages 542–549. IEEE Computer Society, 2015.
- 9 P. Berenbrink, G. Giakkoupis, and P. Kling. Optimal time and space leader election in population protocols. In *Proc. of the 52nd ACM-SIGACT Symposium on Theory of Computing, STOC*, pages 119–129. ACM, 2020.
- 10 P. Berenbrink, D. Kaaser, and T. Radzik. On counting the population size. In *Proc. of the ACM Symposium on Principles of Distributed Computing, PODC*, pages 43–52. ACM, 2019.
- 11 J. Burman, J. Beauquier, and D. Sohler. Space-optimal naming in population protocols. In *Proc. of the 33rd International Symposium on Distributed Computing, DISC*, volume 146 of *LIPICs*, pages 9:1–9:16. Dagstuhl - LZI, 2019.
- 12 J. Burman, H. Chen, H. Chen, D. Doty, T. Nowak, E. Severson, and C. Xu. Time-optimal self-stabilizing leader election in population protocols. In *Proc. of the ACM Symposium on Principles of Distributed Computing, PODC*, pages 33–44. ACM, 2021.
- 13 S. Cai, T. Izumi, and K. Wada. How to prove impossibility under global fairness: On space complexity of self-stabilizing leader election on a population protocol model. *Theory of Computing Systems*, 50:433–445, 2012.
- 14 A. Castañeda, S. Rajsbaum, and M. Raynal. The renaming problem in shared memory systems: An introduction. *Comput. Sci. Rev.*, 5(3):229–251, 2011.
- 15 S. Dolev, M.G. Gouda, and M. Schneider. Memory requirements for silent stabilization. *Acta Inf.*, 36(6):447–462, 1999.
- 16 D. Doty and M. Eftekhari. A survey of size counting in population protocols. *Theoretical Computer Science*, 894:91–102, 2021.
- 17 D. Doty, M. Eftekhari, O. Michail, P. G. Spirakis, and M. Theofilatos. Exact size counting in uniform population protocols in nearly logarithmic time. *ArXiv*, 2018. preprint. [arXiv: 1805.04832](https://arxiv.org/abs/1805.04832).
- 18 R. Elsässer and T. Radzik. Recent results in population protocols for exact majority and leader election. *Bull. EATCS*, 126, 2018.
- 19 L. Gąsieniec and G. Stachowiak. Enhanced phase clocks, population protocols, and fast space optimal leader election. *J. ACM*, 68(1):2:1–2:21, 2021.

## **A** The computational model of population protocols

There is given a population of  $n$  agents that can pairwise interact in order to change their states and in this way perform a computation. A population protocol can be formally specified by providing a set  $Q$  of possible states, a set  $O$  of possible outputs, a transition function  $\delta : Q \times Q \rightarrow Q \times Q$ , and an output function  $o : Q \rightarrow O$ . The current state  $q \in Q$  of an agent is updated during interactions. Consequently, the current output  $o(q)$  of the agent also becomes updated during interactions. The current state of the set of  $n$  agents is given by a vector in  $Q^n$  with the current states of the agents. A computation of a population protocol

## 12:18 Efficient Assignment of Identities in Anonymous Populations

is specified by a sequence of pairwise interactions between agents. In every time step, an ordered pair of agents is selected for interaction by a probabilistic scheduler independently and uniformly at random. The first agent in the selected pair is called the initiator while the second one is called the responder. The states of the two agents are updated during the interaction according to the transition function  $\delta$ .

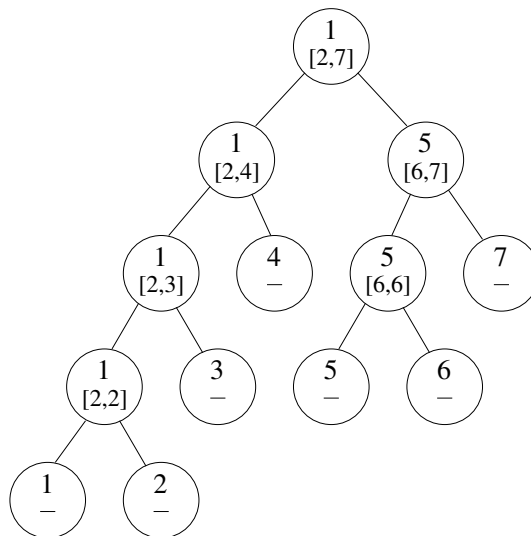
We can specify a problem to solve by a population protocol by providing the set of input configurations, the set  $O$  of possible outputs, and the desired output configurations for given input configurations. For the unique labeling problem, all agents are initially in the same state  $q_0$ . The set  $O$  is just the set of positive integers. A desired configuration is when all agents output their distinct labels. The *stabilization time* of an execution of a protocol is the number of interactions until the states of agents form a desired configuration from which no sequence of pairwise interactions can lead to a configuration outside the set of desired configurations.

### B Related work (Table 3)

■ **Table 3** Upper bounds on the number of interactions, the number of states and the range used by the known labeling protocols. In case of the self-stabilizing labeling protocols in [12], the “safe” property can eventually hold only for their initialized versions.

$n$	# interactions	# states	Range	Properties	Paper
unknown	$O(n^3)$ w.h.p.	$n$	$[1, n]$	silent	[13]
unknown	$O(n \log n \log \log n)$ w.h.p.	$n^{O(1)}$	$[1, n^{O(1)}]$	silent	[17]
known	$O(n^2)$ expected	$O(n)$	$[1, n]$	silent, safe	[12]
known	$O(n \log n)$ w.h.p.	$\exp(O(n^{\log n} \log n))$	$[1, n]$	safe	[12]

### C Figure 1



■ **Figure 1** An example of the partition tree of the start interval  $[1, 7]$ .

## D Labeling with asymptotically optimal number of interactions within $[1, (1 + \epsilon)n]$ (proofs)

**Proof of Lemma 6.** The proof is a generalization of that for Lemma 1. Define  $F_\epsilon$  as a set of at least  $\epsilon n/4$  agents without labels that survive at least  $cn \log / \epsilon$  interactions in the first phase. Note that for an arbitrary agent, the probability of interaction with a member in  $F_\epsilon$  is at least  $\frac{\epsilon}{4n}$ . The rest of the proof is analogous to that of Lemma 1. It is sufficient to replace  $F$  by  $F_\epsilon$  and the probability  $\frac{1}{4n}$  of an interaction with a member in  $F$  with that  $\frac{\epsilon}{4n}$  of an interaction with a member in  $F_\epsilon$ . ◀

**Proof of Lemma 7.** The number of agents without labels at the beginning of the second phase is smaller than  $\epsilon n/4$  w.h.p. Hence, at the beginning of the second phase the number of agents with labels in the range  $[1, \epsilon n]$  is at least  $\frac{3\epsilon n}{4}$  w.h.p. Recall that such an agent at a leaf of the tree can give a label to an agent without label only once. It follows that the number of agents with labels in  $[1, \epsilon n]$  that can give a label to an agent without label is always at least  $\frac{\epsilon n}{2}$  w.h.p. We conclude that for an agent without label the probability of an interaction with an agent that can give a label is at least almost  $\frac{\epsilon}{2n}$ . Hence, after each  $O(n/\epsilon)$  interactions the expected number of agents without labels halves. It follows that the expected number of such interactions rounds is  $O(\log n)$ . Consequently, the number of the rounds is also  $O(\log n)$  w.h.p. by Fact 2.

An alternative way to obtain the  $O(n \log n/\epsilon)$  bound on the number of interactions w.h.p. is to use Fact 3 analogously as in the proof of Lemma 2. The difference is that  $C$  is set to  $O(\frac{2}{\epsilon})$  instead of  $O(2)$  since the set of agents that can give a label is of size at least  $\frac{n\epsilon}{2}$  now. ◀

**Proof of Lemma 8.** During the initial part of the broadcasting process, after every  $O(n/\epsilon)$  interactions, the expected number of agents participating in the broadcasting process doubles. Hence, after  $O(n \log / \epsilon)$  interactions, the expected number of informed agents will be  $\Omega(n\epsilon)$ . Then, the expected number of uninformed agents will be halved for every  $O(n/\epsilon)$  interactions. So the expected number of rounds, each consisting of  $O(n/\epsilon)$  interactions, needed to complete the broadcasting is  $O(\log n)$ . It remains to turn the latter bound to a w.h.p. one. This can be done by using the Chernoff bounds (Fact 2).

Alternatively, we can define for the purpose of the analysis of the doubling part, a binary broadcast tree. An informed agent at an intermediate node of the tree after an interaction with an uninformed agent moves to a child of the node while the other agent now informed places at the other child (cf. the partition tree in the proofs of Lemmata 1, 6). Then, we can use the technique from the proofs of Lemmata 1, 6 to show that only  $O(n \log n/\epsilon)$  interactions are required w.h.p. to achieve a configuration where only a constant fraction of the agents participating in the broadcasting is uninformed. To derive the same asymptotic upper bound on the number of interactions required by the halving part w.h.p., we can use Fact 3 with  $C = O(\epsilon^{-1})$  analogously as in the proofs of Lemmata 2, 7. ◀

## E The transition function of the state optimal protocol

**State utilization in *Single-Cycle* protocol.**

- [Agent  $A$ ] Since  $\text{label}(a) \in \{0, \dots, \sqrt{n}-1\}$  dispenser  $A$  utilizes  $2 \cdot \sqrt{n} + 2$  states including:
- $A.\text{init} = (1)$  # the initial (leadership) state of dispenser  $A$ ,
  - $A[\text{label}(a), \text{await}(F)]$  # dispenser  $A$  carrying partial label  $\text{label}(a)$  awaits interaction with a free agent  $F$ ,

## 12:20 Efficient Assignment of Identities in Anonymous Populations

- $A[\text{label}(a), \text{await}(B)]$  # dispenser  $A$  carrying partial label  $\text{label}(a)$  awaits interaction with dispenser  $B$ ,
- $A.\text{final} = (0, 1)$  # the final state of  $A$ .

[**Agent  $B$** ] Since  $\text{label}(b) \in \{0, \dots, \sqrt{n}\}$  dispenser  $B$  utilizes  $2 \cdot \sqrt{n} + 3$  states including:

- $B[\text{label}(b), \text{await}(F)]$  # dispenser  $B$  carrying partial label  $\text{label}(b)$  awaits interaction with a free agent  $F$ ,
- $B[\text{label}(b), \text{await}(A)]$  # dispenser  $B$  carrying partial label  $\text{label}(b)$  awaits interaction with dispenser  $A$
- $B.\text{final} = (0, 2)$  # the final state of  $B$ .

[**Agent  $F$** ] Since free agents carry partial labels  $\text{label}(a) \in \{0, \dots, \sqrt{n} - 1\}$  and eventually adopt one of the  $n - 2$  destination labels (excluding dispensers) they utilize  $n + \sqrt{n} - 1$  states including:

- $F.\text{init} = (0)$  # the initial (non-leader) state of  $F$
- $F[\text{label}(a), \text{await}(B)]$  # free agent  $F$  carrying partial label  $\text{label}(a)$  awaits interaction with dispenser  $B$ ,
- $F.\text{final} = (\text{label}(a), \text{label}(b))$  # the final state of  $F$ .

In total *Single-Cycle* protocol requires  $n + 5 \cdot \sqrt{n} + 4$  states.

### Transition function in *Single-Cycle* protocol.

**Step 0: Initialization.** During the first interaction of  $A$  with a free agent the second dispenser  $B$  is nominated. Both dispensers adopt their largest labels. Agent  $A$  awaits a free agent in the initial state while agent  $B$  awaits a free agent carrying a partial label obtained from  $A$ .

- $(A.\text{init}, F.\text{init})$   
 $\rightarrow (A[\text{label}(a) = \sqrt{n} - 1, \text{await}(F)], B[\text{label}(b) = \sqrt{n}, \text{await}(F)]),$

The three steps  $C_1, C_2$ , and  $C_3$  of the labeling cycle are given below.

**Step  $C_1$ : Agent  $A$  dispenses partial label.** During an interaction of agent  $A$  with a free agent  $F$  the current partial label  $\text{label}(a)$  is dispensed to  $F$ . Both agents await interactions with dispenser  $B$  which is ready to interact with partially labeled  $F$  but not  $A$ .

- $(A[\text{label}(a), \text{await}(F)], F.\text{init})$   
 $\rightarrow (A[\text{label}(a), \text{await}(B)], F[\text{label}(a), \text{await}(B)])$  # Go to Step  $C_2$

**Step  $C_2$ : Agent  $B$  dispenses partial label.** During an interaction of agent  $B$  with a free agent  $F$  which carries partial label  $\text{label}(a)$ , the complementary current partial label  $\text{label}(b)$  is dispensed to  $F$ . Agent  $F$  concludes in the final state with the combined label  $(\text{label}(a), \text{label}(b))$ . Agent  $B$  is now ready for interaction with  $A$ .

- $(B[\text{label}(b), \text{await}(F)], F[\text{label}(a), \text{await}(B)])$   
 $\rightarrow (B[\text{label}(b), \text{await}(A)], F.\text{final} = (\text{label}(a), \text{label}(b)))$  # Go to Step  $C_3$

**Step  $C_3$ : Agent  $A$  and  $B$  negotiate a new label or conclude.** In the case when  $\text{label}(a) = 0$  and  $\text{label}(b) = 2$  the dispensers  $A$  and  $B$  conclude in states  $(0, 1)$  and  $(0, 2)$  respectively, see the first transition. Otherwise a new combination of partial labels is agreed and the protocol goes back to Step  $C_1$ .

- $(A[\text{label}(a) = 0, \text{await}(B)], B[\text{label}(b) = 2, \text{await}(A)])$   
 $\rightarrow (A.\text{final} = (0, 1), B.\text{final} = (0, 2))$  # Conclude the labeling process

- $(A[\text{label}(a) = 0, \text{await}(B)], B[\text{label}(b) > 2, \text{await}(A)])$  or  
 $(A[\text{label}(a) > 0, \text{await}(B)], B[\text{label}(b) > 1, \text{await}(A)])$   
 $\rightarrow (A[\text{label}(a), \text{await}(F)], B[\text{label}(b) - 1, \text{await}(F)])$  # Go to Step  $C_1$
- $(A[\text{label}(a) > 0, \text{await}(B)], B[\text{label}(b) = 1, \text{await}(A)])$   
 $\rightarrow (A[\text{label}(a) - 1, \text{await}(F)], B[\text{label}(b) = \sqrt{n}, \text{await}(F)])$  # Go to Step  $C_1$

## F Lower bounds (proofs)

**Proof of Claim 15.** The proof of the claim is by a contradiction with the assumptions on the labeling protocol. The general intuition is that if  $\text{pred}_Z(x) = \text{pred}_Z(y)$  for two agents  $x, y \in A_Z$  then we can associate with a prefix of  $Z$  a slightly modified equally likely run  $Z'$  which assigns the same label to a pair of agents. Hence, the modified run does not produce a valid labeling or it has to assign at least two different labels to some agent.

To obtain the contradiction, we assume that for each finite run  $Z$  in which the agents achieve final states with distinct labels without assigning distinct labels to any single agent during the run, there is a pair of agents  $x, y \in A_Z$ , where  $\text{pred}_Z(x) = \text{pred}_Z(y)$ . Let us consider such a pair of agents  $x, y \in A_Z$  that minimizes the length of the prefix of  $Z$  in which both agents achieve their final states in  $F_Z$ . We may assume w.l.o.g. that  $x$  gets its final state  $f_Z(x)$  in an interaction  $i_1$  with an agent  $x'$  that is in a state in  $F_Z$ , and in a later interaction  $i_2$ ,  $y$  gets its final state  $f_Z(y)$ , in the run  $Z$ . (Note that  $x'$  cannot be in a final state different from its own, i.e., in  $F_Z \setminus \{f_Z(x')\}$  since this would require updating its label contradicting the assumption on  $Z$ .) Thus, the shortest prefix of  $Z$  in which both  $x$  and  $y$  get their final states has the form  $Z_1 i_1 Z_2 i_2$ . Then, if we replace the latter interaction  $i_2$  by the interaction  $i_3$  between  $y$  and the agent  $x'$  in the state  $f_Z(x')$  analogous to  $i_1$ , it will result in achieving by  $y$  the state  $f_Z(x)$  since  $\text{pred}_Z(x) = \text{pred}_Z(y)$ . Thus, neither the run  $Z_1 i_1 Z_2 i_3$  nor any of its extensions can yield a valid labeling of the agents without updating labels for some of them. Importantly, the runs  $Z_1 i_1 Z_2 i_2$  and  $Z_1 i_1 Z_2 i_3$  are equally likely (\*).

We initialize two sets  $S_{\text{valid}}$  and  $S_{\text{invalid}}$  of strings (sequences) over the alphabet  $I$ . Then, for each run  $Z$  in which the agents achieve final states with distinct labels without updating the label of any single agent, we insert the prefix  $Z_1 i_1 Z_2 i_2$  into  $S_{\text{valid}}$  and the corresponding sequence  $Z_1 i_1 Z_2 i_3$  into  $S_{\text{invalid}}$ . Note that by the choice of  $i_1, i_2$ , no string in  $S_{\text{valid}}$  is a prefix of another string in  $S_{\text{valid}}$ . The analogous property holds for  $S_{\text{invalid}}$ . By the construction of the sets, each run  $Z$  in which the agents achieve final states with distinct labels without updating the label of any single agent has to overlap with or be a lengthening of a string in  $S_{\text{valid}}$ . Furthermore, no run of the protocol that overlaps with a string in  $S_{\text{invalid}}$  or it is a lengthening of a string in  $S_{\text{invalid}}$  results in a valid labeling without updating the label of any single agent. Define the function  $g : S_{\text{valid}} \rightarrow S_{\text{invalid}}$  by  $g(Z_1 i_1 Z_2 i_2) = Z_1 i_1 Z_2 i_3$ . By the property (\*), the probability that a string over  $I$  is equal to  $Z_1 i_1 Z_2 i_2$  or it is a lengthening of  $Z_1 i_1 Z_2 i_2$  is not greater than the probability that a string over  $I$  is equal to  $g(Z_1 i_1 Z_2 i_2)$  or it is a lengthening of  $g(Z_1 i_1 Z_2 i_2)$ . The function  $g$  is not necessarily a bijection. Suppose that  $g(Z_1 i_1 Z_2 i_2) = g(Z'_1 i'_1 Z'_2 i'_2)$ . Then, we have  $Z_1 i_1 Z_2 i_3 = Z'_1 i'_1 Z'_2 i'_3$ . Consequently, the strings  $Z_1 i_1 Z_2 i_2$  and  $Z'_1 i'_1 Z'_2 i'_2$  may only differ in the last interaction, i.e.,  $i_2$  may be different from  $i'_2$ . However,  $i_2$  and  $i'_2$  have to include the same agent ( $y$  in the earlier construction) that appears in  $i_3$ . We conclude that the aforementioned two strings in  $S_{\text{valid}}$  can differ by at most one agent in the last interaction. It follows that  $g$  maps at most  $n - 1$  strings in  $S_{\text{valid}}$  to the same string in  $S_{\text{invalid}}$ . Hence, the event that the agents eventually achieve their final states yielding a valid labeling without updating the label of any single agent is at most  $n - 1$  times more likely than the complement event, contradicting our assumptions. ◀



# Population Protocols for Graph Class Identification Problems

Hiroto Yasumi ✉

Nara Institute of Science and Technology, Japan

Fukuhito Ooshita ✉

Nara Institute of Science and Technology, Japan

Michiko Inoue ✉

Nara Institute of Science and Technology, Japan

---

## Abstract

In this paper, we focus on graph class identification problems in the population protocol model. A graph class identification problem aims to decide whether a given communication graph is in the desired class (*e.g.* whether the given communication graph is a ring graph). Angluin et al. proposed graph class identification protocols with directed graphs and designated initial states under global fairness [Angluin et al., DCOSS2005]. We consider graph class identification problems for undirected graphs on various assumptions such as initial states of agents, fairness of the execution, and initial knowledge of agents. In particular, we focus on lines, rings,  $k$ -regular graphs, stars, trees, and bipartite graphs. With designated initial states, we propose graph class identification protocols for  $k$ -regular graphs and trees under global fairness, and propose a graph class identification protocol for stars under weak fairness. Moreover, we show that, even if agents know the number of agents  $n$ , there is no graph class identification protocol for lines, rings,  $k$ -regular graphs, trees, or bipartite graphs under weak fairness, and no graph class identification for lines, rings,  $k$ -regular graphs, stars, trees, or bipartite graphs with arbitrary initial states.

**2012 ACM Subject Classification** Theory of computation → Distributed algorithms; Theory of computation → Concurrent algorithms

**Keywords and phrases** population protocol, graph class identification, distributed protocol

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2021.13

**Related Version** *Full Version:* <https://arxiv.org/abs/2111.05111> [26]

**Funding** This work was supported by JSPS KAKENHI Grant Numbers 18K11167, 20H04140, 20J21849, and JST SICORP Grant Number JPMJSC1806.

## 1 Introduction

The population protocol model is an abstract model for low-performance devices, introduced by Angluin et al. [4]. In this model, a network, called population, consists of multiple devices called agents. Those agents are anonymous (*i.e.*, they do not have identifiers), and move unpredictably (*i.e.*, they cannot control their movements). When two agents approach, they are able to communicate and update their states (this communication is called an interaction). By a sequence of interactions, the system proceeds a computation. In this model, there are various applications such as sensor networks used to monitor wild birds and molecular robot networks [24].

In this paper, we study the computability of graph properties of communication graphs in the population protocol model. Concretely, we focus on graph class identification problems that aim to decide whether the communication graph is in the desired graph class. In most distributed systems, it is essential to understand properties of the communication graph in order to design efficient algorithms. Actually, in the population protocol model,



© Hiroto Yasumi, Fukuhito Ooshita, and Michiko Inoue;  
licensed under Creative Commons License CC-BY 4.0

25th International Conference on Principles of Distributed Systems (OPODIS 2021).

Editors: Quentin Bramas, Vincent Gramoli, and Alessia Milani; Article No. 13; pp. 13:1–13:19



Leibniz International Proceedings in Informatics

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

efficient protocols are proposed with limited communication graphs (*e.g.*, ring graphs and regular graphs) [2, 6, 16, 17]. In the population protocol model, the computability of the graph property was first considered in [3]. In [3], Angluin et al. proposed various graph class identification protocols with directed graphs and designated initial states under global fairness. Concretely, Angluin et al. proposed graph class identification protocols for directed lines, directed rings, directed stars, and directed trees. Moreover, they proposed graph class identification protocols for other graphs such as 1) graphs having degree bounded by a constant  $k$ , 2) graphs containing a fixed subgraph, 3) graphs containing a directed cycle, and 4) graphs containing a directed cycle of odd length. However, there are still some open questions such as “What is the computability for undirected graphs?” and “How do other assumptions (*e.g.*, initial states, fairness, etc.) affect the computability?” In this paper, we answer those questions. That is, we clarify the computability of graph class identification problems for undirected graphs under various assumptions such as initial states of agents, fairness of the execution, and an initial knowledge of agents. More concretely, in this paper, we consider the problems with designated or arbitrary initial states, under global or weak fairness, and with the number of agents  $n$ , the upper bound  $P$  of the number of agents, or no knowledge. The assumption of initial states bears on the requirement of initialization and the fault-tolerant property. To execute a protocol with designated initial states, it is necessary to initialize all agents. Alternatively, a protocol with arbitrary initial states does not need to initialize agents. This implies that, even if agents transition to incorrect states by transient faults, the protocol can recover to desired configurations. Fairness is an assumption of interaction patterns. Intuitively, global fairness guarantees that, if a reachable configuration can occur infinitely often, the reachable configuration actually occurs infinitely often. On the other hand, weak fairness only guarantees that interactions occur infinitely often between each pair of adjacent agents. The initial knowledge is given to agents for helping the agents solve the problem. The initial knowledge enables us to construct efficient protocols although it may be difficult to know the knowledge in some situations.

In the population protocol, researchers also considered other assumptions such as symmetry and randomness (deterministic or non-deterministic). In this paper, we consider only deterministic asymmetric protocols. Note that, with designated initial states under global fairness, there is a transformer that transforms an asymmetric protocol into a symmetric protocol by assuming additional states [13]. Although we deal only with asymmetric protocols, we can transform most of our asymmetric protocols to symmetric protocols by applying this transformer.

We remark that some protocols in [3] for directed graphs can be easily extended to undirected graphs with designated initial states under global fairness (see Table 1). Concretely, graph class identification protocols for directed lines, directed rings, and directed stars can be easily extended to protocols for undirected lines, undirected rings, and undirected stars, respectively. In addition, the graph class identification protocol for bipartite graphs can be deduced from the protocol that decides whether a given graph contains a directed cycle of odd length. This is because, if we replace each edge of an undirected non-bipartite graph with two opposite directed edges, the directed non-bipartite graph always contains a directed cycle of odd length. On the other hand, the graph class identification protocol for directed trees cannot work for undirected trees because the protocol uses a property of directed trees such that in-degree (resp., out-degree) of each agent is at most one on an out-directed tree (resp., an in-directed tree). Note that agents can identify trees if they understand the graph contains no cycle. However, the graph class identification protocol for graphs containing a directed cycle in directed graphs cannot be used to identify a (simple) cycle in undirected graphs. This is because, if we replace an undirected edge with two opposite directed edges, the two directed edges compose a directed cycle.



■ **Table 1** The number of states to solve the graph class identification problems.  $n$  is the number of agents and  $P$  is an upper bound of the number of agents.

Model			Graph Properties						
Initial states	Fairness	Initial knowledge	<i>Line</i>	<i>Ring</i>	<i>Bipartite</i>	<i>Tree</i>	<i>k-regular</i>	<i>Star</i>	
Designated	Global	$n$	$O(1)^\dagger$	$O(1)^\dagger$	$O(1)^\dagger$	$O(1)^*$	$O(k \log n)^*$	$O(1)^\dagger$	
		$P$	$O(1)^\dagger$	$O(1)^\dagger$	$O(1)^\dagger$	$O(1)^*$	$O(k \log P)^*$	$O(1)^\dagger$	
		None	$O(1)^\dagger$	$O(1)^\dagger$	$O(1)^\dagger$	$O(1)^*$	–	$O(1)^\dagger$	
	Weak	$n$	Unsolvable*						$O(n)^*$
		$P/\text{None}$	Unsolvable*						
Arbitrary	Global/Weak	$n/P/\text{None}$	Unsolvable*						

\* Contributions of this paper †Deduced from Angluin et al. [3]

## Our Contributions

In this paper, we clarify the computability of graph class identification problems for undirected graphs under various assumptions. Recall that we consider only deterministic asymmetric protocols. A summary of our results is given in Table 1. Under global fairness, we propose two graph class identification protocols. One is a graph class identification protocol for trees with designated initial states. This protocol works with constant number of states even if no initial knowledge is given. The other is a graph class identification protocol for  $k$ -regular graphs with designated initial states and the initial knowledge of the upper bound  $P$  of the number of agents. On the other hand, under weak fairness, we show that there exists no graph class identification protocol for lines, rings,  $k$ -regular graphs, stars, trees, or bipartite graphs even if the upper bound  $P$  of the number of agents is given. In the case where the number of agents  $n$  is given, we propose a graph class identification protocol for stars and prove that there exists no graph class identification protocol for lines, rings,  $k$ -regular graphs, trees, or bipartite graphs. With arbitrary initial states, we prove that there is no protocol for lines, rings,  $k$ -regular graphs, stars, trees, or bipartite graphs. In this paper, because of space constraints, we omit the details of protocols and some proofs (see the full version in [26]).

## Related Works

The population protocol model was proposed by Angluin et al. [4]. While they mainly studied the computability of the model in the paper, subsequent works studied various problems (*e.g.*, leader election [1, 11, 18, 21], counting [7, 8, 12, 22], majority [5, 10, 20], etc) under different assumptions (*e.g.*, fairness assumption [7, 8], initial states of agents [6, 9], and initial knowledge of agents [14, 25]).

Although those problems are usually considered with complete communication graphs (*i.e.*, every pairwise interaction can occur), some researchers proposed efficient protocols with limited communication graphs (*e.g.*, ring graph, regular graph, etc.) [2, 6, 16, 17]. More concretely, Angluin et al. proposed a protocol that constructs a spanning tree with regular graphs [6]. Chen et al. proposed self-stabilizing leader election protocols with ring graphs [16] and regular graphs [17]. Alistarh et al. showed that protocols for complete graphs (including the leader election protocol, the majority protocol, etc.) can be simulated efficiently in regular graphs [2].

For graph class identification problems, after Angluin et al. studied some solvabilities [3], Chatzigiannakis et al. studied solvabilities for directed graphs with some properties on the mediated population protocol model [15], where the mediated population protocol model is an extension of the population protocol model. In this model, a communication link (on which agents interact) has a state. Agents can read and update the state of the communication

link when agents interact on the communication link. In [15], they proposed graph class identification protocol for some graphs such as 1) graphs having degree bounded by a constant  $k$ , 2) graphs in which the degree of each agent is at least  $k$ , 3) graphs containing an agent such that in-degree of the agent is greater than out-degree of the agent, 4) graphs containing a directed path of at least  $k$  edges, etc. Since Chatzigiannakis et al. proposed protocols for the mediated population protocol model, the protocols cannot work in the population protocol model. As impossibility results, they showed that there is no graph class identification protocol that decides whether the given directed graph has two edges  $(u, v)$  and  $(v, u)$  for two agents  $u$  and  $v$ , or whether the given directed graph is weakly connected.

As another perspective of communication graphs, Michail and Spirakis proposed a network constructors model that is an extension of the mediated population protocol [23]. The network constructors model aims to construct a desired graph on the complete communication graph by using communication links with two states. Each communication link only has *active* or *inactive* state. Initially, all communication links have inactive state. By activating/deactivating communication links, the protocol of this model constructs a desired communication graph that consists of agents and activated communication links. In [23], they proposed protocols that construct spanning lines, spanning rings, spanning stars, and regular graphs. Moreover, by relaxing the number of states, they proposed a protocol that constructs a large class of graphs.

## 2 Definitions

### 2.1 Population Protocol Model

A communication graph of a population is represented by a simple undirected connected graph  $G = (V, E)$ , where  $V$  represents a set of agents, and  $E \subseteq V \times V$  is a set of edges (containing neither multi-edges nor self-loops) that represent the possibility of an interaction between two agents (i.e., only if  $(a, b) \in E$  holds, two agents  $a \in V$  and  $b \in V$  can interact).

A protocol  $\mathcal{P} = (Q, Y, \gamma, \delta)$  consists of a finite set  $Q$  of possible states of agents, a finite set of output symbols  $Y$ , an output function  $\gamma : Q \rightarrow Y$ , and a set of transitions  $\delta$  from  $Q \times Q$  to  $Q \times Q$ . Output symbols in  $Y$  represent outputs as the results according to the purpose of the protocol. Output function  $\gamma$  maps a state of an agent to an output symbol in  $Y$ . Each transition in  $\delta$  is denoted by  $(p, q) \rightarrow (p', q')$ . This means that, when an agent  $a$  in state  $p$  interacts with an agent  $b$  in state  $q$ , their states become  $p'$  and  $q'$ , respectively. We say that such  $a$  is an initiator and such  $b$  is a responder. When  $a$  and  $b$  interact as an initiator and a responder, respectively, we simply say that  $a$  interacts with  $b$ . Transition  $(p, q) \rightarrow (p', q')$  is null if both  $p = p'$  and  $q = q'$  hold. We omit null transitions in the descriptions of protocols. Protocol  $\mathcal{P} = (Q, Y, \gamma, \delta)$  is deterministic if, for any pair of states  $(p, q) \in Q \times Q$ , exactly one transition  $(p, q) \rightarrow (p', q')$  exists in  $\delta$ . Recall that we consider only deterministic protocols in this paper.

A configuration represents a global state of a population, defined as a vector of states of all agents. A state of agent  $a$  in configuration  $C$  is denoted by  $s(a, C)$ . Moreover, when  $C$  is clear from the context, we simply use  $s(a)$  to denote the state of agent  $a$ . A transition from configuration  $C$  to configuration  $C'$  is denoted by  $C \rightarrow C'$ , and means that, by a single interaction between two agents, configuration  $C'$  is obtained from configuration  $C$ . For two configurations  $C$  and  $C'$ , if there exists a sequence of configurations  $C = C_0, C_1, \dots, C_m = C'$  such that  $C_i \rightarrow C_{i+1}$  holds for every  $i$  ( $0 \leq i < m$ ), we say  $C'$  is reachable from  $C$ , denoted by  $C \xrightarrow{*} C'$ .

An execution of a protocol is an infinite sequence of configurations  $\Xi = C_0, C_1, C_2, \dots$  where  $C_i \rightarrow C_{i+1}$  holds for every  $i$  ( $i \geq 0$ ). An execution  $\Xi$  is weakly-fair if, for any adjacent agents  $a$  and  $a'$ ,  $a$  interacts with  $a'$  and  $a'$  interacts with  $a$  infinitely often<sup>1</sup>. An execution  $\Xi$  is globally-fair if, for each pair of configurations  $C$  and  $C'$  such that  $C \rightarrow C'$ ,  $C'$  occurs infinitely often when  $C$  occurs infinitely often. Intuitively, global fairness guarantees that, if configuration  $C$  occurs infinitely often, then any possible interaction in  $C$  also occurs infinitely often. Then, if  $C$  occurs infinitely often,  $C'$  satisfying  $C \rightarrow C'$  occurs infinitely often, and we can deduce that  $C''$  satisfying  $C' \rightarrow C''$  also occurs infinitely often. Overall, with global fairness, if a configuration  $C$  occurs infinitely often, then every configuration  $C^*$  reachable from  $C$  also occurs infinitely often.

In this paper, we consider three possibilities for an initial knowledge of agents: the number of agents  $n$ , the upper bound  $P$  of the number of agents, and no knowledge. Note that the protocol depends on this initial knowledge. When we explicitly state that an integer  $x$  is given as the number of agents, we write the protocol as  $\mathcal{P}_{n=x}$ . Similarly, when we explicitly state that an integer  $x$  is given as the upper bound of the number of agents, the protocol is denoted by  $\mathcal{P}_{P=x}$ .

## 2.2 Graph Properties and Graph Class Identification Problems

We define graph properties treated in this paper as follows:

- A graph  $G$  satisfies property *tree* if there is no cycle on graph  $G$ .
- A graph  $G = (V, E)$  satisfies property *k-regular* if the degree of every agent in  $V$  is  $k$ .
- A graph  $G$  satisfies property *star* if  $G$  is a tree with one internal agent and  $n - 1$  leaves.
- A graph  $G = (V, E)$  satisfies property *bipartite* if  $V$  can be divided into two disjoint and independent sets  $U$  and  $W$  (i.e.,  $U \cap W = \emptyset$  holds and there is no edge connecting two agents in  $U$  or  $W$ ).
- A graph  $G = (V, E)$  satisfies property *line* if  $E = \{(v_0, v_1), (v_1, v_2), (v_2, v_3), \dots, (v_{n-1}, v_n)\}$  for  $V = \{v_1, v_2, \dots, v_n\}$ .
- A graph  $G = (V, E)$  satisfies property *ring* if the degree of every agent in  $V$  is 2.

Let  $gp$  be an arbitrary graph property. The  $gp$  identification problem aims to decide whether a given communication graph  $G$  satisfies property  $gp$ . In the  $gp$  identification problem, the output set is  $Y = \{yes, no\}$ . Recall that the output function  $\gamma$  maps a state of an agent to an output symbol in  $Y$  (i.e., *yes* or *no*). A configuration  $C$  is stable if  $C$  satisfies the following conditions: There exists  $yn \in \{yes, no\}$  such that 1)  $\forall a \in V : \gamma(s(a, C)) = yn$  holds, and 2) for every configuration  $C'$  such that  $C \xrightarrow{*} C'$ ,  $\forall a \in V : \gamma(s(a, C')) = yn$  holds.

An execution  $\Xi = C_0, C_1, C_2, \dots$  solves the  $gp$  identification problem if  $\Xi$  includes a stable configuration  $C_t$  that satisfies the following conditions.

1. If a given graph  $G = (V, E)$  satisfies graph property  $gp$ ,  $\forall a \in V : \gamma(s(a, C_t)) = yes$  holds.
2. If a given graph  $G = (V, E)$  does not satisfy graph property  $gp$ ,  $\forall a \in V : \gamma(s(a, C_t)) = no$  holds.

<sup>1</sup> We use this definition only for the lower bound under weak fairness. For the upper bound, we use a slightly weaker assumption. We show that our proposed protocol for weak fairness works if, for any adjacent agents  $a$  and  $a'$ ,  $a$  and  $a'$  interact infinitely often (i.e., it is possible that, for any interaction between some adjacent agents  $a$  and  $a'$ ,  $a$  becomes an initiator and  $a'$  never becomes an initiator). Note that, in the protocol, if a transition  $(p, q) \rightarrow (p', q')$  exists for  $p \neq q$ , a transition  $(q, p) \rightarrow (q', p')$  also exists.

A protocol  $\mathcal{P}$  solves the  $gp$  identification problem under weak fairness (resp., under global fairness) if every weakly-fair execution (resp., every globally-fair execution) of protocol  $\mathcal{P}$  solves the  $gp$  identification problem.

### 3 Graph Class Identification Protocols

#### 3.1 Tree Identification with No Initial Knowledge under Global Fairness

In this section, we give a tree identification protocol with 18 states and designated initial states under global fairness (see the pseudocode in the appendix and the full version in [26]).

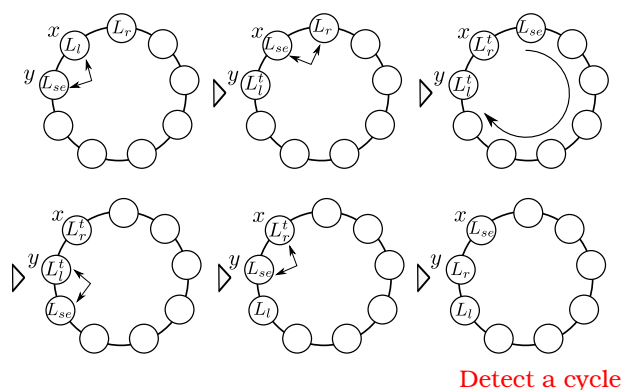
The basic strategy of the protocol is as follows. Initially agents think that the graph is a tree, and, if they detect a cycle, they confirm that the graph is not a tree. To detect a cycle, first, agents elect one leader token, one right token, and one left token. Initially, all agents have right tokens. When two agents with right tokens interact, agents make one of the right tokens transition to a left token. Similarly, when two agents with left tokens interact, agents make one of the left tokens transition to a leader token. When two agents with leader tokens interact, agents delete one of the leader tokens. Agents carry these tokens on a graph by interactions as if each token moves freely on the graph. Thus, by the above behaviors, eventually agents elect one leader token, one right token, and one left token.

Agents behave as if the leader token has an opinion (tree/non-tree), and agents follow the opinion (this opinion is hereinafter referred to as “decision”). Initially the leader token has the decision such that the graph is a tree (i.e., there is no cycle). Since the leader token moves freely on the graph and we assume global fairness, the leader token visits all agents infinitely often. Thus, eventually all agents will know the decision of the leader token.

Agents reset the decision of the leader token if agents notice that the token election is not yet over. Concretely, when two agents with leader tokens interact and delete one of the leader token, agents reset the decision of the remaining leader token. By this behavior, all agents virtually reset their decision because each agent follows the decision of the leader token. Hence, when agents complete the token election, all agents (and the leader token) virtually reset their decision. Now, we explain that, after the token election, agents correctly detect a cycle and the leader token make a correct decision.

After the election, agents repeatedly execute a trial to detect a cycle by using the tokens. The trial starts when an agent with the leader token places the right token and the left token to two adjacent agents  $x$  and  $y$ , respectively. During the trial,  $x$  and  $y$  hold the right token and the left token, respectively. To detect a cycle, agents use the right token and the left token as a single landmark. The right token and the left token correspond to a right side and a left side of the landmark, respectively. If agents can carry the leader token from the right side of the landmark to the left side of the landmark without passing through the landmark, the trial succeeds.

From now, we explain the behaviors of the trial in more details. An image of the trial is shown in Figure 1, where  $L_{se}$ ,  $L_r$ ,  $L_l$ ,  $L_r^t$ , and  $L_l^t$  represent tokens (we will show the details later). To begin with, we explain the start of the trial (the first and second steps of Figure 1). To start the trial, agents place the left token and the right token next to each other. To distinguish between a moving token and a placed token, we use a trial mode. Agents regard right and left tokens in a trial mode as placed tokens. An  $L_r^t$  token (resp., an  $L_l^t$  token) represents the right token (resp., the left token) in the trial mode. An  $L_r$  token (resp., an  $L_l$  token) represents the right token (resp., the left token) in a non-trial mode. An  $L_{se}$  token represents the leader token.



■ **Figure 1** An image of the trial.

More concretely, agents start the trial as follows. When an agent  $x$  having the  $L_l$  token interacts with an agent  $y$  having the leader token, agents make the  $L_l$  token transition to an  $L_l^t$  token, and they exchange their tokens (the first step of Figure 1). Then, if agent  $x$  having the leader token interacts with an agent having the  $L_r$  token, agents make the  $L_r$  token transition to an  $L_r^t$  token, and they exchange their tokens (the second step of Figure 1). By above behaviors, agents place an  $L_r^t$  token next to the  $L_l^t$  token, and a trial of the cycle detection starts.

After that, agents try to carry the leader token to agent  $y$  without passing through agent  $x$  (the third step of Figure 1). To do this, agents try to carry the leader token to an agent having the  $L_l^t$  token. Note that the left and right tokens can move even while agents carry the leader token. However, if they move, they transition to the non-trial mode. Thus, if an agent having the leader token interacts with an agent having the  $L_l^t$  token, agents confirm that the  $L_l^t$  token is still placed at  $y$  (the fourth step of Figure 1). Then, to confirm that the  $L_r^t$  token is also still placed at  $x$ , agent  $y$  having the leader token tries to interact with an agent having the  $L_r^t$  token. Since the right token may move and the leader token may pass through agent  $x$  without meeting the right token, this confirmation is necessary. If agents succeed both confirmations, agents succeed the trial and decide that there is a cycle (the fifth step of Figure 1). Hence, in the case, the leader token makes a decision that the given graph is not a tree, and the decision is conveyed to all agents. Since each token moves freely on the graph and we assume global fairness, agents perform the trial on any place (i.e., agents place the left token and the right token on any adjacent agents). Thus, if there is a cycle, eventually agents decide that the given graph is not a tree. Recall that initially all agents think that the given graph is a tree. Hence, unless the trial succeeds, all agents continue to think that the given graph is a tree.

► **Theorem 1.** *There exists a protocol with constant states and designated initial states that solves the tree identification problem under global fairness.*

### 3.2 $k$ -regular Identification with Knowledge of $P$ under Global Fairness

In this subsection, we give a  $k$ -regular identification protocol with  $O(k \log P)$  states and designated initial states under global fairness (see the pseudocode in the appendix and the full version in [26]).

The basic strategy of the protocol is as follows. First, agents elect a leader token. In this protocol, agents with leader tokens leave some information in agents. To keep only the information that is left after completion of the election, we introduce *level* of an agent. If an

agent at level  $i$  has the leader token, we say that the leader token is at level  $i$ . Agents with leader tokens leave the information with their levels. Before agents complete the election of leader tokens, agents keep increasing their levels, and agents discard the information with smaller levels when agents increase their levels. When agents complete the election of leader tokens, the agent with the leader token is the only agent that has the largest level. Then, all agents eventually converge to the level. We guarantee that the maximum level of agents is  $\lfloor \log P \rfloor$ . Since agents discard the information with smaller levels, agents virtually discard any information that was left before agents complete the election. From now on, we consider configurations after agents elect a leader token and discard any outdated information.

Now, we explain how the protocol solves the  $k$ -regular identification problem by using the leader token. First, an agent with the leader token examines whether its degree is at least  $k$ , and whether its degree is at least  $k + 1$ . If the agent confirms that its degree is at least  $k$  but does not confirm that its degree is at least  $k + 1$ , then the agent thinks that its degree is  $k$ . Since the leader token moves freely on the graph and we assume global fairness, eventually each agent confirms whether its degree is  $k$ . The agent with the leader token examines whether its degree is at least  $k$  as follows: An agent  $a$  with the leader token checks whether  $a$  can interact with  $k$  different agents. To check it, agent  $a$  with the leader token marks adjacent agents and counts how many times  $a$  has marked. Concretely, when agent  $a$  having the leader token interacts with an agent  $b$ , agent  $a$  marks agent  $b$  by making  $b$  change to a marked state. Agent  $a$  counts how many times  $a$  interacts with an agent having a non-marked state (hereinafter referred to as “a non-marked agent”). If agent  $a$  having the leader token interacts with  $k$  non-marked agents successively,  $a$  decides that  $a$  can interact with  $k$  different agents (i.e., its degree is at least  $k$ ).

If an agent confirms that its degree is at least  $k$ , the agent stores this information locally. To do this, we introduce a variable  $loc_a$  at agent  $a$ : Variable  $loc_a \in \{yes, no\}$ , initialized to  $no$ , represents whether the degree of agent  $a$  is at least  $k$ . If  $loc_a = yes$  holds, agent  $a$  thinks that its degree is at least  $k$ . If an agent  $a$  confirms that its degree is at least  $k$ , agent  $a$  makes  $loc_a$  transition from  $no$  to  $yes$ .

Next, we show how agents decide whether the graph is  $k$ -regular. In this protocol, first an agent with the leader token decides whether the graph is  $k$ -regular, and then the decision is conveyed to all agents by the leader token. We use variable  $reg_a$  at agent  $a$  for the decision: Variable  $reg_a \in \{yes, no\}$ , initialized to  $no$ , represents the decision of the  $k$ -regular graph. If  $reg_a = yes$  holds for agent  $a$ , then  $\gamma(s_a) = yes$  holds. If  $reg_a = no$  holds, then  $\gamma(s_a) = no$  holds. Whenever an agent  $a$  with the leader token makes  $loc_a$  transition to  $yes$ , agent  $a$  makes  $reg_a$  transition to  $yes$ . If an agent  $a$  with the leader token finds an agent  $b$  such that  $loc_b = no$  or its degree is at least  $k + 1$ , agents reset  $reg_a$  to  $no$ . Note that, since all agents follow the decision of the leader token, this behavior practically resets  $reg$  of each agent. If there is such agent  $b$ , agent  $a$  with the leader token eventually finds agent  $b$  since the leader token moves freely on the graph. Hence, if the graph is not  $k$ -regular,  $reg$  of the leader token (i.e.,  $reg_a$  such that agent  $a$  has the leader token) transitions to  $no$  infinitely often. On the other hand, if the graph is  $k$ -regular, eventually  $loc_a$  of each agent  $a$  transitions from  $no$  to  $yes$ . Let us consider a configuration where  $loc$  of each agent other than an agent  $x$  is  $yes$  and  $loc_x$  is  $no$ . After the configuration, when agent  $x$  makes  $loc_x$  and  $reg_x$  transition to  $yes$ , agent  $x$  has the leader token (i.e.,  $reg$  of the leader token transitions to  $yes$ ). Hence, since there is no agent such that its  $loc$  is  $no$  or its degree is at least  $k + 1$ ,  $reg$  of the leader token never transitions to  $no$  afterwards and thus  $reg$  of the leader token converges to  $yes$ . Thus, since agents convey the decision of the leader token to all agents, eventually all agents make a correct decision.

► **Remark.** We have introduced the level of agents to reset all agents virtually. In the case of tree identification, only the leader token needs to be reset because, after the leader token is reset, the leader token is not affected by incorrect information (the decision of agent in this case). On the other hand, in the case of  $k$ -regular identification, all agents need to be reset because, even after the leader token is reset, the leader token is affected by incorrect information (the value of  $loc$  in this case). More concretely, the leader token has to refer to variable  $loc$  of agent in order to understand whether all agents have been examined. Hence, since agents may store an incorrect value in  $loc$ , the leader token may make a wrong decision unless the agents are reset.

To count the degree, agents require  $O(k)$  states, and the maximum level of agents is  $\lfloor \log P \rfloor$ . Hence, the protocol works with  $O(k \log P)$  states.

► **Theorem 2.** *If the upper bound  $P$  of the number of agents is given, there exists a protocol with  $O(k \log P)$  states and designated initial states that solves the  $k$ -regular identification problem under global fairness.*

When the number of agents  $n$  is given, the protocol works even if the maximum level is  $\lfloor \log n \rfloor$ . Thus, we have the following theorem.

► **Theorem 3.** *If the number of agents  $n$  is given, there exists a protocol with  $O(k \log n)$  states and designated initial states that solves the  $k$ -regular identification problem under global fairness.*

### 3.3 Star Identification with Knowledge of $n$ under Weak Fairness

In this subsection, we give a star identification protocol with  $O(n)$  states and designated initial states under weak fairness (see the pseudocode in the appendix and the full version in [26]). In this protocol, the number of agents  $n$  is given. Since a given graph is a star if  $n \leq 2$  holds, we consider the case where  $n$  is at least 3.

The basic strategy of the protocol is as follows. Initially, each agent thinks that the given graph is not a star. First, agents elect an agent with degree two or more as a central agent (i.e., an agent that connects to all other agents in the star graph). Then, by counting the number of agents adjacent to the central agent, agents examine whether there is a star subgraph in the given graph such that the subgraph consists of  $n$  agents. Concretely, if the central agent confirms by counting that there are  $n - 1$  adjacent agents, agents confirm that there is the subgraph. In this case, agents think that the given graph is a star. Then, if two agents other than the central agent interact, agents decide that the graph is not a star. If such an interaction does not occur, agents continue to think that the given graph is a star.

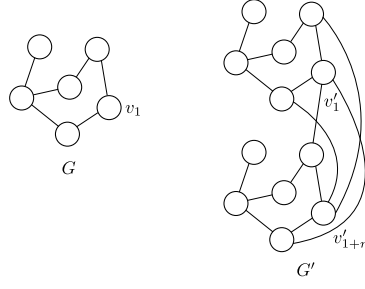
To count  $n - 1$  agents adjacent to the central agents, agents require  $O(n)$  states. Hence, the protocol works with  $O(n)$  states.

► **Theorem 4.** *There exists a protocol with  $O(n)$  states and designated initial states that solves the star identification problem under weak fairness if the number of agents  $n$  is given.*

## 4 Impossibility Results

### 4.1 A Common Property of Graph Class Identification Protocols for Impossibility Results

In this subsection, we present a common property that holds for protocols with designated initial states under weak fairness.



■ **Figure 2** An example of graphs  $G$  and  $G'$ .

With designated initial states under weak fairness, we assume that a protocol  $\mathcal{P}$  solves some of the graph class identification problems. From now, we show that, with  $\mathcal{P}$ , there exists a case where agents cannot distinguish between some different connected graphs. Note that  $\mathcal{P}$  has no constraints for an initial knowledge (i.e., for some integer  $x$ ,  $\mathcal{P}$  is  $\mathcal{P}_{n=x}$ ,  $\mathcal{P}_{P=x}$ , or a protocol with no initial knowledge). Because of space constraints, we omit the details of the proof (see the full version in the [26]).

► **Lemma 5.** *Let us consider a communication graph  $G = (V, E)$ , where  $V = \{v_1, v_2, v_3, \dots, v_n\}$ . Let  $G' = (V', E')$  be a communication graph that satisfies the following, where  $V' = \{v'_1, v'_2, v'_3, \dots, v'_{2n}\}$ .*

■  $E' = \{(v'_x, v'_y), (v'_{x+n}, v'_{y+n}) \in V' \times V' \mid (v_x, v_y) \in E\} \cup \{(v'_1, v'_{z+n}), (v'_{1+n}, v'_z) \in V' \times V' \mid (v_1, v_z) \in E\}$  (Figure 2 shows an example of the graphs).

Let  $\Xi$  be a weakly-fair execution of  $\mathcal{P}$  with  $G$ . If there exists a configuration  $C$  of  $\Xi$  after which  $\forall v \in V : \gamma(s(v)) = yn \in \{\text{yes}, \text{no}\}$  holds, there exists an execution  $\Xi'$  of  $\mathcal{P}$  with  $G'$  such that there exists a configuration  $C'$  of  $\Xi'$  after which  $\forall v' \in V' : \gamma(s(v')) = yn$  holds.

**Proof.** (Proof sketch) With  $G'$ , we consider a particular weakly-fair execution  $\Xi'$ . In  $\Xi'$ , agents repeat the following four sub-executions: 1) agents  $v'_1, v'_2, \dots, v'_n$  behave similarly to  $\Xi$ , 2) agents  $v'_{1+n}, v'_{2+n}, \dots, v'_{2n}$  behave similarly to  $\Xi$ , 3) agent  $v'_{1+n}$  and agents  $v'_2, v'_3, \dots, v'_n$  behave similarly to  $\Xi$  by joining  $v'_{1+n}$  instead of  $v'_1$ , and 4) agent  $v'_1$  and agents  $v'_{2+n}, v'_{3+n}, \dots, v'_{2n}$  behave similarly to  $\Xi$  by joining  $v'_1$  instead of  $v'_{1+n}$ . Since  $v'_1$  (resp.,  $v'_{1+n}$ ) can join interactions instead of  $v'_{1+n}$  (resp.,  $v'_1$ ) by the definition of  $G$  and  $G'$ , this execution is possible. Clearly, in  $\Xi'$ , the decision of each agent is the same as the decision of agent in  $\Xi$ . Therefore, the lemma holds. ◀

## 4.2 Impossibility with Knowledge of $P$ under Weak Fairness

For the purpose of the contradiction, we assume that, for an integer  $x$ , there exists a protocol  $\mathcal{P}_{P=x}$  that solves some of the graph class identification problems with designated initial states under weak fairness. We can apply Lemma 5 to  $\mathcal{P}_{P=x}$  because we can apply the same protocol  $\mathcal{P}_{P=x}$  to both  $G$  and  $G'$  in Lemma 5. Clearly, we can construct  $G$  and  $G'$  in Lemma 5 such that, for any of properties *line*, *ring*, *tree*, *k-regular*, and *star*,  $G$  is a graph that satisfies the property, and  $G'$  is a graph that does not satisfy the property. Therefore, we have the following theorem.

► **Theorem 6.** *Even if the upper bound of the number of agents is given, there exists no protocol that solves the line, ring, k-regular, star, or tree identification problem with the designated initial states under weak fairness.*



Note that, in Theorem 6, the bipartite identification problem is not included. However, we show later that there is no protocol that solves the bipartite identification problem even if the number of agents is given.

### 4.3 Impossibility with Knowledge of $n$ under Weak Fairness

In this subsection, we show that, even if the number of agents  $n$  is given, there exists no protocol that solves the *line*, *ring*, *k-regular*, *tree*, or *bipartite* identification problem with designated initial states under weak fairness.

#### Case of Line, Ring, $k$ -regular, and Tree

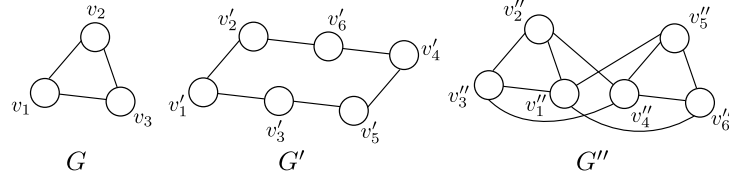
First, we show that there exists no protocol that solves the *line*, *ring*, *k-regular*, or *tree* identification problem. Concretely, we show that there is a case where a line graph and a ring graph are not distinguishable. To show this, first we give some notations. Let  $G = (V, E)$  be a line graph with four agents, where  $V = \{v_1, v_2, v_3, v_4\}$  and  $E = \{(v_1, v_2), (v_2, v_3), (v_3, v_4)\}$ . Let  $G' = (V', E')$  be a ring graph with four agents, where  $V' = \{v'_1, v'_2, v'_3, v'_4\}$  and  $E' = \{(v'_1, v'_2), (v'_2, v'_3), (v'_3, v'_4), (v'_4, v'_1)\}$ . Let  $s_0$  be an initial state of agents. Let us consider a transition sequence  $T = (s_0, s_0) \rightarrow (s_{a_1}, s_{b_1}), (s_{b_1}, s_{a_1}) \rightarrow (s_{b_2}, s_{a_2}), (s_{a_2}, s_{b_2}) \rightarrow (s_{a_3}, s_{b_3}), (s_{b_3}, s_{a_3}) \rightarrow (s_{b_4}, s_{a_4}), \dots$ . Since the number of states is finite, there are  $i$  and  $j$  such that  $s_{a_i} = s_{a_j}$ ,  $s_{b_i} = s_{b_j}$ , and  $i < j$  hold. Let  $sa$  and  $sb$  be states such that  $sa = s_{a_i} = s_{a_j}$  and  $sb = s_{b_i} = s_{b_j}$  hold.

Because of space constraints, we omit the details of the proof (see the full version in the [26]). The proof sketch is as follows: We construct a particular execution  $\Xi$  with  $G$  such that the decision of each agent converges to  $yn \in \{yes, no\}$ . In  $\Xi$ , agents repeat the following three sub-executions: 1) agents  $v_1$  and  $v_2$  interact repeatedly until  $v_1$  and  $v_2$  obtain  $sa$  and  $sb$ , respectively, 2) agents  $v_3$  and  $v_4$  interact repeatedly until  $v_3$  and  $v_4$  obtain  $sa$  and  $sb$ , respectively, and 3)  $v_3$  and  $v_2$  interact repeatedly until  $v_3$  and  $v_2$  obtain  $sa$  and  $sb$  again, respectively. Next, we construct a particular execution  $\Xi'$  with  $G'$ . In  $\Xi'$ , agents repeat the following four sub-executions: 1) agents  $v'_1$  and  $v'_2$  interact repeatedly until  $v'_1$  and  $v'_2$  obtain  $sa$  and  $sb$ , respectively, 2) agents  $v'_3$  and  $v'_4$  interact repeatedly until  $v'_3$  and  $v'_4$  obtain  $sa$  and  $sb$ , respectively, 3)  $v'_3$  and  $v'_2$  interact repeatedly until  $v'_3$  and  $v'_2$  obtain  $sa$  and  $sb$  again, respectively, and 4)  $v'_1$  and  $v'_4$  interact repeatedly until  $v'_1$  and  $v'_4$  obtain  $sa$  and  $sb$  again, respectively. From the definition of  $sa$  and  $sb$ , we can construct those executions such that the executions satisfy weak fairness and agents converge to the decision  $yn$ .

► **Lemma 7.** *There exists a weakly-fair execution  $\Xi'$  of  $\mathcal{P}$  with  $G'$  such that  $\forall v' \in V' : \gamma(s(v')) = yn$  holds in a stable configuration of  $\Xi'$ .*

Note that, even if the number of agents is given, Lemma 7 holds because  $|V| = |V'| = 4$  holds in the lemma. In Lemma 7,  $G$  is a line graph and a tree graph whereas  $G'$  is neither a line graph nor a tree graph. Furthermore,  $G'$  is a ring graph and a 2-regular graph whereas  $G$  is neither a ring graph nor a 2-regular graph. Hence, by Lemma 7, there is no protocol that solves the *line*, *ring*, *tree*, or *k-regular* identification problem, and thus we have the following theorem.

► **Theorem 8.** *Even if the number of agents  $n$  is given, there exists no protocol that solves the *line*, *ring*, *k-regular*, or *tree* identification problem with designated initial states under weak fairness.*



■ **Figure 3** Graphs  $G$ ,  $G'$ , and  $G''$ .

### Case of Bipartite

Next, we show that there exists no protocol that solves the bipartite identification problem. For the purpose of the contradiction, we assume that there exists a protocol  $\mathcal{P}_{n=6}$  that solves the bipartite identification problem with designated initial states under weak fairness if the number of agents 6 is given.

We define a ring graph  $G = (V, E)$  with three agents, a ring graph  $G' = (V', E')$  with 6 agents, and a graph  $G'' = (V'', E'')$  with 6 agents as follows:

- $V = \{v_1, v_2, v_3\}$  and  $E = \{(v_1, v_2), (v_2, v_3), (v_3, v_1)\}$ .
- $V' = \{v'_1, v'_2, v'_3, v'_4, v'_5, v'_6\}$  and  $E' = \{(v'_1, v'_2), (v'_2, v'_6), (v'_6, v'_4), (v'_4, v'_5), (v'_5, v'_3), (v'_3, v'_1)\}$ .
- $V'' = \{v''_1, v''_2, v''_3, v''_4, v''_5, v''_6\}$  and  $E'' = \{(v''_x, v''_y), (v''_{x+n}, v''_{y+n}) \in V'' \times V'' \mid (v_x, v_y) \in E\} \cup \{(v''_1, v''_5), (v''_1, v''_6), (v''_4, v''_2), (v''_4, v''_3)\}$ .

Figure 3 shows graphs  $G$ ,  $G'$ , and  $G''$ .

From now, we show that there exists an execution  $\Xi''$  of  $\mathcal{P}_{n=6}$  with  $G''$  such that all agents converge to *yes* whereas  $G''$  does not satisfy *bipartite*. To show this, we first show that, in any execution  $\Xi$  of  $\mathcal{P}_{n=6}$  with  $G$  (i.e., the protocol for 6 agents is applied to a population consisting of 3 agents), all agents converge to *yes*. To prove this, we borrow the proof technique in [19]. In [19], Fischer and Jiang proved the impossibility of leader election for a ring graph.

► **Lemma 9.** *In any weakly-fair execution  $\Xi$  of  $\mathcal{P}_{n=6}$  with  $G$ , all agents converge to *yes*. That is, in  $\Xi$ , there exists  $C_t$  such that  $\forall v \in V : \gamma(s(v, C_i)) = \text{yes}$  holds for  $i \geq t$ .*

**Proof sketch.** For  $\Xi$ , we construct an execution  $\Xi'$  of  $\mathcal{P}_{n=6}$  with  $G'$  such that  $v'_1, v'_2$ , and  $v'_3$  behave similarly to  $v_1, v_2$ , and  $v_3$  in  $\Xi$ , respectively, and  $v'_4, v'_5$ , and  $v'_6$  also behave similarly to  $v_1, v_2$ , and  $v_3$  in  $\Xi$ , respectively. Note that agents  $v'_1, v'_2$ , and  $v'_3$  and agents  $v'_4, v'_5$ , and  $v'_6$  operate in parallel. Since  $v'_2$  (resp.,  $v'_5$ ) is not adjacent to  $v'_3$  (resp.,  $v'_6$ ),  $v'_2$  (resp.,  $v'_5$ ) cannot interact with  $v'_3$  (resp.,  $v'_6$ ). When  $v'_2$  (resp.,  $v'_5$ ) should interact with  $v'_3$  (resp.,  $v'_6$ ),  $v'_2$  (resp.,  $v'_5$ ) interacts with  $v'_6$  instead of  $v'_3$  (resp.,  $v'_3$  instead of  $v'_6$ ). Since  $v'_2$  and  $v'_5$  (resp.,  $v'_3$  and  $v'_6$ ) behave similarly to  $v_2$  (resp.,  $v_3$ ),  $v'_2$  and  $v'_5$  (resp.,  $v'_3$  and  $v'_6$ ) have the same state. Hence, even if  $v'_2$  (resp.,  $v'_5$ ) interacts with  $v'_6$  instead of  $v'_3$  (resp.,  $v'_3$  instead of  $v'_6$ ),  $v'_2$  and  $v'_3$  (resp.,  $v'_5$  and  $v'_6$ ) can transition similarly to  $v_2$  and  $v_3$ .

In  $\Xi'$ , since the number of agents is given correctly, a stable configuration exists. Hence, since  $G'$  is a bipartite graph, all agents converge to *yes* in  $\Xi'$ . This implies that all agents converge to *yes* even in  $\Xi$ . ◀

Now, we show that there exists execution  $\Xi''$  of  $\mathcal{P}_{n=6}$  with  $G''$  such that all agents converge to *yes*. We show this by applying Lemma 5 to protocol  $\mathcal{P}_{n=6}$  and graphs  $G$  and  $G''$ . Graphs  $G$  and  $G''$  satisfy the condition of  $G$  and  $G'$  in Lemma 5, and the protocol  $\mathcal{P}_{n=6}$  satisfies the condition of protocol  $\mathcal{P}$  in Lemma 5. Thus, we can apply Lemma 5 to protocol

$\mathcal{P}_{n=6}$  and graphs  $G$  and  $G''$ . By applying Lemma 5, since all agents converge to *yes* in an execution of  $\mathcal{P}_{n=6}$  with  $G$  by Lemma 9, there exists a weakly-fair execution  $\Xi''$  of  $\mathcal{P}_{n=6}$  with  $G''$  in which all agents converge to *yes*.

► **Lemma 10.** *With the designated initial states, there exists a weakly-fair execution  $\Xi''$  of  $\mathcal{P}_{n=6}$  with  $G''$  such that  $\forall v'' \in V'' : \gamma(s(v'')) = \text{yes}$  in a stable configuration.*

Graph  $G''$  does not satisfy *bipartite*. Thus, from Lemma 10,  $\mathcal{P}_{n=6}$  is incorrect. Therefore, we have the following theorem.

► **Theorem 11.** *Even if the number of agents  $n$  is given, there exists no protocol that solves the bipartite identification problem with the designated initial states under weak fairness.*

#### 4.4 Impossibility with Arbitrary Initial States

In this subsection, we show that, even if the number of agents  $n$  is given, there exists no protocol that solves the *line, ring,  $k$ -regular, star, tree, or bipartite* identification problem with arbitrary initial states under global fairness.

For the purpose of the contradiction, we assume that there exists a protocol  $\mathcal{P}$  that solves some of the above graph class identification problems with arbitrary initial states under global fairness if the number of agents  $n$  is given. From now, we show that there are two executions  $\Xi$  and  $\Xi'$  of  $\mathcal{P}$  such that the decision of all agents in the executions converges to the same value whereas  $\Xi$  and  $\Xi'$  are for graphs  $G$  and  $G' (\neq G)$ , respectively.

► **Lemma 12.** *Let  $G = (V, E)$  and  $G' = (V', E')$  be connected graphs that satisfy the following condition, where  $V = \{v_1, v_2, v_3, \dots, v_n\}$  and  $V' = \{v'_1, v'_2, v'_3, \dots, v'_n\}$ .*

■ *For some edge  $(v_\alpha, v_\beta)$  in  $E$ ,  $E' = \{(v'_x, v'_y) \in V' \times V' \mid (v_x, v_y) \in E\} \setminus \{(v'_\alpha, v'_\beta)\}$ .*

*If there exists a globally-fair execution  $\Xi$  of  $\mathcal{P}$  with  $G$  such that  $\forall v \in V : \gamma(s(v)) = yn \in \{\text{yes}, \text{no}\}$  holds in a stable configuration of  $\Xi$ , there exists a globally-fair execution  $\Xi'$  of  $\mathcal{P}$  with  $G'$  such that  $\forall v' \in V' : \gamma(s(v')) = yn$  holds in a stable configuration of  $\Xi'$ .*

**Proof.** Let  $\Xi = C_0, C_1, C_2, \dots$  be a globally-fair execution of  $\mathcal{P}$  with  $G$  such that  $\forall v \in V : \gamma(s(v)) = yn \in \{\text{yes}, \text{no}\}$  holds in a stable configuration  $C_t$ . For the purpose of the contradiction, we assume that there exists no execution of  $\mathcal{P}$  with  $G'$  such that  $\forall v' \in V' : \gamma(s(v')) = yn$  holds in a stable configuration.

Let us consider an execution  $\Xi' = C'_0, C'_1, C'_2, \dots, C'_{t'}, \dots$  of  $\mathcal{P}$  with  $G'$  such that, for  $1 \leq i \leq n$ ,  $s(v'_i, C'_0) = s(v_i, C_t)$  holds and  $C'_{t'}$  is a stable configuration. By the assumption,  $\exists v'_z \in V' : \gamma(s(v'_z, C'_{t'})) = yn' (\neq yn)$  holds.

Next, let us consider an execution  $\Xi'' = C''_0, C''_1, C''_2, \dots, C''_t, \dots$  of  $\mathcal{P}$  with  $G$  as follows:

- For  $0 \leq i \leq t$ ,  $C''_i = C_i$  holds (i.e., agents behave similarly to  $\Xi$ ).
- For  $t < i \leq t + t'$ , when  $v'_x$  interacts with  $v'_y$  at  $C'_{i-t-1} \rightarrow C'_{i-t}$ ,  $v_x$  interacts with  $v_y$  at  $C''_{i-1} \rightarrow C''_i$ . This is possible because  $E' \subset E$  holds.

Since  $C_t$  is a stable configuration,  $C''_t$  is also a stable configuration and  $\forall v \in V : \gamma(s(v, C''_t)) = yn$  holds. Since agents behave similarly to  $\Xi'$  after  $C''_t$ ,  $\gamma(s(v_z, C''_{t+t'})) = yn'$  holds. This contradicts the fact that  $C''_t$  is a stable configuration. ◀

We can construct a non-line graph, a non-ring graph, a non-star graph, a non-tree graph, and a non-bipartite graph by adding an edge to a line graph, a ring graph, a star graph, a tree graph, and a bipartite graph, respectively. Moreover, we can construct a  $k$ -regular graph by adding an edge to some non- $k$ -regular graph. From Lemma 12, there is a case where the decision of all agents converges to the same value for each pair of graphs. Therefore, we have the following theorem.

► **Theorem 13.** *There exists no protocol that solves the line, ring,  $k$ -regular, star, tree, or bipartite identification problem with arbitrary initial states under global fairness.*

## 5 Concluding Remarks

In this paper, we consider the graph class identification problems on various assumptions. We have interesting open problems for future researches as follows:

- What is the space complexity of  $k$ -regular identification problem under global fairness with designated initial states and no initial knowledge.
- What is the time complexity of graph class identification problems?
- Are there some graph class identification protocols for other graph properties?

---

## References

- 1 Dan Alistarh and Rati Gelashvili. Polylogarithmic-time leader election in population protocols. In *Proceedings of the 42nd International Colloquium on Automata, Languages, and Programming*, pages 479–491, 2015.
- 2 Dan Alistarh, Rati Gelashvili, and Joel Rybicki. Fast graphical population protocols. *arXiv preprint*, 2021. [arXiv:2102.08808](https://arxiv.org/abs/2102.08808).
- 3 Dana Angluin, James Aspnes, Melody Chan, Michael J Fischer, Hong Jiang, and René Peralta. Stably computable properties of network graphs. In *Proceedings of International Conference on Distributed Computing in Sensor Systems*, pages 63–74, 2005.
- 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. A simple population protocol for fast robust approximate majority. *Distributed Computing*, 21(2):87–102, 2008.
- 6 Dana Angluin, James Aspnes, Michael J Fischer, and Hong Jiang. Self-stabilizing population protocols. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 3(4):13, 2008.
- 7 James Aspnes, Joffroy Beauquier, Janna Burman, and Devan Sohier. Time and space optimal counting in population protocols. In *Proceedings of International Conference on Principles of Distributed Systems*, pages 13:1–13:17, 2016.
- 8 Joffroy Beauquier, Janna Burman, Simon Claviere, and Devan Sohier. Space-optimal counting in population protocols. In *Proceedings of International Symposium on Distributed Computing*, pages 631–646, 2015.
- 9 Joffroy Beauquier, Julien Clement, Stephane Messika, Laurent Rosaz, and Brigitte Rozoy. Self-stabilizing counting in mobile sensor networks with a base station. In *Proceedings of International Symposium on Distributed Computing*, pages 63–76, 2007.
- 10 Petra Berenbrink, Robert Elsässer, Tom Friedetzky, Dominik Kaaser, Peter Kling, and Tomasz Radzik. Time-space trade-offs in population protocols for the majority problem. *Distributed Computing*, pages 1–21, 2020.
- 11 Petra Berenbrink, George Giakkoupis, and Peter Kling. Optimal time and space leader election in population protocols. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*, pages 119–129, 2020.
- 12 Petra Berenbrink, Dominik Kaaser, and Tomasz Radzik. On counting the population size. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 43–52, 2019.
- 13 Olivier Bournez, Jérémie Chalopin, Johanne Cohen, Xavier Koegler, and Mikael Rabie. Population protocols that correspond to symmetric games. *International Journal of Unconventional Computing*, 9, 2013.
- 14 Shukai Cai, Taisuke Izumi, and Koichi Wada. How to prove impossibility under global fairness: On space complexity of self-stabilizing leader election on a population protocol model. *Theory of Computing Systems*, 50(3):433–445, 2012.

- 15 Ioannis Chatzigiannakis, Othon Michail, and Paul G. Spirakis. Stably decidable graph languages by mediated population protocols. In *Stabilization, Safety, and Security of Distributed Systems - 12th International Symposium, SSS*, volume 6366 of *Lecture Notes in Computer Science*, pages 252–266. Springer, 2010. doi:10.1007/978-3-642-16023-3\_21.
- 16 Hsueh-Ping Chen and Ho-Lin Chen. Self-stabilizing leader election. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 53–59, 2019.
- 17 Hsueh-Ping Chen and Ho-Lin Chen. Self-stabilizing leader election in regular graphs. In *Proceedings of the 39th Symposium on Principles of Distributed Computing*, pages 210–217, 2020.
- 18 David Doty and David Soloveichik. Stable leader election in population protocols requires linear time. *Distributed Computing*, 31(4):257–271, 2018.
- 19 Michael Fischer and Hong Jiang. Self-stabilizing leader election in networks of finite-state anonymous agents. In *International Conference On Principles Of Distributed Systems*, pages 395–409. Springer, 2006.
- 20 Leszek Gąsieniec, David Hamilton, Russell Martin, Paul G Spirakis, and Grzegorz Stachowiak. Deterministic population protocols for exact majority and plurality. In *Proceedings of International Conference on Principles of Distributed Systems*, pages 14:1–14:14, 2016.
- 21 Leszek Gąsieniec, Grzegorz Stachowiak, and Przemyslaw Uznanski. Almost logarithmic-time space optimal leader election in population protocols. In *The 31st ACM on Symposium on Parallelism in Algorithms and Architectures*, pages 93–102. ACM, 2019.
- 22 Tomoko Izumi, Keigo Kinpara, Taisuke Izumi, and Koichi Wada. Space-efficient self-stabilizing counting population protocols on mobile sensor networks. *Theoretical Computer Science*, 552:99–108, 2014.
- 23 Othon Michail and Paul G Spirakis. Simple and efficient local codes for distributed stable network construction. *Distributed Computing*, 29(3):207–237, 2016.
- 24 Satoshi Murata, Akihiko Konagaya, Satoshi Kobayashi, Hirohide Saito, and Masami Hagiya. Molecular robotics: A new paradigm for artifacts. *New Generation Computing*, 31(1):27–45, 2013.
- 25 Yuichi Sudo, Masahiro Shibata, Junya Nakamura, Yonghwan Kim, and Toshimitsu Masuzawa. Self-stabilizing population protocols with global knowledge. *IEEE Transactions on Parallel and Distributed Systems*, 2021.
- 26 Hiroto Yasumi, Fukuhito Ooshita, and Michiko Inoue. Population protocols for graph class identification problems, 2021. arXiv:2111.05111.

## A Pseudocode of Protocols

The descriptions of pseudocodes in this appendix are appeared in [26].

### A.1 Tree Identification Protocol

Algorithm 1 shows the tree identification protocol in Subsection 3.1. The roles of the variables at agent  $a$  in Protocol 1 are as follows.

- $LF_a \in \{L_{se}, L_l, L_r, L_{se}^t, L_{se'}^t, L_{se'}, L_l^t, L_r^t, \phi\}$ : Variable  $LF_a$ , initialized to  $L_r$ , represents a token held by agent  $a$ . If  $LF_a$  is not  $\phi$ , agent  $a$  has  $LF_a$  token. There are three types of tokens: a leader token ( $L_{se}, L_{se}^t, L_{se'}^t$ , and  $L_{se'}$ ), a left token ( $L_l$  and  $L_l^t$ ), and a right token ( $L_r$  and  $L_r^t$ ).  $L_l$ , and  $L_r$  tokens are the tokens in non-trial modes.  $L_l^t$  and  $L_r^t$  tokens represent the left token and the right token in the trial mode, respectively.  $L_{se}^t$ ,  $L_{se'}^t$ , and  $L_{se'}$  tokens represent a progress of a trial of the cycle detection.  $L_{se}^t$  token represents that the left token has been placed.  $L_{se'}^t$  token represents that the right token has been placed.  $L_{se'}$  token represents that the token confirms that the  $L_l^t$  token is still placed on the certain agent.  $\phi$  represents no token.

## 13:16 Population Protocols for Graph Class Identification Problems

- $tre_a \in \{yes, no\}$ : Variable  $tre_a$ , initialized to *yes*, represents a decision of the tree. If  $tre_a = yes$  holds for agent  $a$ , then  $\gamma(s_a) = yes$  holds (i.e.,  $a$  decides that the given graph is a tree). If  $tre_a = no$  holds, then  $\gamma(s_a) = no$  holds (i.e.,  $a$  decides that the given graph is not a tree).

The protocol uses 18 states because the number of values taken by variable  $LF_a$  is 9 and the number of values taken by variable  $tre_a$  is 2.

■ **Algorithm 1** A tree identification protocol (1/2).

**Variables at an agent  $a$ :**

$LF_a \in \{L_{se}, L_l, L_r, L_{se}^t, L_{se'}^t, L_{se'}, L_l^t, L_r^t, \phi\}$ : Token held by the agent, initialized to  $L_r$ .

$tre_a \in \{yes, no\}$ : Decision of the tree, initialized to *yes*.

```

1: when agent  $a$  interacts with agent  $b$  do
  { The election of tokens }
2:   if  $LF_a, LF_b \in \{L_r^t, L_r\}$  then
3:      $LF_b \leftarrow L_l$ 
4:   else if  $LF_a, LF_b \in \{L_l^t, L_l\}$  then
5:      $LF_b \leftarrow L_{se}$ 
6:   else if  $LF_a, LF_b \in \{L_{se}, L_{se}^t, L_{se'}^t, L_{se'}\}$  then
7:      $LF_a \leftarrow L_{se}, LF_b \leftarrow \phi$ 
8:      $tre_a \leftarrow yes$ 
  { Movement of tokens }
9:   else if  $LF_a \neq \phi \wedge LF_b = \phi$  then
10:    if  $LF_a \in \{L_{se}, L_{se}^t, L_{se'}^t, L_{se'}\}$  then
11:       $tre_b \leftarrow tre_a$ 
12:    end if
13:    if  $LF_a = L_\kappa^t$  for  $\kappa \in \{l, r\}$  then
14:       $LF_a \leftarrow L_\kappa$ 
15:    else if  $LF_a = L_{se'} \vee LF_a = L_{se'}^t$  then
16:       $LF_a \leftarrow L_{se}$ 
17:    end if
18:     $LF_a \leftrightarrow LF_b$  *
  { Decision }
19:   else if  $LF_a = L_{se} \wedge LF_b = L_l$  then
20:      $LF_a \leftarrow L_l^t, LF_b \leftarrow L_{se}'$ 
21:      $tre_b \leftarrow tre_a$ 
22:   else if  $LF_a = L_{se'} \wedge LF_b = L_r$  then
23:      $LF_a \leftarrow L_r^t, LF_b \leftarrow L_{se}^t$ 
24:      $tre_b \leftarrow tre_a$ 
25:   else if  $LF_a = L_{se}^t \wedge LF_b = L_l^t$  then
26:      $LF_a \leftarrow L_l, LF_b \leftarrow L_{se}'^t$ 
27:      $tre_b \leftarrow tre_a$ 
28:   else if  $LF_a = L_{se'}^t \wedge LF_b = L_r^t$  then
29:      $LF_a \leftarrow L_r, LF_b \leftarrow L_{se}$ 
30:      $tre_b \leftarrow no$ 

```

\*  $p \leftrightarrow q$  means that  $p$  and  $q$  exchange values.

▷ Continued on the next page

■ **Algorithm 1** A tree identification protocol (2/2).

---

```

31:   else if  $LF_a \neq \phi \wedge LF_b \neq \phi$  then
32:     if  $LF_{ab} \in \{L_{se}, L_{se}^t, L_{se'}^t, L_{se'}\}$  for  $ab \in \{a, b\}$  then
33:        $tre_a \leftarrow tre_{ab}, tre_b \leftarrow tre_{ab}$ 
34:     end if
35:     if  $LF_{ab} = L_{\kappa}^t$  for  $ab \in \{a, b\}$  and  $\kappa \in \{l, r\}$  then
36:        $LF_{ab} \leftarrow L_{\kappa}$ 
37:     end if
38:     if  $LF_{ab} = L_{se'} \vee LF_{ab} = L_{se}^t \vee LF_{ab} = L_{se'}^t$  for  $ab \in \{a, b\}$  then
39:        $LF_{ab} \leftarrow L_{se}$ 
40:     end if
41:      $LF_a \leftrightarrow LF_b$ 
42:   end if
43: end

```

---

## A.2 $k$ -regular Identification Protocol

Algorithm 2 shows the  $k$ -regular identification protocol in Subsection 3.2.

The roles of the variables at agent  $a$  in Protocol 2 are as follows.

- $LF_a \in \{L_0, L_1, \dots, L_k, \phi, \phi'\}$ : Variable  $LF_a$ , initialized to  $L_0$ , represents states for a leader token and marked agents. If  $LF_a$  is neither  $\phi$  nor  $\phi'$ , agent  $a$  has a leader token. In particular, if  $LF_a = L_i$  ( $i \in \{0, 1, \dots, k\}$ ) holds, agent  $a$  has an  $L_i$  token. Moreover,  $LF_a = L_i$  represents that agent  $a$  has interacted with  $i$  different non-marked agents (i.e., agent  $a$  has at least  $i$  edges). If  $LF_a = \phi$  holds, agent  $a$  has no leader token. If  $LF_a = \phi'$  holds, agent  $a$  has no leader token and  $a$  is marked by other agents.
- $level_a \in \{0, 1, 2, \dots, \lfloor \log P \rfloor\}$ : Variable  $level_a$ , initialized to 0, represents the level of agent  $a$ .
- $reg_a \in \{yes, no\}$ : Variable  $reg_a$ , initialized to *no*, represents the decision of the  $k$ -regular graph. If  $reg_a = yes$  holds for agent  $a$ , then  $\gamma(s_a) = yes$  holds. If  $reg_a = no$  holds, then  $\gamma(s_a) = no$  holds.

The protocol uses  $O(k \log P)$  states because the number of values taken by variable  $LF_a$  is  $k + 2$ , the number of values taken by variable  $level_a$  is  $\lfloor \log P \rfloor + 1$ , and the number of values taken by other variables ( $loc_a$  and  $reg_a$ ) is constant.

## A.3 Star Identification Protocol

Algorithm 3 shows the star identification protocol in Subsection 3.3.

The roles of the variables at agent  $a$  in Protocol 3 are as follows.

- $LF_a \in \{\phi, \phi', l', L_2, L_3, \dots, L_{n-1}\}$ : Variable  $LF_a$ , initialized to  $\phi$ , represents a role of agent  $a$ .  $LF_a = L_i$  means that a central agent  $a$  has marked  $i$  agents (i.e., agent  $a$  has at least  $i$  edges).  $LF_a = l'$  means that  $a$  is a candidate of a central agent and is a marked agent.  $LF_a = \phi$  means that agent  $a$  is a non-marked agent.  $LF_a = \phi'$  means that agent  $a$  is a marked agent. When  $LF_a = x$  holds, we refer to  $a$  as an  $x$ -agent.
- $star_a \in \{yes, no, never\}$ : Variable  $star_a$ , initialized to *no*, represents a decision of a star. If  $star_a = yes$  holds,  $\gamma(s_a) = yes$  holds (i.e.,  $a$  decides that a given graph is a star). If  $star_a = no$  or  $star_a = never$  holds,  $\gamma(s_a) = no$  holds (i.e.,  $a$  decides that a given graph is not a star).  $star_a = never$  means the stronger decision of *no*. If agent  $a$  with  $star_a = never$  interacts with agent  $b$ ,  $star_b$  transitions to *never* regardless of the value of  $star_b$ .

## 13:18 Population Protocols for Graph Class Identification Problems

■ **Algorithm 2** A  $k$ -regular identification protocol.

**Variables at an agent  $a$ :**

$LF_a \in \{L_0, L_1, \dots, L_k, \phi, \phi'\}$ : States for a leader token and marked agents, initialized to  $L_0$ .

$level_a \in \{0, 1, 2, \dots, \lfloor \log P \rfloor\}$ : States for the level of agent  $a$ , initialized to 0.

$loc_a \in \{yes, no\}$ : States representing whether the degree of agent  $a$  is at least  $k$ , initialized to  $no$ .

$reg_a \in \{yes, no\}$ : Decision of the  $k$ -regular graph, initialized to  $no$ .

```

1: when agent  $a$  interacts with agent  $b$  do
  { The behavior when agents have the same level }
2:   if  $level_a = level_b$  then
  { The election of leader tokens }
3:     if  $LF_a = L_x \wedge LF_b = L_y$  ( $x, y \in \{0, 1, 2, \dots, k\}$ ) then
4:        $level_a \leftarrow level_a + 1$ 
5:        $LF_a \leftarrow L_0, LF_b \leftarrow \phi$ 
6:        $reg_a \leftarrow no$ 
7:        $loc_a \leftarrow no$ 
  { Decision and movement of the token }
8:     else if  $LF_a = L_x \wedge LF_b = \phi$  ( $x \in \{0, 1, 2, \dots, k-2\}$ ) then
9:        $LF_a \leftarrow L_{x+1}, LF_b \leftarrow \phi'$ 
10:    else if  $LF_a = L_x \wedge LF_b = \phi'$  ( $x \in \{0, 1, 2, \dots, k\}$ ) then
11:       $LF_a \leftarrow \phi, LF_b \leftarrow L_0$ 
12:       $reg_b \leftarrow reg_a$ 
13:    else if  $LF_a = L_{k-1} \wedge LF_b = \phi$  then
14:       $LF_a \leftarrow L_k, LF_b \leftarrow \phi'$ 
15:    if  $loc_a = no$  then
16:       $reg_a \leftarrow yes$ 
17:       $loc_a \leftarrow yes$ 
18:    end if
  { Reset of  $reg$  of the leader token (the degree of agent  $a$  is at least  $k+1$ ) }
19:    else if  $LF_a = L_k \wedge LF_b = \phi$  then
20:       $LF_a \leftarrow L_0, LF_b \leftarrow \phi'$ 
21:       $reg_a \leftarrow no$ 
22:    end if
  { Reset of  $reg$  of the leader token ( $loc_a$  or  $loc_b$  is  $no$ ) }
23:    if  $loc_a = no \vee loc_b = no$  then
24:       $reg_a \leftarrow no, reg_b \leftarrow no$ 
25:    end if
  { The behavior when agents have different levels }
26:    else if  $level_a > level_b$  then
27:       $level_b \leftarrow level_a$ 
28:       $loc_b \leftarrow no$ 
29:       $LF_b \leftarrow \phi$ 
30:    end if
31: end

```

The protocol is given in Algorithm 3. Algorithm 3 uses  $3n + 3$  states because the number of values taken by variable  $LF_a$  is  $n + 1$  and the number of values taken by variable  $star_a$  is 3.



■ **Algorithm 3** A star identification protocol.

**A variable at an agent  $a$ :**

$LF_a \in \{\phi, \phi', l', L_2, L_3, \dots, L_{n-1}\}$ : States that represent roles of agents, initialized to  $\phi$ .

$L_i$  represents a central agent,  $l'$  represents a candidate of the central agent,  $\phi'$  represents a marked agent, and  $\phi$  represents a non-marked agent.

$star_a \in \{yes, no, never\}$ : Decision of a star, initialized to  $no$ .

```

1: when agent  $a$  interacts with agent  $b$  do
  { The behavior when  $star_a$  or  $star_b$  is  $never$  }
2:   if  $star_a = never \vee star_b = never$  then
3:      $star_a \leftarrow never, star_b \leftarrow never$ 
  { The behaviors when  $star_a \neq never$  and  $star_b \neq never$  holds }
4:   else
  { The election of a central agent }
5:     if  $LF_a = \phi \wedge LF_b = \phi$  then
6:        $LF_a \leftarrow l', LF_b \leftarrow l'$ 
7:     else if  $LF_a = l' \wedge LF_b = \phi$  then
8:        $LF_a \leftarrow L_2, LF_b \leftarrow \phi'$ 
  { Counting the number of adjacent agents by the central agent }
9:     else if  $LF_a = L_i \wedge LF_b = \phi$  ( $2 \leq i \leq n - 2$ ) then
10:       $LF_a \leftarrow L_{i+1}, LF_b \leftarrow \phi'$ 
11:    end if
12:    if  $LF_a = L_{n-1}$  then
13:       $star_a \leftarrow yes, star_b \leftarrow yes$ 
14:    end if
  { Decision of  $never$  }
15:    if  $LF_a = \phi' \wedge LF_b = \phi'$  then
16:       $star_a \leftarrow never, star_b \leftarrow never$ 
17:    else if  $LF_a = \phi' \wedge LF_b = l'$  then
18:       $star_a \leftarrow never, star_b \leftarrow never$ 
19:    end if
  { Conveyance of  $yes$  }
20:    if  $star_a = yes \vee star_b = yes$  then
21:       $star_a \leftarrow yes, star_b \leftarrow yes$ 
22:    end if
23:  end if
24: end

```



# Fast Graphical Population Protocols

Dan Alistarh ✉

IST Austria, Klosterneuburg, Austria

Rati Gelashvili ✉

Novi Research, Menlo Park, CA, USA

Joel Rybicki ✉ 

IST Austria, Klosterneuburg, Austria

---

## Abstract

Let  $G$  be a graph on  $n$  nodes. In the stochastic population protocol model, a collection of  $n$  indistinguishable, resource-limited nodes collectively solve tasks via pairwise interactions. In each interaction, two randomly chosen neighbors first read each other's states, and then update their local states. A rich line of research has established tight upper and lower bounds on the complexity of fundamental tasks, such as majority and leader election, in this model, when  $G$  is a *clique*. Specifically, in the clique, these tasks can be solved *fast*, i.e., in  $n$  polylog  $n$  pairwise interactions, with high probability, using at most polylog  $n$  states per node.

In this work, we consider the more general setting where  $G$  is an arbitrary regular graph, and present a technique for simulating protocols designed for fully-connected networks in any connected regular graph. Our main result is a simulation that is *efficient* on many interesting graph families: roughly, the simulation overhead is polylogarithmic in the number of nodes, and quadratic in the conductance of the graph. As a sample application, we show that, in any regular graph with conductance  $\varphi$ , both leader election and exact majority can be solved in  $\varphi^{-2} \cdot n$  polylog  $n$  pairwise interactions, with high probability, using at most  $\varphi^{-2} \cdot \text{polylog } n$  states per node. This shows that there are fast and space-efficient population protocols for leader election and exact majority on graphs with good expansion properties. We believe our results will prove generally useful, as they allow efficient technology transfer between the well-mixed (clique) case, and the under-explored spatial setting.

**2012 ACM Subject Classification** Theory of computation → Distributed algorithms

**Keywords and phrases** population protocols, leader election, exact majority, graphs

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2021.14

**Related Version** *Full Version*: <https://arxiv.org/abs/2102.08808> [6]

**Funding** *Dan Alistarh*: This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 805223 ScaleML).

*Joel Rybicki*: This project has received from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 840605.

**Acknowledgements** We grateful to Giorgi Nadiradze for pointing out a generalisation of the phase clock construction to non-regular graphs. We also thank anonymous reviewers for their useful comments on earlier versions of this manuscript.

## 1 Introduction

Since the early days of computer science, there has been significant interest in developing an algorithmic theory of molecular and biological systems [49]. In distributed computing, *population protocols* [8] have become a popular model for investigating the collective computational power of large collections of communication- and computationally-bounded agents. This model consists of  $n$  identical agents, seen as finite state machines, and computation



© Dan Alistarh, Rati Gelashvili, and Joel Rybicki;  
licensed under Creative Commons License CC-BY 4.0

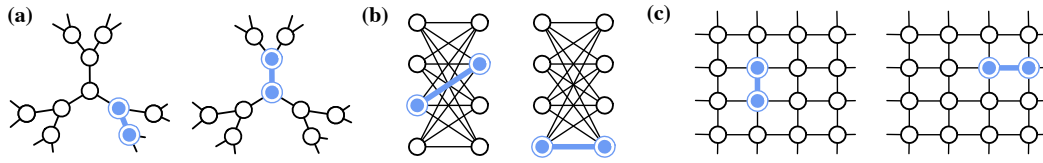
25th International Conference on Principles of Distributed Systems (OPODIS 2021).

Editors: Quentin Bramas, Vincent Gramoli, and Alessia Milani; Article No. 14; pp. 14:1–14:18

Leibniz International Proceedings in Informatics



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



■ **Figure 1** The graphical population protocol model. In each step, a random edge  $\{u, v\}$  is selected and the nodes  $u$  and  $v$  interact (blue nodes). Examples of graph classes covered by our construction: (a) regular high-girth expanders, (b) bipartite complete graphs, (c) toroidal grids.

proceeds via pairwise interactions which trigger local state transitions. The sequence of interactions is provided by a scheduler, which picks pairs of agents to interact. The goal is to have the system reach a configuration satisfying a given predicate, while minimising the number of interactions (time complexity) and the number of states per node (space complexity) required by the protocol.

Early work on population protocols focused on the computational power of the model under various interaction graphs [8, 11]. More recently, the focus has shifted to understanding complexity thresholds, often in the form of fundamental complexity trade-offs between time and space complexity, e.g. [10, 7, 32, 35, 4, 16, 19, 31]; for recent surveys please see [34, 5].

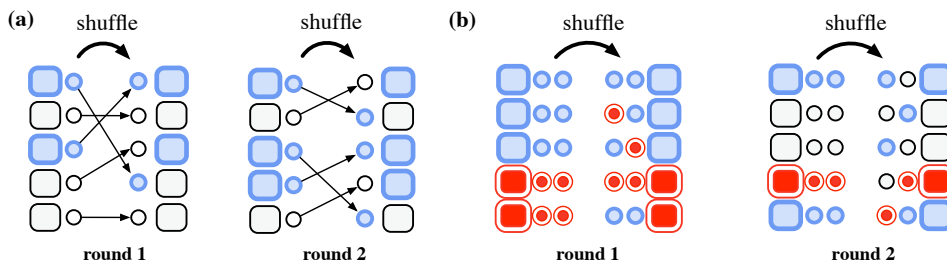
The second line of work almost focuses mainly on the *uniform* stochastic scheduler, where each interaction pair is chosen uniformly at random *among all pairs* of agents in the population, and the time complexity of a protocol is measured by the number of interactions needed to solve a task. This is analogous to having a large *well-mixed* solution of interacting particles when modelling chemical reactions. However, many natural systems exhibit spatial structure and this structure can significantly influence the system dynamics.

Indeed, there is a separation in terms of computational power for population protocols in the clique versus other interaction graphs: connected interaction graphs can simulate adversarial interactions on the clique graph by shuffling the states of the nodes [8] and population protocols on some interaction graphs can compute a strictly larger set of predicates than protocols on the clique; see e.g. [13] for a survey of computability results.

By comparison, surprisingly little is known about the *complexity* of basic tasks in general interaction graphs under the stochastic scheduler. So far, only a handful of protocols have been analysed on general graphs. Existing analyses tend to be complex, and specialised to specific algorithms on limited graph classes [33, 27, 42, 43, 17]. This is natural: given the intricate dependencies which arise due to the underlying graph structure, the design and analysis of protocols in the spatial setting is understood to be challenging.

**Contributions.** In this work, we provide a general approach showing that standard problems in population protocols can be solved *efficiently* under *graphical* stochastic schedulers, by leveraging solutions designed for complete graphs. Our results are as follows:

1. We give a general framework for simulating a large class of *synchronous* protocols designed for *fully-connected networks*, in the graphical stochastic population protocol model (see Figure 1). Thus, the user can design efficient (and simple to analyse) synchronous algorithms on a clique model, and transport the analysis automatically to the population protocol model on a large class of interaction graphs. For instance, on any  $d$ -regular graph with edge expansion  $\beta > 0$ , the resulting overhead in parallel time and state complexity is in the order of  $(d/\beta)^2 \cdot \text{polylog } n$ .



■ **Figure 2** The synchronous  $k$ -token shuffling model with 5 nodes for  $k = 1$  and  $k = 2$ . Rectangles are nodes and the small circles are tokens. In each round, nodes generate  $k$  tokens based on their current state. Then all  $nk$  tokens are shuffled randomly. After this, nodes update their state based on the vector of  $k$  tokens they hold. (a) An execution of a protocol in the 1-token shuffling model. The arrows between tokens represent the random permutation used to shuffle tokens. (b) An execution of a protocol for  $k = 2$ . Each node sends and receives two tokens.

2. As concrete applications, we show that for any  $d$ -regular graph with edge expansion  $\beta > 0$ , there exist protocols for leader election and exact majority that stabilise both in expectation and with high probability in  $(d/\beta)^2 \cdot \text{polylog } n$  parallel time, using  $(d/\beta)^2 \cdot \text{polylog } n$  states.
3. To complement the results following from the simulation, we also show that, on any graph  $G$  with diameter  $\text{diam}(G)$  and  $m$  edges, leader election can be solved both in expectation and with high probability in  $O(\text{diam}(G) \cdot mn^2 \log n)$  parallel time, by analysing the time complexity of the constant-state protocol by Beauquier et al. [15].

**Technical Overview.** Our reduction framework combines several techniques from different areas, and can be distilled down to the following ingredients.

We start by defining a simple *synchronous, fully-connected* model of communication for the  $n$  nodes, called the  *$k$ -token shuffling model*. This is the model in which the algorithm should be designed and analysed, and is similar, and in some ways simpler, relative to the standard population model. Specifically, nodes proceed in *synchronous* rounds, in which every node  $v$  first generates  $k$  tokens based on its current state. Tokens are then shuffled uniformly at random among the nodes. At the end of a round, every node  $v$  updates its local state based on its current state, and the tokens it received in the round. Figure 2 illustrates the model. This simple model is quite powerful, as it can simulate both *pairwise* and *one-way* interactions between all sets of agents, for well-chosen settings of the parameter  $k$ .

Our key technical result is that any algorithm specified in this round-synchronous  $k$ -token shuffling model can be *efficiently* simulated in the graphical population model. Although intuitive, formally proving this result, and in particular obtaining bounds on the efficiency of the simulation, is non-trivial. First, to show that simulating a *single round* of the  $k$ -token shuffling model can be done efficiently, we introduce new type of *card shuffling process* [28, 50, 23, 38], which we call the  $k$ -stack interchange process, and analyse its mixing time by linking it to random walks on the symmetric group.

Second, to allow correct and efficient asynchronous simulation of the synchronous token shuffling model, we introduce two new gadgets: (1) a *graphical* version of *decentralised phase clocks* [4, 36, 35], combined with (2) an *asynchronous* token shuffling protocol, which simulates the  $k$ -token interchange process in a graphical population protocol. The latter ingredient is our main technical result, as it requires both efficiently combining the above components, and carefully bounding the probability bias induced by simulating a synchronous model under asynchronous pairwise-random interactions.

■ **Table 1** Protocols for exact majority (EM) and leader election (LE) for different graph classes. The state complexity is the number of states used by the protocol. The parallel time column gives the expected parallel time (expected number of interaction steps divided by  $n$ ) to stabilise. (\*) In [33], the running time of the protocol is bounded by the initial discrepancy in the inputs and the spectral properties of the contact rate matrix; bounds in terms of  $n$  are only given for select graph classes (paths, cycles, stars, random graphs and cliques). No sublinear in  $n$  bounds on parallel time are given in [33]. Protocols marked with (★) stabilise also in non-regular graphs in  $\text{poly}(n)$  time.

Graph class	Task	States	Parallel time	Note
cliques	EM	4	$O(n \log n)$	[33]
	EM	$O(\log n)$	$\Theta(\log n)$	[31]
	LE	2	$\Theta(n)$	[32]
	LE	$\Theta(\log \log n)$	$\Theta(\log n)$	[19]
connected	EM	4	$\text{poly}(n)$	[33, 17], (*)
	LE	6	$O(\text{diam}(G) \cdot mn^2 \log n)$	<b>new</b> analysis of [15]
$d$ -regular	EM	$(d/\beta)^2 \cdot \text{polylog } n$	$(d/\beta)^2 \cdot \text{polylog } n$	<b>new</b> , (★)
	LE	$(d/\beta)^2 \cdot \text{polylog } n$	$(d/\beta)^2 \cdot \text{polylog } n$	<b>new</b> , (★)

Finally, we instantiate this framework to solve exact majority and leader election in the graphical setting. We provide simple token-shuffling protocols for these problems, as well as backup protocols to ensure their correctness in all executions.

**Implications.** Our results imply new and improved upper bounds on the time and state complexity of majority and leader election for a wide range of graph families. In some cases, they improve upon the best known upper bounds for these problems. Please see Table 1 for a systematic comparison. Specifically, our results show that:

- In *sparse* graphs with good expansion properties, such as constant-degree graphs with constant edge expansion (Figure 1a), our simulation has polylogarithmic time and state complexity overhead, relative to clique-based algorithms. Thus, good expanders admit fast protocols using polylogarithmic states, despite being sparser than the clique.
- In *dense* graphs, we obtain similar bounds whenever  $d/\beta \in \text{polylog } n$  holds. This is the case for instance in  $d$ -dimensional hypercubes with  $n = 2^d$  nodes, but also in highly-dense clique-like graphs, such as regular complete multipartite graphs (Figure 1b), where the degree and expansion are both  $\Theta(n)$ .
- In  $D$ -dimensional toroidal grids, we get algorithms with  $n^{2/D}$   $\text{polylog } n$  parallel time and state complexity. These graphs include cycles (1-dimensional toroidal grids), two-dimensional grids (Figure 1c), three-dimensional lattices, and so on.

While our protocols guarantee *fast* stabilisation in regular graphs with high expansion, they will stabilise in polynomial expected time in *any connected graph*. The results can be carried over to certain classes of *non-regular* graphs provided that they are not highly irregular and have high expansion; we discuss this in Section 8, and provide examples in the Appendix.

It is known that, in the clique setting, constant-state protocols are necessarily slower than protocols with super-constant states [32, 4]. Our results suggest the existence of a similar complexity gap in the graphical setting. Specifically, on  $d$ -regular graphs with good expansion, such that  $d/\beta \in \text{polylog } n$ , we provide polylogarithmic-time protocols for both leader election and exact majority. This opens a significant complexity gap relative to known constant-state protocols on graphs. For instance, the 4-state exact majority protocol for general graphs [33] requires  $\Omega(n)$  parallel time even in regular graphs with high expansion, if

node degrees are  $\Theta(n)$ . (A simple example is the complete bipartite graph given in Figure 1b.) Yet, our protocols guarantee stabilisation in only polylog  $n$  parallel time in both low and high degree graphs, as long as  $d/\beta$  is at most polylog  $n$ .

Due to space constraints, in the following we focus on the simulation framework and its properties; the correctness proofs for our framework and the algorithmic applications are given in the full version of the paper [6].

## 2 Related Work

**Computability for Graphical Population Protocols.** A variant of the graphical setting was already considered in the foundational work of Angluin et al. [8], which also uses a state shuffling approach. However, the resulting line of work focused on *computational power* in the case where the number of states per node is constant [8, 9, 12, 11, 21]. A key difference is that we aim to simulate pairwise interactions under the uniform stochastic scheduler, as fast protocols in the clique require that pairwise interactions are uniformly random [34]. Thus, one of the main technical challenges is to devise an *efficient* shuffling procedure that guarantees that the simulated interactions are (almost) uniform.

In addition, *self-stabilising* population protocols on graphs have been investigated particularly in the context of leader election [12, 15, 51, 47, 24, 25, 39, 48]. This considers more stringent *transient fault* models than ours: we will thus be able to obtain better bounds, but our results will not directly transfer to self-stabilising protocols. This is natural, since self-stabilising leader election is not solvable on all graph families [12].

Beauquier et al. [15] noted that without the requirement of self-stabilisation, leader election can be solved on every connected graph by a constant-state protocol. We provide the first running-time upper bounds for this protocol; please see Table 1 for a summary of the known bounds. Concurrent work by Sudo et al. [48] on self-stabilising leader election on general graphs uses a similar approach to our analysis of the leader election protocol, presented in the full version [6].

**Complexity in the Clique Model.** A parallel line of work has focused on determining the fundamental space-time trade-offs for key tasks, such as majority and leader election, when the interaction graph is a *clique* [32, 33, 42, 3, 4, 20, 16, 19, 31]. In this case, tight or almost-tight complexity trade-offs are now known for these problems [19, 37, 4].

The vast majority of the work on complexity has focused on the clique case [34, 5]. Two natural justifications for this choice are that: (1) the clique is a good approximation for well-mixed solutions, and (2) the analysis of population protocols can be difficult enough even without additional complications due to graph structure. Bounds on non-complete graphs have been studied for exact [33] and approximate majority [42, 43], with some recent work considering *plurality consensus* [26, 27, 17] in a related model. The recent survey of [34] points out that running time on general graphs is poorly understood, and sets this as an open question. We take a first step towards addressing this gap.

**Interacting Particle Systems.** Another related line of work investigated dynamics of interacting particle systems on graphs, e.g. [2]. However, in this context dynamics are often assumed to be round-synchronous, which allows the use of more powerful techniques, related to independent random walks on graphs. Cooper et al. [26] analysed the coalescence time of independent random walks on a graph in terms of the expansion properties of the graph, where each node initially holds a unique particle, and in each step particles randomly move to

another node. Whenever, two particles meet, they coalesce into a single one, which continues its walk. We also employ token-based protocols on graphs, but in our case tokens are shuffled between nodes instead of coalescing.

Token-based processes have also been used to implement efficient, randomised rumour spreading protocols. For example, Berenbrink et al. [18] analysed the cover time of a synchronous coalescing-branching random walk on regular graphs. Similarly to our work, they use conductance to bound the behaviour of this process in regular graphs. In this work, we use token-based population protocols on graphs, where the tokens are shuffled between nodes during an interaction and the tokens instead of coalescing, may also interact in other ways.

**Plurality Consensus on Expanders.** In plurality consensus, there are  $k > 1$  opinions and the task is to agree on the opinion supported by the most nodes. Berenbrink et al. [17] present a protocol for the plurality consensus problem in a synchronous pull-based interaction model. Their protocol also circulates tokens, and samples their count periodically (after mixing) to estimate opinion counts, running into the issue that the token movements are correlated. The authors provide a generalisation of a result by Sauerwald and Sun [45] in order to show that the joint token distribution is negatively correlated, and therefore the token counting mechanism concentrates.

In this work, we also employ a token exchange protocol, and encounter non-trivial correlation issues. However, we resolve these issues differently: we characterise the distribution of the token interactions using the  $k$ -stack interchange process, and bound its total variation distance relative to the uniform distribution, showing that the two distributions are indistinguishable in polynomial time with high probability. More generally, the goal of our construction is different, as we aim to provide a general framework to efficiently simulate pairwise random node interactions.

**Shuffling Processes.** Our results also connect to the work on card shuffling processes, and in particular, the interchange process, which has a long and rich history, e.g. [29, 1, 28, 50, 38]. While many of these processes are simple to describe, they are often surprisingly challenging to analyse. In the classic interchange process, a card is placed on every node of a graph and the shuffling is performed by randomly exchanging cards between adjacent nodes.

Diaconis and Shahshahani [29] gave sharp bounds on the mixing time of the random transpositions shuffle, i.e., interchange process on the clique. Diaconis and Saloff-Coste [28] developed a powerful comparison technique for upper bounding the mixing time of a random walk on a finite group. This is one of the key techniques for upper bounding mixing times of the interchange process.

Later, Wilson [50] developed a general technique for proving lower bounds for many shuffling processes. For example, he showed that the mixing time of the interchange process on the two-dimensional  $\sqrt{n} \times \sqrt{n}$  grid is  $\Theta(n^2 \log n)$  and  $\Omega(n \log^2 n)$  on the hypercube. Subsequently, Jonasson [38] gave additional upper and lower bounds on the interchange process on various graphs.

### 3 Preliminaries

**Graphs.** A graph  $G = (V, E)$  is  $d$ -regular if every node  $v \in V$  is adjacent to exactly  $d$  other nodes. The edge boundary of a set  $S \subseteq V$  is the set  $\partial S \subseteq E$  of edges with exactly one endpoint in  $S$ . The *edge expansion* of the graph  $G$  is defined as  $\beta = \min \{|\partial S|/|S| : S \subseteq V, |S| \leq n/2\}$ . If  $G$  is regular, its *conductance* is  $\beta/d$ . Unless otherwise mentioned, all graphs are assumed to be regular and connected.



**Probability distributions.** Let  $E$  be a finite set. We say  $\mu: E \rightarrow [0, 1]$  is a probability distribution on  $E$  if  $\sum_{x \in E} \mu(x) = 1$  holds. For  $A \subseteq E$  we write  $\mu(A) = \sum_{x \in A} \mu(x)$ . The *uniform distribution* on  $E$  is the distribution  $\nu$  defined by  $\nu(x) = 1/|E|$ . The *support* of  $\mu$  is the set  $\{x : \mu(x) > 0\}$ . The *total variation distance* between distributions  $\mu_1$  and  $\mu_2$  on  $E$  is

$$\|\mu_1 - \mu_2\|_{\text{TV}} = \frac{1}{2} \sum_{x \in E} |\mu_1(x) - \mu_2(x)| = \max_{A \subseteq E} |\mu_1(A) - \mu_2(A)|.$$

We say that  $\mu$  is  $\varepsilon$ -uniform on  $E$  if  $\|\mu - \nu\|_{\text{TV}} \leq \varepsilon$ .

**Permutations and the symmetric group.** Let  $N > 0$  be a positive integer and  $[N] = \{0, \dots, N-1\}$ . A permutation on  $[N]$  is a bijection from  $[N]$  to  $[N]$ . The symmetric group  $S_N$  over  $[N]$  is the group consisting of the set of all permutations on  $[N]$  with function composition as the group operation and identity element  $\text{id}$  defined by  $\text{id}(i) = i$ . The inverse  $x^{-1}$  of an element  $x \in S_N$  is the map satisfying  $x^{-1} \cdot x = x \cdot x^{-1} = \text{id}$ . A *transposition*  $(i \ j) \in S_N$  of  $i$  and  $j$  is the permutation that swaps the elements  $i$  and  $j$ , but leaves other elements in place. We say that a set  $H \subseteq S_N$  generates  $S_N$  if every element of  $S_N$  can be expressed as a finite product of elements in  $H$  and their inverses. We use  $\cdot$  and  $\circ$  interchangeably to denote function composition.

Let  $\mu$  be a symmetric probability distribution on  $S_N$ , i.e.,  $\mu(x) = \mu(x^{-1})$ . The *random walk on  $S_N$*  with increment distribution  $\mu$  is a discrete time Markov chain with state space  $S_N$ . In each step, a random element  $x$  is sampled according  $\mu$  and the chain moves from state  $y$  to state  $xy$ . Thus, the probability of transitioning from state  $x$  to state  $yx$  is  $\mu(y)$ . The holding probability of the random walk is  $\alpha = \mu(\text{id})$ . The following remark summarises some useful properties of such random walks; see e.g. [41] for proofs.

► **Remark 1.** Let  $\mu$  be an increment distribution for a random walk on  $S_N$ .

1. The uniform distribution  $\nu$  on  $S_N$  is a stationary distribution for the random walk.
2. The random walk is reversible if and only if  $\mu$  is symmetric.
3. The random walk is irreducible if and only if the support of  $\mu$  generates  $S_N$ .
4. If  $\mu(\text{id}) > 0$ , then the random walk is aperiodic.

**Mixing times.** Let  $\nu$  be the uniform distribution on  $S_N$  and be  $p^{(t)}$  be the probability distribution over states of the chain after  $t$  steps. Following [28], we define the  $\ell^s$ -norm and the normalised  $\ell^s$ -distance to stationarity for  $s > 0$  as:

$$\|\mu\|_s = \left( \sum_x |\mu(x)|^s \right)^{1/s} \quad \text{and} \quad d_s(t) = |S_N|^{1-1/s} \cdot \|p^{(t)} - \nu\|_s.$$

The total variation distance and the normalised distances satisfy  $2\|p^{(t)} - \nu\|_{\text{TV}} = d_1(t) \leq d_2(t)$ , where the latter inequality follows from the Cauchy-Schwarz inequality. We define the  $\varepsilon$ -mixing time as  $\tau(\varepsilon) = \min\{t : d_1(t) \leq 2\varepsilon\}$ . We refer to the value  $\tau_{\text{mix}} = \tau(1/2)$  as the *mixing time* of the walk. Note that  $\tau(\varepsilon) \leq \lceil \log_2 \varepsilon^{-1} \rceil \cdot \tau_{\text{mix}}$ .

**Tasks.** Let  $\Sigma$  and  $\Gamma$  be nonempty finite sets of input and output labels, respectively. A task  $\Pi$  on a set  $V$  of  $n$  nodes is a function  $\Pi$  that maps any input labelling  $z: V \rightarrow \Sigma$  to a set  $\Pi(z) \subseteq \Gamma^V$  of feasible output labellings. If  $\Pi(z) = \emptyset$ , then we say that  $z$  is an infeasible input. We focus on two tasks:

- In leader election, the input is the constant function  $z(v) = 1$  and the output labelling  $z'$  is feasible iff there exists  $v \in V$  such that  $z'(v) = 1$  and  $z'(u) = 0$  for all  $u \neq v$ . That is, exactly one node should output 1 and all others should output 0.
- In the majority task, the inputs are given by  $z: V \rightarrow \{0, 1\}$  and  $z' \in \Pi(z)$  if  $z'(v) = b$ , where  $b$  is the input value held by the majority of the nodes. As conventional, the input with equally many zeros and ones is taken to be infeasible.

**Graphical stochastic population protocols.** Let  $G = (V, E)$  be a graph. In the graphical stochastic population model, the computation proceeds *asynchronously*, where in each time step  $t > 0$ :

1. a stochastic scheduler picks uniformly at random a pair  $e_t = (u, v)$  of neighbouring nodes,
2. the nodes  $u$  and  $v$  read each other's states and update their local states.

As is common in population protocols, we assume that the node pairs are *ordered*, which will allow us to distinguish the two nodes: node  $u$  is called the *initiator* and  $v$  is the *responder*. We assume that nodes have access to independent and uniform random bits. Specifically, upon each interaction, both  $u$  and  $v$  are provided with a single random bit each. We note that this assumption is common in the context of population protocols, e.g. [35], and can be justified practically by the fact that chemical reaction network (CRN) implementations can directly obtain random bits given the structure of their interactions [22].

Formally, a protocol for a task  $\Pi$  is a tuple  $\mathbf{A} = (f, \ell_{\text{in}}, \ell_{\text{out}})$ , where  $f: S \times \{0, 1\} \times S \times \{0, 1\} \rightarrow S \times S$  is the state transition function and  $S$  is the set of states,  $\ell_{\text{in}}: \Sigma \rightarrow S$  maps inputs to initial states, and  $\ell_{\text{out}}: S \rightarrow \Gamma$  maps states to outputs. A configuration is a map  $x: V \rightarrow S$  and  $x_0 = \ell_{\text{in}} \circ z$  is the initial configuration on input  $z$ . An asynchronous schedule is a random sequence  $(e_t)_{t \geq 1}$  of the interaction pairs. An execution is the sequence  $(x_t)_{t \geq 0}$  of configurations given by

$$x_{t+1}(u), x_{t+1}(v) = f(x_t(u), q_{t+1}(u), x_t(v), q_{t+1}(v)) \text{ and } x_{t+1}(w) = x_t(w) \text{ for } w \in V \setminus \{u, v\},$$

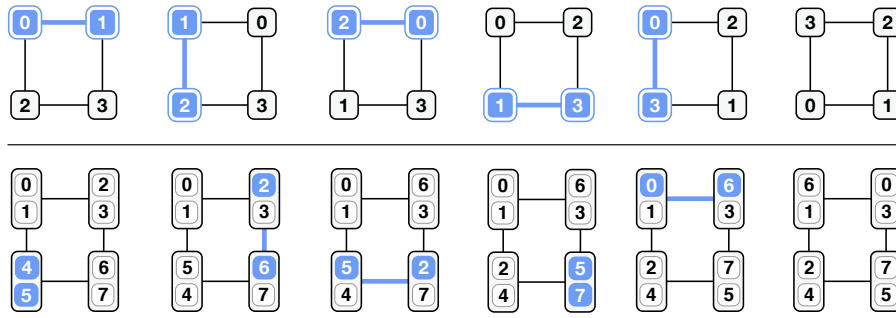
where  $(u, v) = e_{t+1}$  and  $q_{t+1}(u) \in \{0, 1\}$  is the random bit provided to the node  $u$  during the interaction. The output of the protocol at step  $t$  is given by  $z'_t = \ell_{\text{out}} \circ x_t$ .

We say that  $\mathbf{A}$  stabilises on input  $z$  by step  $T$  if  $z'_{t+1} = z'_t$  and  $z'_t \in \Pi(z)$  holds for all  $t \geq T$ . Moreover,  $\mathbf{A}$  solves the task  $\Pi$  with probability at least  $p$  in  $T(\mathbf{A})$  steps if the protocol stabilises by step  $T(\mathbf{A})$  on any feasible input with probability at least  $p$ . The state complexity of the protocol is  $S(\mathbf{A}) = |S|$ , i.e., the number of states used by the protocol.

**Synchronous token protocols.** In the synchronous  $k$ -token shuffling model, we assume that there are  $n$  agents which communicate in a round-based fashion using *tokens*. In each round,

1. every node  $v$  generates exactly  $k$  tokens based on its current state,
2. all  $nk$  tokens are shuffled uniformly at random so that each node gets exactly  $k$  tokens,
3. every node  $v$  updates its local state based on its current state and the  $k$  tokens it received.

Let  $X$  be the set of states a node can take and  $Y$  be a set of distinct token types. An algorithm in the token shuffling model is a tuple  $\mathbf{B} = (f, g, \ell_{\text{in}}, \ell_{\text{out}})$ . The map  $f: X \times Y^k \rightarrow X$  is a state transition function, and  $g: X \rightarrow Y^k$  determines which tokens each node creates at the start of each round. As before,  $\ell_{\text{in}}: \Sigma \rightarrow X$  maps input values to initial states and  $\ell_{\text{out}}: X \rightarrow \Gamma$  maps the state of a node onto an output value. The initial configuration on input  $z$  is  $x_0 = \ell_{\text{out}} \circ z$ .



**Figure 3** Interchange dynamics on a 4-cycle. In each step, blue cards are swapped. Top row: The 1-stack interchange process. Bottom row: The 2-stack interchange process. In each step, a randomly selected node either moves its top card to the bottom of its stack or exchanges it with the top card of a randomly selected neighbour.

A *synchronous schedule* is a sequence  $(\sigma_r)_{r \geq 1}$ , where the permutation  $\sigma_r \in S_{nk}$  describes how the tokens are shuffled in round  $r$ . For any  $y: [nk] \rightarrow Y$ , we let  $y(v_0, \dots, v_{k-1}) = (y(v_0), \dots, y(v_{k-1}))$ . A synchronous execution induced by  $(\sigma_r)_{r \geq 1}$  on input  $z$  is defined by

$$y_{r+1}(v_0, \dots, v_{k-1}) = (g \circ x_r)(v) \quad \text{and} \quad x_{r+1}(v) = f(x_r(v), (y_{r+1} \circ \sigma_{r+1})(v_0, \dots, v_{k-1})),$$

where  $y_r(v_0, \dots, v_{k-1})$  and  $(y_r \circ \sigma_{r+1})(v_0, \dots, v_{k-1})$ , respectively, are the  $k$  tokens generated and received by node  $v$  during round  $r$ .

We assume the uniform synchronous scheduler, which picks each permutation  $\sigma_r$  independently and uniformly at random from the set of all permutations  $S_{nk}$ . The output of node  $v$  at the end of round  $r$  is  $z'_r(v) = (\ell_{\text{out}} \circ x_r)(v)$ . The synchronous algorithm **B** stabilises on input  $z$  in  $R$  rounds if  $z_{r+1} = z'_r$  and  $z'_r \in \Pi(z)$  holds for all  $r \geq R$ . The algorithm solves the problem  $\Pi$  if it stabilises in  $R$  rounds on any feasible input with probability at least  $p$ .

#### 4 Shuffling states on graphs: the $k$ -stack interchange process

We now describe a shuffling process on graphs, which we call the  *$k$ -stack interchange process*. This process will be useful in our analysis, and is a variant of the classic graph interchange process, e.g. [30, 38]. We analyse its mixing time using the path comparison method of Diaconis and Saloff-Coste [28], leveraging a classical flow result of Leighton and Rao [40].

**The  $k$ -stack interchange process.** Let  $G = (V, E)$  a graph with  $n$  vertices  $\{0, \dots, n-1\}$  and  $N = kn$  for  $k > 0$ . Assume each node of  $G$  holds a stack of exactly  $k$  cards, and consider the shuffling process where, in every time step, one of the following actions is taken:

1. with probability  $1/2$ , move the top card of a random node to the bottom of its stack,
2. with probability  $1/4$ , choose a random edge  $\{u, v\}$  and swap the top cards of  $u$  and  $v$ ,
3. with probability  $1/4$ , do nothing.

We refer to this process as the  *$k$ -stack interchange process on  $G$* . The special case of  $k = 1$  is the classic interchange process on  $G$  with holding probability  $3/4$ , as the first rule does not do anything on stacks of size 1. For  $k > 1$ , the holding probability will be  $1/4$ . Instances of the process for  $k = 1$  and  $k = 2$  are illustrated in Figure 3.

► **Theorem 2.** *Let  $G$  be a  $d$ -regular graph with edge expansion  $\beta > 0$ . For any constant  $k > 0$ , the mixing time of the  $k$ -stack interchange process on  $G$  is  $O\left(\left(\frac{d}{\beta}\right)^2 n \log^3 n\right)$ .*

We prove this theorem in the full version of the paper [6]. In Section 6, we will show that this shuffling process can be implemented efficiently in the graphical population protocol model.

## 5 Decentralised *graphical* phase clocks

We now describe a *bounded phase clock* construction for the stochastic population protocol model over regular graphs. Interestingly, the construction can be generalised to *non-regular* graphs, assuming that node degrees do not deviate too much from the average degree; see the full version [6] for details. Our approach generalises that of Alistarh et al. [4], who built a leaderless phase clock on cliques leveraging the classic two-choice load balancing process [14, 44].

**Phase clocks.** Let  $\phi > 0$  be an integer and consider a population protocol  $\mathbf{C}$  with state variables  $c(v) \in \{0, \dots, \phi - 1\}$  for each  $v \in V$ . The variable  $c(v)$  represents the value of the clock at node  $v$ . Let  $c(v, t)$  be the clock value node  $v$  has at the end of time step  $t$  (regardless of whether it was active during that step). We define the distance  $D$  between two clock values and the skew  $\Delta$  of the clock at the end of step  $t$ , respectively, as follows:

$$D(x, y) = \min\{|x - y|, \phi - |x - y|\} \quad \text{and} \quad \Delta(t) = \max_{u, v \in V} D(c(u, t), c(v, t)).$$

We say that the protocol  $\mathbf{C}$  implements a  $(\phi, \gamma, \kappa)$ -clock if for all  $t \geq 0$  the following hold:

1.  $\Pr[\Delta(t) \geq \gamma] < t/n^\kappa$ , and
2.  $c(v, t + 1) = c(v, t) + 1 \bmod \phi$  for exactly one  $v \in V$  and  $c(u, t + 1) = c(u, t)$  for all  $u \in V \setminus \{v\}$ .

Intuitively,  $\phi$  is the length of a phase,  $\gamma$  is the skew of the clock, and  $\kappa$  controls the failure probability. The above properties guarantee that the clocks (1) have a skew bounded by  $\gamma$  for polynomially many steps, w.h.p.; and (2) in each step, the clocks make progress (at some node). A clock protocol  $\mathbf{C}$  *fails* at time  $t$  if  $\Delta(t) \geq \gamma$  occurs. Several types of phase clocks have been proposed in the population protocol literature, e.g. [10, 35, 4, 46].

**Bounded phase clocks via graphical load balancing.** Let  $G$  be a graph and suppose that each node of  $G$  contains a bin, which is initially empty. Our phase clock is based on the classic graphical load-balancing process [44] where, in each step, a directed edge  $(u, v)$  is sampled uniformly at random and a ball is placed into the *least* loaded of bin among the two nodes connected by the edge (in case of ties, place the ball into bin  $u$ ). Using this idea, we obtain *bounded* phase clocks in the graphical population protocol model. We note that this is the only place in our framework where the initiator/responder distinction is used. The proof of this result can be found in the full version [6].

► **Theorem 3.** *Let  $G = (V, E)$  be a  $d$ -regular graph with  $n$  nodes and edge expansion  $\beta > 0$  and let  $\kappa > 1$  be a constant. There exists a constant  $c(\kappa)$  such that for any  $\gamma$  and  $\phi$  satisfying*

$$\gamma \geq c(\kappa) \frac{d}{\beta} \log n \quad \text{and} \quad \phi \geq 2\gamma$$

*there exists  $(\phi, \gamma, \kappa)$ -clock on  $G$  that uses  $\phi$  states per node.*

## 6 Simulating synchronous token shuffling protocols

In this section, we give our main technical result: synchronous protocols in the fully-connected token shuffling model can be simulated in the graphical, stochastic population protocol model.

► **Theorem 4.** *Let  $k > 0$  be a constant and  $\mathbf{A}$  be a synchronous  $k$ -token shuffling protocol on  $n$  nodes, where  $X$  is the set of local states and  $Y$  the set of token types used the protocol  $\mathbf{A}$ . If  $\mathbf{A}$  solves the task  $\Pi$  with high probability in  $R \in \text{poly}(n)$  rounds, then there exists a stochastic population protocol  $\mathbf{B}$  that also solves task  $\Pi$  with high probability on any  $n$ -node  $d$ -regular graph  $G$  with edge expansion  $\beta > 0$ . The step complexity  $T(\mathbf{B})$  and state complexity  $S(\mathbf{B})$  of the protocol  $\mathbf{B}$  satisfy*

$$T(\mathbf{B}) \in O(R \cdot n \cdot \zeta) \quad \text{and} \quad S(\mathbf{B}) \in O(|X| \cdot |Y|^k \cdot \zeta) \quad \text{with} \quad \zeta = \log n \cdot \left( \frac{d}{\beta} + \frac{\tau_{\text{mix}}}{n} \right),$$

where  $\tau_{\text{mix}}$  is the mixing time of the  $k$ -stack interchange process on  $G$ .

**Notation.** The rest of this section is dedicated to proving this theorem. Throughout, we fix  $R = R(n) \in \text{poly}(n)$  and  $\varepsilon = 1/n^a < 1/(Rn^\lambda)$  for an arbitrary large constant  $a > 0$ . Let  $G = (V, E)$  be  $d$ -regular  $n$ -node graph and  $N = kn$ . We use  $\mu$  to denote the increment distribution of the  $k$ -stack interchange process on the graph  $G$ . The support of  $\mu$  is the set  $H \subseteq S_N$  and  $\tau = \tau(\varepsilon)$  is the  $\varepsilon$ -mixing time of the  $k$ -stack interchange process.

### 6.1 The token shuffling protocol

We now give a stochastic population protocol that simulates uniform schedules of the synchronous token shuffling model. The protocol simulates the random walk made by the  $k$ -stack interchange process, synchronised by phase clocks.

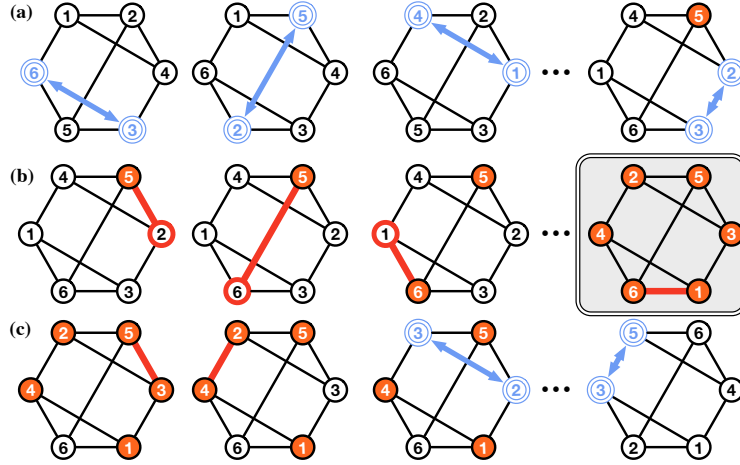
**Setting up the clock.** We choose the parameter  $\kappa > 0$  such that a  $(\phi, \gamma, \kappa)$ -clock  $\mathbf{C}$  with parameters given by

$$\gamma \in \Theta\left(\frac{d}{\beta} \log n\right) \quad \phi = \gamma + \vartheta \quad \vartheta = \frac{2\tau}{n} + 3\gamma \quad t^* = (R\phi + \gamma)n$$

fails (i.e., the clock skew becomes  $\gamma$  or greater) with probability at most  $1/n^\lambda$  during the first  $t^*$  steps. Since  $\phi \geq 2\gamma$ ,  $R \in \text{poly}(n)$ , and  $t^* \in \text{poly}(n)$  hold, such a protocol exists by Theorem 3 for any constant  $\lambda > 0$  by choosing a sufficiently large  $\kappa$ . The fact that  $t^*$  is polynomially bounded follows from Theorem 2 and that  $\beta \geq 1/n^2$  for any regular connected graph. Further,  $\tau \leq \lceil \log 1/\varepsilon \rceil \cdot \tau_{\text{mix}} \in \text{poly}(n)$ , and hence,  $\phi, \gamma \in \text{poly}(n)$ .

**The token shuffling protocol.** The parameter  $\vartheta$  is used as a special threshold value for the token shuffling protocol. We assume that each node  $v$  holds exactly  $k$  tokens, which are ordered from 0 to  $k - 1$ , in the same manner as cards ordered are in the  $k$ -stack interchange process. We say that the first token is the *top token*. We say that node  $u$  is *receptive* whenever its clock satisfies  $c(u) < \vartheta$  and that it is *suspended* otherwise. When nodes in  $\{u, v\}$  interact, they apply the following rule:

1. If both are receptive, that is,  $c(u) < \vartheta$  and  $c(v) < \vartheta$  holds, then
  - a. Let  $q(u)$  and  $q(v)$  be the random coin flips of  $u$  and  $v$ , respectively.
  - b. If  $q(u) = q(v) = 0$ , then  $u$  and  $v$  swap their top tokens.
  - c. If  $q(u) < q(v)$ , then  $v$  moves its top token to the bottom of its stack;  $u$  does nothing.
  - d. If  $q(u) = q(v) = 1$ , then do nothing.
2. Otherwise, do nothing.



■ **Figure 4** The dynamics of the shuffling protocol for  $k = 1$ . Circles filled with white and red denote receptive and suspended nodes, respectively. The blue arrows connect nodes who exchange their tokens in the given step. Red lines denote steps, where at least one of the interacting nodes is suspended, and thus, no swap is made. (a) Initially all nodes are receptive and swap tokens with their interaction partners. After sufficiently many interactions, nodes become suspended and refrain from swapping tokens. (b) Eventually all nodes are suspended. The highlighted panel shows the resulting permutation, which will act as the interaction pattern for the simulated round. (c) As the phase clocks reset back to 0, nodes become receptive again, and the tokens are shuffled once more.

The protocol uses at most one random bit per node per interaction and that this is the only part of our framework, where the random bits provided to the nodes are used. The interacting nodes exchange at most 4 bits (i.e., whether they receptive or not, and the result of their coin flip) in addition to the contents of the swapped tokens in Step (1b). Finally, observe that when all nodes are receptive, the tokens are shuffled according to the increment distribution  $\mu$  of the  $k$ -stack interchange process on  $G$ . Figure 4 illustrates the dynamics of the shuffling protocol in the case  $k = 1$ .

## 6.2 Properties of the shuffling protocol

We now analyse the above shuffling protocol. Let  $c(u, t)$  indicate the clock value of node  $u$  at the end of step  $t$ . Let  $c(u, 0) = 0$  and  $t(v, 0) = 0$ . We say that the clock of node  $u$  resets at time step  $t$  if its value transitions from  $\phi - 1$  to 0. For  $r \geq 0$ , define

- $t(v, r+1) = \min\{t > t(v, r) : c(v, t) = 0\}$ ; the step when  $v$  resets its clock for the  $r$ th time,
  - $t_{\min}(r) = \min\{t(v, r) : v \in V\}$ ; the *earliest* step when some clock is reset for the  $r$ th time,
  - $t_{\max}(r) = \max\{t(v, r) : v \in V\}$ ; the *latest* step when some clock is reset for the  $r$ th time.
- Similarly, we define the times with respect to the events when the clocks reach the value  $\vartheta$ :
- $s(v, r) = \min\{t > t(v, r) : c(v, t) = \vartheta\}$ ,
  - $s_{\min}(r) = \min\{s(v, r) : v \in V\}$ ,
  - $s_{\max}(r) = \max\{s(v, r) : v \in V\}$ .

The following lemma captures the relationship between the timing of these events.

► **Lemma 5.** *With high probability, the following inequalities hold:*

1.  $t_{\max}(R+1) \leq t^* = (R\phi + \gamma)n$ ,
2.  $s_{\min}(r) - t_{\max}(r) \geq \tau$  for each  $1 \leq r \leq R$ .
3.  $t_{\max}(r) < s_{\max}(r) < t_{\min}(r+1)$  for each  $1 \leq r \leq R$ .

**Distribution of tokens.** We now show that the distribution tokens mix to an  $\varepsilon$ -uniform distribution during the intervals  $\{t_{\max}(r)+1, \dots, s_{\min}(r)\}$  for  $1 \leq r \leq R$ . Let  $\pi_0 = \text{id}$  and  $\pi_t$  denote the locations of the tokens after  $t$  steps of the shuffling protocol. Define  $\sigma_0 = \text{id}$  and

$$\sigma_r = \pi_{s_{\max}(r)} \text{ for } 1 \leq r \leq R.$$

Observe that  $\sigma_r = \rho_3 \cdot \rho_2 \cdot \rho_1 \cdot \sigma_{r-1}$ , where each  $\rho_i$  is product of elements from the support  $H \subseteq S_N$  of the increment distribution  $\mu$  of the  $k$ -stack interchange process, where

- $\rho_1 = x_{t_{\max}(r)} \cdots x_{t_{\min}(r-1)+1}$  (a subset of nodes have become receptive for the  $r$ th time),
- $\rho_2 = x_{s_{\min}(r)} \cdots x_{t_{\max}(r)+1}$  (all nodes are receptive),
- $\rho_3 = x_{s_{\max}(r)} \cdots x_{s_{\min}(r)+1}$  (a subset of nodes have become suspended for the  $r$ th time).

(Recall that permutations are applied from right to left.) Observe that while each  $x_i$  is a random element of  $H$ , only the elements  $\rho_2$  are guaranteed to be distributed according to the increment distribution  $\mu$  of the  $k$ -stack interchange process. The elements of  $\rho_1$  and  $\rho_3$  are skewed towards the identity permutation, as some nodes are suspended whenever their clock values are in  $\{\vartheta, \dots, \phi - 1\}$ . The next lemma establishes that this does not interfere with the mixing behaviour.

► **Lemma 6.** *Let  $0 \leq r < R$ . For any  $A \subseteq S_N$ , we have  $|\Pr[\sigma_{r+1} \in A \mid \sigma_r] - \nu(A)| \leq \varepsilon$ .*

### 6.3 The simulation protocol

Using the shuffling protocol in the population protocol model, we can simulate an  $R$ -round algorithm **A** in the synchronous  $k$ -token shuffling model. Let  $f: X \times Y^k \rightarrow X$  be the state transition function and  $g: X \rightarrow Y^k$  be the token generation function of the algorithm **A**. Recall that  $X$  and  $Y$  denote the sets of local states and token types, respectively.

**The simulation protocol.** Each node  $v$  maintains the following variables:

- $a(v) \in X$  to simulate the local state of the synchronous protocol **A**,
- $b_0(v), \dots, b_{k-1}(v) \in Y$  to store the sent and received tokens, and
- $r(v) \in \{0, 1, \dots, R\}$  to store the number of simulated rounds.

The variable  $a(v)$  is initialised to the initial state  $x_0(v)$  of node  $v$  in the algorithm **A** and  $b_0(v), \dots, b_{k-1}(v)$  are initialised to the values given by  $g(x_0(v))$ . The variable  $r(v)$  is initially set to 0. When node  $v$  interacts (in the asynchronous population protocol model),  $v$  updates its state according to the following rules:

1. Run the clock and the shuffling protocol using  $b_0(v), \dots, b_{k-1}(v)$  to hold the  $k$  tokens.
2. If  $c(v) = \vartheta$ , then
  - update the round counter and set  $r(v) \leftarrow \min\{r(v) + 1, R\}$ ,
  - compute the new state  $a(v) \leftarrow f(a(v), b_0(v), \dots, b_{k-1}(v))$ , and
  - generate new tokens  $b_0(v), \dots, b_{k-1}(v) \leftarrow g(a(v))$ .

As output value of the simulation, node  $v$  uses the output value algorithm **A** associates to state  $a(v)$ . The above algorithm simulates an execution of the synchronous algorithm **A** under the schedule  $\sigma_1, \dots, \sigma_R$  given by the shuffling protocol. To this end, define  $x_0(v) = a(v, 0)$  and  $x(r) = a(v, s(v, r))$  for all  $1 \leq r \leq R$ .

► **Lemma 7.** *With high probability, the sequence  $(x_r)_{0 \leq r \leq R}$  is an execution induced by the schedule  $(\sigma_r)_{1 \leq r \leq R}$ .*

## 6.4 From almost-uniform schedules to uniform schedules

The schedules provided by the shuffling protocol are only  $\varepsilon$ -uniform, as the shuffling process is executed for finitely many steps. We now show that this does not matter: any synchronous protocol behaves statistically similarly under  $\varepsilon$ -uniform and uniform schedules.

To formalise this, let  $\Phi$  be the distribution over sequences  $(\sigma_1, \dots, \sigma_R) \in S_N^R$  of permutations generated by the shuffling protocol under the assumption that the clock protocol works correctly for  $T$  time steps. Let  $\nu^R = \nu \times \dots \times \nu$  denote the distribution of a sequence of  $R$  independently and uniformly sampled random permutations from  $S_N$ . That is,  $\nu^R$  is the distribution of the uniform  $R$ -round schedules. The following then holds:

► **Lemma 8.** *The total variation distance between  $\Phi$  and  $\nu^R$  satisfies  $\|\Phi - \nu^R\|_{\text{TV}} \leq \varepsilon R$ .*

Together with the following lemma, we can show that protocols simulated under the  $\varepsilon$ -uniform schedules behave almost the same as under perfectly uniform schedules.

► **Lemma 9.** *Let  $\mu$  and  $\nu$  be probability distributions over a finite domain  $\Omega$ . For any function  $F: \Omega \rightarrow \Omega'$ , the total variation distance satisfies  $\|F(\mu) - F(\nu)\|_{\text{TV}} \leq \|\mu - \nu\|_{\text{TV}}$ .*

## 6.5 The main simulation theorem

With all the pieces now in place, we can now state our simulation theorem.

► **Theorem 4.** *Let  $k > 0$  be a constant and  $\mathbf{A}$  be a synchronous  $k$ -token shuffling protocol on  $n$  nodes, where  $X$  is the set of local states and  $Y$  the set of token types used the protocol  $\mathbf{A}$ . If  $\mathbf{A}$  solves the task  $\Pi$  with high probability in  $R \in \text{poly}(n)$  rounds, then there exists a stochastic population protocol  $\mathbf{B}$  that also solves task  $\Pi$  with high probability on any  $n$ -node  $d$ -regular graph  $G$  with edge expansion  $\beta > 0$ . The step complexity  $T(\mathbf{B})$  and state complexity  $S(\mathbf{B})$  of the protocol  $\mathbf{B}$  satisfy*

$$T(\mathbf{B}) \in O(R \cdot n \cdot \zeta) \quad \text{and} \quad S(\mathbf{B}) \in O(|X| \cdot |Y|^k \cdot \zeta) \quad \text{with} \quad \zeta = \log n \cdot \left( \frac{d}{\beta} + \frac{\tau_{\text{mix}}}{n} \right),$$

where  $\tau_{\text{mix}}$  is the mixing time of the  $k$ -stack interchange process on  $G$ .

## 7 Applications: leader election and exact majority

Using Theorem 4, we can automatically transport algorithms from the *fully-connected synchronous token shuffling model* to the graphical, asynchronous population protocol model. We utilise this result to obtain fast protocols for leader election and exact majority in the graphical population protocol model.

The leader election protocol for the token shuffling model uses a one-way information dissemination protocol and a protocol for generating synthetic coins in the token shuffling model with  $k > 1$ . Specifically, we show the following result.

► **Theorem 10.** *There is a synchronous 2-token shuffling protocol for the leader election task that stabilises in  $O(\log^2 n)$  rounds with high probability, uses  $O(\log n)$  states per node and two token types.*

For exact majority in the token shuffling model, we give an algorithm that simulates two-way interactions in a population of  $2n$  virtual agents. The algorithm uses the classic cancellation-doubling dynamics used in the clique model [13, 34], yielding the following result.

► **Theorem 11.** *There is a synchronous 2-token shuffling protocol for the exact majority task that stabilises in  $O(\log^2 n)$  rounds with high probability, uses  $O(\log n)$  states and five token types.*



## 8 Conclusions

In this work, we established a general framework for simulating clique-based protocols in arbitrary, connected regular graphs. We now conclude by briefly discussing some limitations of our approach and summarise key problems left open by this work.

First, we focused on regular interaction graphs. The justification for this assumption is two-fold. First, this assumption is only used once: in Section 5, to obtain clean bounds for the skew of the phase clock. However, upon close inspection, we notice that this regularity assumption can be relaxed in many cases if the minimum and maximum degrees do not deviate too much from the average degree of the graph [6]. Second, regular graphs give a natural extension of the notion of *parallel time*, since all nodes interact at the same rate.

The simulation overhead has a polylogarithmic dependency on  $n$ . We have made no particular effort to optimise the degree of this polylogarithmic dependency. The dependency can be improved by providing better bounds on the  $k$ -stack interchange process. Indeed, even in the case of the well-studied (1-stack) interchange process, exact bounds on mixing time have been – and still remain – an open question for many graph classes [38]. Improved bounds for these processes imply better running time bounds for our simulations.

Finally, our complexity bounds have a quadratic dependency on  $d/\beta$ . We suspect a polynomial dependency on the expansion properties is necessary for step complexity and leave the investigation of tight space-time trade-offs for population protocols in the general graphical setting as an intriguing open problem.

---

### References

- 1 David Aldous. Random walks on finite groups and rapidly mixing Markov chains. In *Séminaire de Probabilités XVII 1981/82*, pages 243–297. Springer, 1983.
- 2 David Aldous and James Allen Fill. Reversible markov chains and random walks on graphs, 2002. Unfinished monograph, recompiled 2014, available at <http://www.stat.berkeley.edu/users/aldous/RWG/book.html>.
- 3 Dan Alistarh, James Aspnes, David Eisenstat, Rati Gelashvili, and Ronald L Rivest. Time-space trade-offs in population protocols. In *Proc. 28th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2017)*, pages 2560–2579, 2017.
- 4 Dan Alistarh, James Aspnes, and Rati Gelashvili. Space-optimal majority in population protocols. In *Proc. 29th ACM-SIAM Symposium on Discrete Algorithms (SODA 2018)*. SIAM, 2018. doi:10.1137/1.9781611975031.144.
- 5 Dan Alistarh and Rati Gelashvili. Recent algorithmic advances in population protocols. *SIGACT News*, 49(3):63–73, 2018. doi:10.1145/3289137.3289150.
- 6 Dan Alistarh, Rati Gelashvili, and Joel Rybicki. Fast graphical population protocols. Full version. [arXiv:2102.08808](https://arxiv.org/abs/2102.08808).
- 7 Dan Alistarh, Rati Gelashvili, and Milan Vojnović. Fast and exact majority in population protocols. In *Proc. 34th ACM Symposium on Principles of Distributed Computing (PODC 2015)*, pages 47–56, 2015.
- 8 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.
- 9 Dana Angluin, James Aspnes, and David Eisenstat. Stably computable predicates are semilinear. In *Proc. 25th ACM Symposium on Principles of distributed computing (PODC 2006)*, pages 292–299, 2006.
- 10 Dana Angluin, James Aspnes, and David Eisenstat. Fast computation by population protocols with a leader. *Distributed Computing*, 21(3):183–199, 2008. doi:10.1007/s00446-008-0067-z.

- 11 Dana Angluin, James Aspnes, David Eisenstat, and Eric Ruppert. The computational power of population protocols. *Distributed Computing*, 20(4):279–304, 2007.
- 12 Dana Angluin, James Aspnes, Michael J Fischer, and Hong Jiang. Self-stabilizing population protocols. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 3(4):1–28, 2008.
- 13 James Aspnes and Eric Ruppert. An introduction to population protocols. In *Middleware for Network Eccentric and Mobile Applications*, pages 97–120. Springer, 2009.
- 14 Yossi Azar, Andrei Z. Broder, Anna R. Karlin, and Eli Upfal. Balanced allocations. *SIAM Journal on Computing*, 29(1):180–200, 1999.
- 15 Joffroy Beauquier, Peva Blanchard, and Janna Burman. Self-stabilizing leader election in population protocols over arbitrary communication graphs. In *International Conference on Principles of Distributed Systems*, pages 38–52. Springer, 2013. URL: <https://hal.archives-ouvertes.fr/hal-00867287v2>.
- 16 Petra Berenbrink, Robert Elsässer, Tom Friedetzky, Dominik Kaaser, Peter Kling, and Tomasz Radzik. A population protocol for exact majority with  $O(\log_{5/3} n)$  stabilization time and  $\Theta(\log n)$  states. In *Proc. 32nd International Symposium on Distributed Computing (DISC 2018)*, pages 10:1–10:18, 2018. doi:10.4230/LIPIcs.DISC.2018.10.
- 17 Petra Berenbrink, Tom Friedetzky, Peter Kling, Frederik Mallmann-Trenn, and Chris Wastell. Plurality consensus in arbitrary graphs: Lessons learned from load balancing. In *Proc. 24th Annual European Symposium on Algorithms (ESA 2016)*, volume 57, pages 10:1–10:18, 2016. doi:10.4230/LIPIcs.ESA.2016.10.
- 18 Petra Berenbrink, George Giakkoupis, and Peter Kling. Tight bounds for coalescing-branching random walks on regular graphs. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1715–1733. SIAM, 2018.
- 19 Petra Berenbrink, George Giakkoupis, and Peter Kling. Optimal time and space leader election in population protocols. In *Proc. 52nd Annual ACM SIGACT Symposium on Theory of Computing (STOC 2020)*, pages 119–129, 2020.
- 20 Petra Berenbrink, Dominik Kaaser, Peter Kling, and Lena Otterbach. Simple and efficient leader election. In *Proc. 1st Symposium on Simplicity in Algorithms (SOSA 2018)*, pages 9:1–9:11, 2018. doi:10.4230/OASIcs.SOSA.2018.9.
- 21 Michael Blondin, Javier Esparza, and Stefan Jaax. Large flocks of small birds: on the minimal size of population protocols. In *Proc. 35th Symposium on Theoretical Aspects of Computer Science (STACS 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.
- 22 Robert Brijder. Computing with chemical reaction networks: a tutorial. *Natural Computing*, 18(1):119–137, 2019.
- 23 Pietro Caputo, Thomas M. Liggett, and Thomas Richthammer. Proof of Aldous’ spectral gap conjecture. *Journal of the American Mathematical Society*, 23(3):831–851, 2010.
- 24 Hsueh-Ping Chen and Ho-Lin Chen. Self-stabilizing leader election. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 53–59, 2019.
- 25 Hsueh-Ping Chen and Ho-Lin Chen. Self-stabilizing leader election in regular graphs. In *Proceedings of the 39th Symposium on Principles of Distributed Computing*, PODC ’20, pages 210–217, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3382734.3405733.
- 26 Colin Cooper, Robert Elsässer, Hirotaka Ono, and Tomasz Radzik. Coalescing random walks and voting on connected graphs. *SIAM Journal on Discrete Mathematics*, 27(4):1748–1758, 2013. doi:10.1137/120900368.
- 27 Colin Cooper, Tomasz Radzik, Nicolás Rivera, and Takeharu Shiraga. Fast plurality consensus in regular expanders. In *Proc. 31st International Symposium on Distributed Computing (DISC 2017)*, pages 13:1–13:16, 2017. doi:10.4230/LIPIcs.DISC.2017.13.
- 28 Persi Diaconis and Laurent Saloff-Coste. Comparison techniques for random walk on finite groups. *The Annals of Probability*, 21(4):2131–2156, 1993.

- 29 Persi Diaconis and Mehrdad Shahshahani. Generating a random permutation with random transpositions. *Zeitschrift für Wahrscheinlichkeitstheorie und verwandte Gebiete*, 57(2):159–179, 1981.
- 30 AB Dieker. Interlacings for random walks on weighted graphs and the interchange process. *SIAM Journal on Discrete Mathematics*, 24(1):191–206, 2010.
- 31 David Doty, Mahsa Eftekhari, Leszek Gąsieniec, Eric Severson, Grzegorz Stachowiak, and Przemysław Uznański. A time and space optimal stable population protocol solving exact majority. *arXiv preprint*, 2021. [arXiv:2106.10201](https://arxiv.org/abs/2106.10201).
- 32 David Doty and David Soloveichik. Stable leader election in population protocols requires linear time. *Distributed Computing*, 31(4):257–271, 2018.
- 33 Moez Draief and Milan Vojnović. Convergence speed of binary interval consensus. *SIAM Journal on Control and Optimization*, 50(3):1087–1109, 2012.
- 34 Robert Elsässer and Tomasz Radzik. Recent results in population protocols for exact majority and leader election. *Bulletin of the EATCS*, 126, 2018. URL: <http://bulletin.eatcs.org/index.php/beatcs/article/view/549/546>.
- 35 Leszek Gąsieniec and Grzegorz Stachowiak. Fast space optimal leader election in population protocols. In *Proc. 29th ACM-SIAM Symposium on Discrete Algorithms (SODA 2018)*, 2018. doi:10.1137/1.9781611975031.169.
- 36 Leszek Gąsieniec, Grzegorz Stachowiak, and Przemysław Uznański. Almost logarithmic-time space optimal leader election in population protocols. In *Proc. 31st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2019)*, 2019. doi:10.1145/3323165.3323178.
- 37 Leszek Gąsieniec, Grzegorz Stachowiak, and Przemysław Uznański. Time and space optimal exact majority population protocols, 2020. [arXiv:2011.07392](https://arxiv.org/abs/2011.07392).
- 38 Johan Jonasson. Mixing times for the interchange process. *Latin American Journal of Probability and Mathematical Statistics*, 9(2):667–683, 2012.
- 39 Anissa Lamani and Masafumi Yamashita. Realization of periodic functions by self-stabilizing population protocols with synchronous handshakes. In *International Conference on Theory and Practice of Natural Computing*, pages 21–33. Springer, 2016.
- 40 Tom Leighton and Satish Rao. Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. *Journal of the ACM*, 46(6):787–832, 1999.
- 41 David A. Levin and Yuval Peres. *Markov Chains and Mixing Times*. American Mathematical Society, 2 edition, 2017.
- 42 George B. Mertzios, Sotiris E. Nikolettseas, Christoforos Raptopoulos, and Paul G. Spirakis. Determining majority in networks with local interactions and very small local memory. In *Proc. 41st International Colloquium on Automata, Languages, and Programming (ICALP 2014)*, pages 871–882, 2014. doi:10.1007/978-3-662-43948-7\_72.
- 43 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. *Distributed Computing*, 30(1):1–16, 2017.
- 44 Yuval Peres, Kunal Talwar, and Udi Wieder. Graphical balanced allocations and the  $(1 + \beta)$ -choice process. *Random Structures and Algorithms*, 47(4):760–775, 2014. doi:10.1002/rsa.20558.
- 45 Thomas Sauerwald and He Sun. Tight bounds for randomized load balancing on arbitrary network topologies. In *Proc. 53rd Annual IEEE Symposium on Foundations of Computer Science (FOCS 2012)*, pages 341–350, 2012. doi:10.1109/FOCS.2012.86.
- 46 Yuichi Sudo, Fukuhito Ooshita, Taisuke Izumi, Hirotsugu Kakugawa, and Toshimitsu Masuzawa. Logarithmic expected-time leader election in population protocol model. In *Proc. International Symposium on Stabilizing, Safety, and Security of Distributed Systems (SSS 2019)*, pages 323–337, 2019.
- 47 Yuichi Sudo, Fukuhito Ooshita, Hirotsugu Kakugawa, Toshimitsu Masuzawa, Ajoy K Datta, and Lawrence L Larmore. Loosely-stabilizing leader election for arbitrary graphs in population protocol model. *IEEE Transactions on Parallel and Distributed Systems*, 30(6):1359–1373, 2018.

## 14:18 Fast Graphical Population Protocols

- 48 Yuichi Sudo, Masahiro Shibata, Junya Nakamura, Yonghwan Kim, and Toshimitsu Masuzawa. Self-stabilizing population protocols with global knowledge. *IEEE Transactions on Parallel and Distributed Systems*, 2021.
- 49 Alan M. Turing. The chemical basis of morphogenesis. *Philosophical Transactions of the Royal Society of London, Series B*, 237(641):37–72, 1952.
- 50 David B. Wilson. Mixing times of lozenge tiling and card shuffling Markov chains. *The Annals of Applied Probability*, 14(1):274–325, 2004.
- 51 Daisuke Yokota, Yuichi Sudo, and Toshimitsu Masuzawa. Time-optimal self-stabilizing leader election on rings in population protocols. In *International Symposium on Stabilizing, Safety, and Security of Distributed Systems*, pages 301–316. Springer, 2020.

# Beyond Distributed Subgraph Detection: Induced Subgraphs, Multicolored Problems and Graph Parameters

Amir Nikabadi ✉

ENS de Lyon, France

Janne H. Korhonen ✉

IST Austria, Klosterneuburg, Austria

---

## Abstract

Subgraph detection has recently been one of the most studied problems in the CONGEST model of distributed computing. In this work, we study the distributed complexity of problems closely related to subgraph detection, mainly focusing on *induced subgraph detection*. The main line of this work presents lower bounds and parameterized algorithms w.r.t *structural parameters* of the input graph:

- On general graphs, we give unconditional lower bounds for induced detection of cycles and patterns of treewidth 2 in CONGEST. Moreover, by adapting reductions from centralized parameterized complexity, we prove lower bounds in CONGEST for detecting patterns with a 4-clique, and for induced path detection conditional on the hardness of triangle detection in the congested clique.
- On graphs of bounded degeneracy, we show that induced paths can be detected fast in CONGEST using techniques from parameterized algorithms, while detecting cycles and patterns of treewidth 2 is hard.
- On graphs of bounded vertex cover number, we show that induced subgraph detection is easy in CONGEST for any pattern graph. More specifically, we adapt a centralized parameterized algorithm for a more general *maximum common induced subgraph detection* problem to the distributed setting.

In addition to these induced subgraph detection results, we study various related problems in the CONGEST and congested clique models, including for *multicolored* versions of subgraph-detection-like problems.

**2012 ACM Subject Classification** Theory of computation → Distributed algorithms; Theory of computation → Computational complexity and cryptography

**Keywords and phrases** distributed algorithms, parameterized distributed complexity, CONGEST model, induced subgraph detection, graph parameters, lower bounds

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2021.15

**Related Version** *Full Version*: <https://arxiv.org/abs/2109.06561> [32]

**Funding** *Amir Nikabadi*: Supported by the LABEX MILYON (ANR-10-LABX-0070) of Université de Lyon, within the program “Investissements d’Avenir” (ANR-11-IDEX-0007) operated by the French National Research Agency (ANR).

*Janne H. Korhonen*: Supported by the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agreement No 805223 ScaleML).

**Acknowledgements** We thank François Le Gall and Masayuki Miyamoto for sharing their work on lower bounds for induced subgraph detection [36].



© Amir Nikabadi and Janne H. Korhonen;

licensed under Creative Commons License CC-BY 4.0

25th International Conference on Principles of Distributed Systems (OPODIS 2021).

Editors: Quentin Bramas, Vincent Gramoli, and Alessia Milani; Article No. 15; pp. 15:1–15:18

Leibniz International Proceedings in Informatics



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

## 1 Introduction

*Subgraph detection* is one of the most studied problems in the CONGEST and congested clique models of distributed computing [23, 16, 33, 25, 26, 29, 28, 15, 14]. The complexity of distributed subgraph detection is understood for many pattern graphs – for example, in the CONGEST model, tight bounds are known for *path* [33, 26] and *odd cycle* detection [33, 25], and it is known that pattern graphs requiring almost quadratic time exist [28]. However, unresolved questions remain about the exact complexity of, e.g., *triangle* detection in either CONGEST or congested clique, and *even cycle* detection in CONGEST.

In this work, we look at the closely related *induced subgraph detection* problem, which has so far not received any attention in the distributed setting. In particular, we aim to understand the complexity of induced subgraph detection for common pattern graphs, such as paths and cycles, as well as how the situation contrasts with the non-induced case. It is well known that in the centralized setting, induced subgraph detection is generally more difficult than non-induced subgraph detection, so one would expect that situation is the same also in the distributed setting.

### 1.1 Background and setting

Before presenting our results, we start by discussing the wider context of distributed subgraph detection problems. As mentioned above, we work in the CONGEST and congested clique models of distributed computing, and use  $G$  and  $n$  to denote the input graph and the number number of nodes in the input graph, respectively.

In the paper, we mostly consider subgraph detection and induced subgraph detection problems; we are given a pattern graph  $H$  with  $k$  nodes, known to all nodes in  $G$ , and the task is to decide if the input graph  $G$  contains  $H$  as a subgraph or an induced subgraph; more precisely, any node  $v$  that is part of an admissible copy of  $H$  should report that the input is a yes-instance.

**Fixed-parameter tractability.** Subgraph and induced subgraph detection problems can be viewed as *parameterized problems*; such problems are studied in centralized setting under the field of *parameterized complexity* [20]. A parameterized problem is defined by the input and a problem parameter  $k$  – formally, a (*complexity*) *parameter*  $k$  is a mapping from the input instance to natural numbers. The basic question of centralized parameterized complexity is to understand which problems are *fixed-parameter tractable*, i.e. have algorithms with running time  $f(k)|x|^{O(1)}$ , where  $f$  is an arbitrary function and  $x$  is the binary encoding of the input instance. For example,  $k$ -cycle detection can be viewed as a parameterized problem.

Similarly, one can consider fixed-parameter tractability in the distributed setting. The strictest definition is to ask which problems have distributed algorithm where the running time depends only on the parameter  $k$  [43, 10]. However, this arguably does not capture all fixed-parameter tractability phenomena in distributed models – e.g.  $k$ -cycle detection cannot be solved in  $f(k)$  rounds for any function  $f$  in the CONGEST model.

A more general perspective is to ask what is the smallest function  $T$  such that a parameterized problem can be solved in  $f(k) \cdot T(n)$  rounds, for some function  $f: \mathbb{N} \rightarrow \mathbb{N}$ . Several results of this type are known for subgraph detection problems; for example,  $k$ -cycle detection can be solved in  $O(k2^k n)$  rounds in the CONGEST model [33, 26], and in  $2^{O(k)} n^{0.158}$  rounds in the congested clique model [16], though these bounds are not tight for *even-length* cycles [15, 28].

**Parameters and graph structure.** For subgraph and induced subgraph detection problems, the natural complexity parameter is the number of nodes  $k$  in the pattern graph. However, parameterized complexity frequently studies other complexity parameters – for our purposes, the most relevant are structural graph parameters, in particular degeneracy  $d(G)$ , treewidth  $\text{tw}(G)$ , and vertex cover number  $\tau(G)$  (see Section 2 for the precise definitions). While bounded degeneracy (equivalently, bounded arboricity) [7, 33, 8] has been studied in the distributed setting, bounded treewidth and bounded vertex cover number less so.

Given a structural parameter  $p$ , we can consider the complexity of subgraph or induced subgraph detection parameterized either by the structural parameter  $p(G)$  of the input graph, or by the structural parameter  $p(H)$  of the pattern graph. Note that we have

$$d(G) \leq \text{tw}(G) \leq \tau(G).$$

For parameters  $p_1$  and  $p_2$  with  $p_1(G) \leq p_2(G)$ , upper bounds w.r.t. parameter  $p_2$  imply upper bounds w.r.t. parameter  $p_1$ , and lower bounds w.r.t. parameter  $p_1$  imply lower bounds w.r.t. parameter  $p_2$ .

**Lower bounds and reductions.** The standard technique for proving unconditional CONGEST lower bounds is by reduction from communication complexity problems, most often using *families of lower bound graphs* [42, 25, 17, 1, 30, 21] (see Section 2). By contrast, reductions between problems are less useful in the CONGEST model, as the model can implement only very limited reductions efficiently.

However, there are still uses for reductions in distributed complexity theory, which we will apply in this work. First, in the congested clique, sub-polynomial round reductions can be used to establish relative complexities of problems [34]. Second, as noted by Bacrach et al. [6], centralized reductions can be used to transform families of lower bound graphs for one problem into families of lower bound graphs for a second problem.

## 1.2 Results: induced subgraph detection on general graphs

First, we consider the hardness of induced subgraph detection on general graphs. We show that for common pattern graphs, the induced version of the problem is at least as hard as the non-induced version, and in many cases harder.

**Unconditional lower bounds.** We start with unconditional lower bounds for induced subgraph detection in CONGEST; see Table 1 for a summary of these results.

For cycles of length at least 6, we show that the induced cycle detection problem requires at least  $O(n/\log n)$  rounds in the CONGEST model. The result follows from a combination of the existing lower bound construction for odd-length cycles, and a new construction for induced even cycles. By comparison, the existing lower bounds for non-induced subgraph detection in CONGEST are  $\Omega(n^{1/2}/\log n)$  for even cycle detection [33], and  $\Omega(n/\log n)$  for odd cycle detection excluding triangles [25]; it is also known that even cycles can be detected in  $O(n^\delta)$  time, for  $\delta < 1$  that depends on the length of the cycle [28].

We also prove that there are pattern graphs for which induced subgraph detection (and also non-induced detection) requires near-quadratic time in CONGEST, in similar spirit at the hard pattern graphs for non-induced subgraph detection presented by Fischer et al. [28]. Moreover, we show that these pattern graphs can be constructed to have treewidth 2; contrast this with the centralized setting, where low-treewidth patterns are easy to detect [5].

## 15:4 Beyond Distributed Subgraph Detection

■ **Table 1** Lower bounds on general graphs. Improved lower bounds of Le Gall and Miyamoto [36] are independent and concurrent work (see main text.)

Problem	Bound	
Induced $2k$ -cycle ( $k \geq 3$ )	$\Omega(n/\log n)$	Section 3.3
Induced $H$ -detection		
· any $H$ with 4-clique	$\Omega(n^{1/2}/\log n)$	Section 3.2
· some $H$ with $\text{tw}(H) = 2^\dagger$	$\Omega(n^{2-\varepsilon})$	Section 3.4
Multicolored $k$ -cycle ( $k \geq 4$ )	$\Omega(n/\log n)$	Section 4.2
Multicolored induced path of length $k$ ( $k \geq 6$ )	$\Omega(n/\log n)$	Section 4.2
Induced $k$ -cycle ( $k \geq 4$ )	$\tilde{\Omega}(n)$	[36]
Induced $k$ -cycle ( $k \geq 8$ )	$\tilde{\Omega}(n^{2-\Theta(1/k)})$	[36]

<sup>†</sup>holds for any  $\varepsilon > 0$ , for some  $H$  that is chosen depending  $\varepsilon$

■ **Table 2** Bounds w.r.t. structural graph parameters. Results attributed to [33] follow directly from the proofs in that work, but are not stated in that work for induced subgraphs.

Problem	Bound	
Induced $k$ -tree <sup>†</sup>	$2^{O(kd(G))}k^k + O(\log n)$	Section 5
(Induced) $H$ -detection, some $H$ with $\text{tw}(H) = 2^\ddagger$	$\Omega(n^{1-\varepsilon})$	holds for $d(G) = 2$ Section 5.2
(Induced) $k$ -cycle ( $k \geq 6$ )	$\Omega(n^{1/2}/\log n)$	holds for $d(G) = 2$ [33]
Induced 4-cycle	$O(d(G) + \log n)$	[33]
Induced 5-cycle	$O(d(G)^2 + \log n)$	[33]
MCIS	$2^{O(\tau^2)}$	$\tau = \tau(G) + \tau(H)$ Section 6
Induced subgraph	$2^{O((\tau(G)+k)^2)}$	Section 6

<sup>†</sup>randomized algorithm, can be derandomized with extra assumptions and worse running time

<sup>‡</sup>holds for any  $\varepsilon > 0$ , for some  $H$  that is chosen depending  $\varepsilon$

**Unconditional lower bounds: recent independent work.** After submitting this paper, we learned about the independent and concurrent work of Le Gall and Miyamoto [36], which gives lower bounds for induced cycle detection and diamond listing. In particular, they show that detecting induced  $k$ -cycles requires  $\tilde{\Omega}(n)$  rounds for any  $k \geq 4$ , and  $\tilde{\Omega}(n^{2-\Theta(1/k)})$  rounds for any  $k \geq 8$ . These results subsume our lower bounds for induced cycle and treewidth-2 subgraph detection.

**Reductions.** Next, we turn our attention to conditional lower bounds for problems where standard CONGEST lower bound techniques do not immediately yield unconditional lower bounds. See Figure 1 for a summary of these results.

We adapt a centralized reduction of Dalirrooyfard et al. [22] between clique and independent set detection and induced subgraph detection. Specifically, they show that detecting an induced subgraph  $H$  that contains a  $k$ -clique ( $k$ -independent set) is as hard detecting  $k$ -clique ( $k$ -independent set, resp.). We show that this reduction can also be implemented in the congested clique model.



It follows that detecting induced paths of length at least 5 in either the CONGEST or congested clique model is at least as hard *triangle detection* in the congested clique model, and more generally, detecting paths of length at least  $2k - 1$  in CONGEST or congested clique is as hard as detecting  $k$ -cliques in the congested clique. By comparison, the best known upper bounds in the congested clique are  $O(n^{0.158})$  for triangle detection [16], and  $O(n^{1-1/k})$  for  $k$ -clique detection [23]; while no lower bounds for the congested clique model are known, improving over the  $O(n^{0.158})$ -round matrix multiplication based triangle detection would have major implications for distributed algorithms. However, it is worth noting that induced paths of length 2 can be detected in  $O(1)$  rounds in CONGEST, in contrast to triangles (see Appendix A).

Moreover, the reduction allows us to lift the  $\Omega(n^{1/2}/\log n)$  CONGEST lower bound of Czumaj and Konrad [21] for 4-clique detection to induced and non-induced detection of any pattern graph  $H$  that contains a 4-clique.

**Multicolored problems.** Finally, we consider *multicolored* versions of subgraph detection tasks. In multicolored (induced)  $H$ -detection, we are given a labelling of the input graph  $G$  with  $k$  colors, and the task is to find a (induced) copy of  $H$  that contains exactly one node of each color. Multicolored versions of problems have proven to be useful starting points for reductions in fixed-parameter complexity, and algorithms for a multicolored version of a problem can often be turned into an algorithm for the standard version via color-coding [5].

We observe that multicolored versions of  $k$ -clique and  $k$ -independent set are closely related to their standard versions in the distributed setting, by adapting the simple centralized reductions to distributed setting (see Figure 1). We then prove unconditional lower bounds of  $\Omega(n/\log n)$  in CONGEST for multicolored versions of  $k$ -cycle detection, for  $k \geq 4$ , and for detection of induced paths of length  $k$ , for  $k \geq 6$ . These results imply that color-coding algorithms cannot be used directly to improve the state of the art for these problems – for comparison, note that  $k$ -cycle detection can be solved in CONGEST in  $o(n/\log n)$  rounds for even  $k$ , non-induced multicolored paths can be detected in  $O(1)$  round in CONGEST, and we have no unconditional lower bounds for induced path detection.

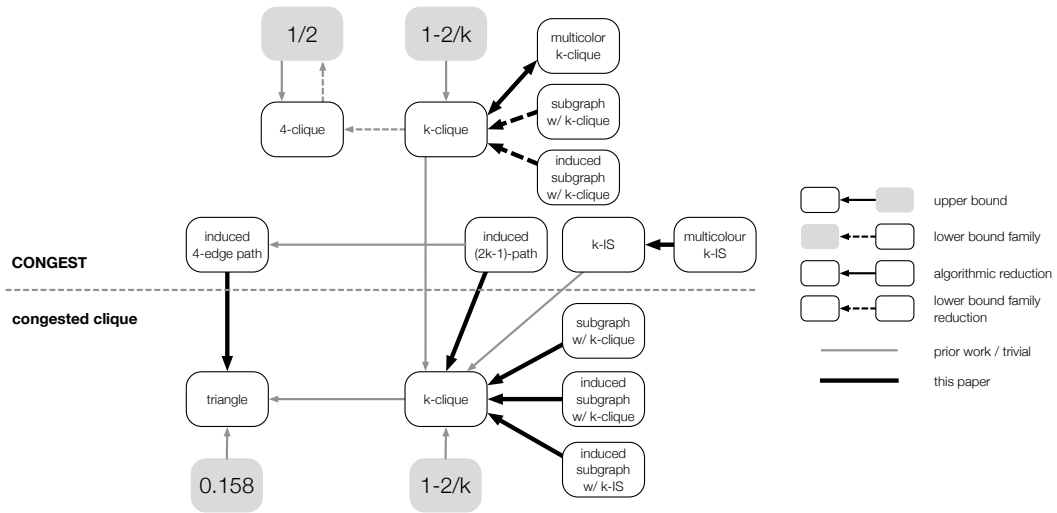
### 1.3 Results: induced subgraph detection with structural parameters

Next, we consider subgraph and induced subgraph detection tasks w.r.t. structural graph parameters. We focus on the degeneracy  $d(G)$  and the vertex cover number  $\tau(G)$  of the input graph as the parameters in this section. See Table 2 for a summary of the results.

**Bounded degeneracy.** We show that induced subgraph detection for any *tree* on  $k$  nodes can be solved in time  $2^{O(kd(G))}k^k + O(\log n)$  rounds in CONGEST. As with the prior results on non-induced path, tree and cycle detection algorithms in CONGEST, this upper bound is based on centralized fixed-parameter algorithms, in this case using color-coding and random separation techniques [4, 13].

On the lower bounds side, we show that there are treewidth 2 pattern graphs that require near-linear time to detect as induced and non-induced subgraphs in CONGEST on input graphs of degeneracy  $d(G) = 2$ , via a slight modification of the proof for the general case discussed above. Note that any fixed pattern graph can be detected in  $O(n)$  rounds when degeneracy is bounded, by having all nodes gather their distance- $k$  neighborhood.

For cycles, we note that results of Korhonen and Rybicki [33] can be easily seen to imply that detecting induced  $k$ -cycles for  $k \geq 6$  requires at least  $\Omega(n^{1/2}/\log n)$  rounds to detect in CONGEST on graphs of degeneracy  $d(G) = 2$ , as well as that induced 4-cycles can be detected in  $O(d(G) + \log n)$  rounds, and induced 5-cycles in  $O(d(G)^2 + \log n)$  rounds.



■ **Figure 1** Relationships between problems in CONGEST and congested clique. Results hold for any sufficiently large constant  $k$ . *Upper bound* indicates an  $\tilde{O}(n^\delta)$  round algorithm for the problem for specified  $\delta$ ; *lower bound family* indicates that there is a lower bound family giving  $\tilde{\Omega}(n^\delta)$  lower bound for the problem for specified  $\delta$ ; *algorithmic reduction* from  $P_1$  to  $P_2$  indicates that an algorithm solving  $P_2$  in  $O(n^\delta)$  rounds implies the existence of an algorithm solving  $P_1$  in  $\tilde{O}(n^\delta)$  rounds, for any  $\delta > 0$ , and *lower bound reduction* from  $P_1$  to  $P_2$  indicates that a lower bound family giving  $\Omega(n^\delta)$  lower bound for  $P_1$  implies the existence of a lower bound family giving  $\tilde{\Omega}(n^\delta)$  lower bound for  $P_2$  for any  $\delta > 0$ . Notation  $\tilde{O}$  and  $\tilde{\Omega}$  hides polylogarithmic factors in  $n$ , as well as factors only depending on  $k$ , as we assume  $k$  to be constant.

**Bounded vertex cover number.** For a more restrictive parameter than degeneracy, we consider induced subgraph detection parameterized by the vertex cover number  $\tau(G)$  of the input graph. More precisely, we show a more general problem of *maximum common induced subgraph (MCIS)* can be solved fast; in this problem, we are given two graphs  $G = (V_G, E_G)$  and  $H = (V_H, E_H)$  as input, and the task is to find the maximum-size graph  $G^*$  such that  $G^*$  appears as induced subgraph of both  $G$  and  $H$ . In the distributed setting, we assume that  $G$  is the input graph, and the second graph  $H$  is known to every node.

In more detail, we show that a centralized branching algorithm from MCIS of Abu-Khzam et al. [3] can be implemented in  $2^{O((\tau(G)+\tau(H))^2)}$  rounds, i.e. without dependence on  $n$ , in the CONGEST model. This immediately implies that induced subgraph detection for any pattern graph  $H$  on  $k$  nodes can also be solved in  $2^{O((\tau(G)+k)^2)}$  rounds.

### 1.4 Additional related work

**Centralized subgraph and induced subgraph detection.** Subgraph detection has been widely studied in the centralized parameterized setting. Fixed-parameter algorithms, parameterized by the number of nodes  $k$  of the pattern graph, are known for example for paths [38, 5, 45, 11], trees [5], even cycles [46], odd cycles [5], and patterns of constant treewidth [5]. By contrast,  $k$ -clique detection is known to be W[1]-hard, suggesting that it does not have a fixed-parameter algorithm [24].

Induced subgraph detection, on the other hand, is W[1]-hard even for paths of length  $k$  [19]. Any induced or non-induced subgraph on  $k$  nodes can be detected in  $n^{\omega k/3+O(1)}$  time, where  $\omega < 2.3729$  is the matrix multiplication exponent, due to a classical result of Nešetřil and Poljak [39].

**Distributed subgraph detection.** As mentioned above, distributed subgraph detection has also received attention in the distributed setting recently. In CONGEST, non-trivial upper bounds are known e.g. for path and tree detection [33, 26], cycle detection [29, 33, 28] and clique detection [14]. Likewise, lower bounds have been studied for cycle detection [25, 33] and cliques [21], and pattern graphs requiring near-quadratic time are known to exist [28]. Triangle detection remains a particularly interesting open question—the best known upper bound is  $n^{1/3+o(1)}$  rounds [18], but no lower bounds are known.

In the congested clique, triangles can be detected in  $O(n^{0.158})$  rounds and odd  $k$ -cycles in  $2^{O(k)}n^{0.158}$  rounds using fast matrix multiplication [16]. Even cycles can be detected even faster, in  $O(k^2)$  rounds for  $k = O(\log n)$  [15]. Moreover, any induced or non-induced subgraph detection for  $k$ -node patterns can be solved in  $O(n^{1-2/k})$  rounds in congested clique [23].

**Distributed parameterized complexity.** Parameterized distributed algorithms have appeared implicitly in many of the above-mentioned subgraph detection works, and recently Ben-Basat et al. [10] and Siebertz and Vigny [43] have explicitly studied aspects of distributed parameterized complexity. In terms of structural parameters, *maximum degree* is a standard parameter in distributed setting, and algorithms parameterized degeneracy has been studied for various problems and models [9, 7, 31]. Recently, Li [37] has show that the treewidth of the input graph can be approximated in  $\tilde{O}(D)$  rounds in CONGEST, and many classical optimization problems that are fixed-parameter tractable w.r.t. treewidth can be solved in  $\tilde{O}(\text{tw}(G)^{O(\text{tw}(G))}D)$  rounds in CONGEST, where  $\tilde{O}$  hides polylogarithmic factors in  $n$ .

## 2 Preliminaries

**Degeneracy.** A graph  $G$  is called  *$d$ -degenerate* if every induced subgraph of  $G$  has a vertex of degree at most  $d$ . The minimum number  $d$  for which  $G$  is  *$d$ -generate* is called *degeneracy* of  $G$ , denoted by  $d(G)$ . It is easy to see that every  $d$ -degenerate graph admits an acyclic orientation such that the out-degree of each vertex is at most  $d$ .

**Vertex cover number.** A *vertex cover* of  $G$  is a subset of vertices  $S \subseteq V(G)$  such that every edge in  $E(G)$  is incident with at least one vertex in  $S$ . The *vertex cover number*  $\tau(G)$  of  $G$  is the minimum size of a vertex cover of  $G$ .

**Treewidth.** A *tree decomposition* of a graph  $G = (V, E)$  is a pair  $(\mathcal{X}, T)$ , where  $\mathcal{X} = \{X_1, X_2, \dots, X_m\}$  is a collection of subsets of  $V$  and  $T$  is a tree on  $\{1, 2, \dots, m\}$ , such that

1.  $\bigcup_{i=1}^m X_i = V$ ,
2. for all edges  $e \in E$  there exist  $i$  with  $e \subseteq X_i$
3. for all  $i, j$  and  $k$ , if  $j$  is on the (unique) path from  $i$  to  $k$  in  $T$ , then  $X_i \cap X_k \subseteq X_j$ .

The *width* of a tree-decomposition  $(\mathcal{X}, T)$  is defined as  $\max_i |X_i| - 1$ . The *treewidth* of a graph  $G$  is the minimum width over all possible tree decompositions of  $G$ . Connected graphs of treewidth 1 are trees, and connected graphs of treewidth 2 are *series-parallel graphs* (see e.g. [12].)

**Lower bound families.** For unconditional lower bounds in the CONGEST model, we use the standard framework of reducing from two-party communication complexity. Let  $f: \{0, 1\}^{2k} \rightarrow \{0, 1\}$  be a Boolean function. In the two-party communication game on  $f$ , there are two players who receive a private  $k$ -bit string  $x_0$  and  $x_1$  as input, and the task is to have at least one of the players compute  $f(x) = f(x_0, x_1)$ .

The template for these reductions is captured by *families of lower bound graphs*:

► **Definition 1** (e.g. [25, 30, 1]). Let  $f_n: \{0, 1\}^{2k(n)} \rightarrow \{0, 1\}$  and  $C: \mathbb{N} \rightarrow \mathbb{N}$  be functions and  $\Pi$  a graph predicate. Suppose there is  $n_0$  such that for any  $n \geq n_0$  and all  $x_0, x_1 \in \{0, 1\}^{k(n)}$  there exists a (weighted) graph  $G(n, x_0, x_1)$  satisfying the following properties:

1.  $G(n, x_0, x_1)$  satisfies  $\Pi$  if and only if  $f_n(x_0, x_1) = 1$ ,
2.  $G(n, x_0, x_1) = (V_0 \cup V_1, E_0 \cup E_1 \cup S)$ , where
  - a.  $V_0$  and  $V_1$  are disjoint and  $|V_0 \cup V_1| = n$ ,
  - b.  $E_i \subseteq V_i \times V_i$  for  $i \in \{0, 1\}$ ,
  - c.  $S \subseteq V_0 \times V_1$  is a cut and has size at least  $C(n)$ , and
  - d. subgraph  $G_i = (V_i, E_i)$  only depends on  $i$ ,  $n$  and  $x_i$ , i.e.,  $G_i = G_i(n, x_i)$ .

We then say that  $\mathcal{F} = (\mathcal{G}(n))_{n \in I}$  is a family of lower bound graphs, where

$$\mathcal{G}(n) = \{G(n, x_0, x_1) : x_0, x_1 \in \{0, 1\}^{k(n)}\}.$$

*Deterministic communication complexity*  $\text{CC}(f)$  of a function  $f$  is the maximum number of bits the two players need to exchange in the worst case, over all deterministic protocols and input strings, in order to compute  $f(x_0, x_1)$ . *Randomized communication complexity*  $\text{RCC}(f)$  is the worst-case complexity of protocols which compute  $f$  with probability at least  $2/3$  on all inputs.

► **Theorem 2** (e.g. [25, 30, 1]). Let  $\mathcal{F}$  be a family of lower bound graphs. Any algorithm deciding  $\Pi$  on a graph family  $\mathcal{H}$  containing  $\bigcup \mathcal{G}(n)$  for all  $n \geq n_0$  in the CONGEST model with bandwidth  $b(n)$  needs  $\Omega(\text{CC}(f_n)/C(n)b(n))$  and  $\Omega(\text{RCC}(f_n)/C(n)b(n))$  deterministic and randomized rounds, respectively.

We reduce from the two-player *set disjointness* function  $\text{DISJ}_n: \{0, 1\}^{2n} \rightarrow \{0, 1\}$ , defined as  $\text{DISJ}_n(x_0, x_1) = 0$  if and only there is  $i \in [n]$  such that  $x_0(i) = x_1(i) = 1$ . The communication complexity of set disjointness is  $\text{CC}(\text{DISJ}_n) = \Omega(n)$  and  $\text{RCC}(\text{DISJ}_n) = \Omega(n)$  [35, 41].

### 3 Induced subgraph detection on general graphs

#### 3.1 Patterns with cliques and independent sets: framework

For the complexity results on detecting pattern graphs that contain large independent sets or clique, we borrow the centralized reduction of of Dalirrooyfard et al. [22]. We present the reduction here in full, as we will need to analyze its implementation in distributed setting.

We will start from instance  $G$  of  $s$ -clique detection. The reduction will transform  $G$  into an instance of (induced)  $H$ -detection, where the pattern graph  $H$  contains a clique of size  $s$ , while increasing the number of nodes by a small factor. We first need the following definition:

► **Definition 3** ([22]). Let  $G = (V, E)$  be a graph. A family  $\mathcal{C} \subseteq 2^V$  is an  $s$ -clique cover if

1. for each  $s$ -clique  $K$  in  $G$ , there is a  $C \in \mathcal{C}$  that contains the nodes of  $K$ , and
2. the induced subgraph  $G[C]$  is  $s$ -colorable for each  $C \in \mathcal{C}$ .

We say that  $\mathcal{C}$  is a minimum  $s$ -clique cover if all  $s$ -clique covers of  $G$  have at least  $|\mathcal{C}|$  sets.

Note that if  $\mathcal{C}$  is a minimum  $s$ -clique cover, all induced subgraphs  $G[C]$  for  $C \in \mathcal{C}$  contain an  $s$ -clique, and thus require exactly  $s$  colors to color.

**Reduction overview.** Let  $G = (V_G, E_G)$  be the original graph and let  $H = (V_H, E_H)$  be the pattern graph. Let  $\mathcal{C} = \{C_1, C_2, \dots, C_t\}$  be a minimum  $s$ -clique cover of  $H$ . We construct a graph  $G^*$  as from the input graph  $G$  follows:

1. The node set  $V_{G^*}$  of  $G^*$  consists of the following nodes:
  - a. For each  $i \in C_1$ , there is a copy  $V_{G,i} = V_G \times \{i\}$  of the node set of  $G$ .
  - b. For each  $j \in V_H \setminus C_1$ , there is a copy  $j^*$  of the node  $j$  in  $G^*$ .
2. The edge set of  $G^*$  is defined by the following rules:
  - a. Each  $V_{G,i}$  is an independent set.
  - b. For each  $i, j \in C_1$  and  $v, u \in V_G$ , we add edge between  $(v, i)$  and  $(u, j)$  if both  $\{i, j\} \in E_H$  and  $\{v, u\} \in E_G$ .
  - c. For each  $i \in C_1$  and  $j \in V_H \setminus C_1$  with  $\{i, j\} \in E_H$ , we add edges between  $j^*$  and all nodes  $(v, i)$  for  $v \in V_G$ .
  - d. For each  $i, j \in V_H \setminus C_1$  with  $\{i, j\} \in E_H$ , we add edge between  $i^*$  and  $j^*$ .

Note that the graph  $G^*$  has  $sn + |V_H|$  nodes.

► **Lemma 4** ([22]). *If  $G$  has an  $s$ -clique, then  $G^*$  has  $H$  as an induced subgraph, and if  $G^*$  has  $H$  as a subgraph, then  $G$  has an  $s$ -clique.* (▷ See full version.)

### 3.2 Patterns with cliques and independent sets: implications

**Implementing the reduction in the congested clique.** Let  $H$  be a pattern graph on  $k$  nodes containing an  $s$ -clique. We now show that the reduction we gave above can be implemented efficiently in the congested clique model.

Assume we have algorithm  $\mathcal{A}$  for (induced)  $H$ -detection running in  $O(n^\delta)$  rounds in the congested clique. We now show that we can implement the above reduction in the congested clique to obtain an algorithm for detecting an  $s$ -clique, as follows:

1. Each node  $v \in V_G$  simulates nodes  $(v, i)$  for  $i \in C_1$ , as well as one node from  $V_H$ .
2. Since the incident edges of  $(v, i)$  for  $i \in C_1$  and nodes in  $V_H \setminus C_1$  in  $G^*$  only depend on the pattern graph  $H$  and on the edges incident to  $v$  in  $G$ , node  $v$  can construct the inputs of its simulated nodes locally.
3. Nodes then simulate the execution of  $\mathcal{A}$  on a congested clique with  $O(sn + k) = O(kn)$  nodes. The running time of  $\mathcal{A}$  on the simulated instance is  $O((kn)^\delta)$ , and the simulation incurs additional overhead of  $O(k^2)$ , for a total running time of  $O(k^{2\delta}n^\delta)$ .

Thus, we obtain the following:

► **Theorem 5.** *Let  $H$  be a pattern graph with  $k$  nodes that has a clique of size  $s$ . Then if we can solve  $H$ -detection or induced  $H$ -detection in the congested clique model in  $O(n^\delta)$  rounds, we can find an  $s$ -clique in the congested clique in  $O(k^{2\delta}n^\delta)$  rounds.*

As an immediate corollary, we obtain a similar hardness result for induced subgraph detection for pattern graphs with large independent set, by observing that we can simply complement the pattern and input graphs. Note that this version only applies for *induced* subgraph detection.

► **Corollary 6.** *Let  $H$  be a pattern graph with  $k$  nodes that has an independent set of size  $s$ . Then if we can solve induced  $H$ -detection in the congested clique model in  $O(n^\delta)$  rounds, we can find an  $s$ -clique in the congested clique in  $O(k^{2\delta}n^\delta)$  rounds.*

## 15:10 Beyond Distributed Subgraph Detection

**Induced path detection.** Corollary 6 immediately implies a conditional lower bound for induced path detection in the CONGEST model, as paths contain large independent sets:

► **Corollary 7.** *Let  $k$  be fixed. If an induced  $2k$ -edge path or an induced  $(2k + 1)$ -edge path can be detected in  $O(n^\delta)$  rounds in the CONGEST model, then a  $k$ -clique can be detected in  $O(n^\delta)$  rounds in the congested clique model. In particular, if an induced 4-edge path can be detected in  $O(n^\delta)$  rounds in the CONGEST model, then triangles can be detected  $O(n^\delta)$  rounds in the congested clique model.*

**Patterns with cliques in CONGEST.** As a further application of the reduction of Dalirrooyfard et al. [22], we can transform the unconditional lower bound of Czumaj and Konrad [21] for 4-clique detection in CONGEST into a lower bound for induced subgraph detection for any pattern containing a 4-clique.

► **Lemma 8** ([21]). *Let  $\Pi$  the graph predicate for existence of a 4-clique. There exists a family of lower bound graphs for  $\Pi$  with  $f_n = \text{DISJ}_{\Theta(n^2)}$  and  $C(n) = \Theta(n^{3/2})$ .*

► **Lemma 9.** *Let  $H$  be a pattern graph on  $k$  nodes that contains a 4-clique, and let  $\Pi$  the graph predicate for existence of either induced or non-induced copy of  $H$ . Then there exists a family of lower bound graphs for  $\Pi$  with  $f_n = \text{DISJ}_{\Theta(n^2)}$  and  $C(n) = \Theta(n^{3/2})$ .*

(▷ See full version.)

Theorem 2 and Lemma 9 now immediately imply the following:

► **Theorem 10.** *Let  $H$  be a pattern graph that contains a 4-clique. Any CONGEST algorithm solving either  $H$ -detection or induced  $H$ -detection needs at least  $\Omega(n^{1/2}/\log n)$  rounds.*

### 3.3 Induced even cycle detection

We next prove an unconditional lower bound for induced even cycle detection in CONGEST. Note that for induced odd cycles, one can easily verify that the construction of Drucker et al. [25] immediately implies a  $\Omega(n/\log n)$  lower bound.

► **Lemma 11.** *Let  $k \geq 3$  be fixed, and let  $\Pi$  the graph predicate for existence of an induced  $2k$ -cycle. There exists a family of lower bound graphs for  $\Pi$  with  $f_n = \text{DISJ}_{\Theta(n^2)}$  and  $C(n) = n$ .*

(▷ See full version.)

Theorem 2 and Lemma 11 immediately imply the following:

► **Theorem 12.** *Any CONGEST algorithm solving induced  $2k$ -cycle detection for  $k \geq 3$  needs at least  $\Omega(n/\log n)$  rounds.*

### 3.4 Induced subgraph detection for bounded treewidth patterns

Finally, we consider subgraph and induced subgraph detection for pattern graphs of low treewidth. Recall that in centralized setting, a subgraph  $H$  with treewidth  $t$  can be detected in time  $2^{O(k)}n^{t+1}\log n$  [5], implying that detecting constant-treewidth subgraphs is fixed-parameter tractable. However, in CONGEST model, turns out that pattern of treewidth 2 are already maximally hard.

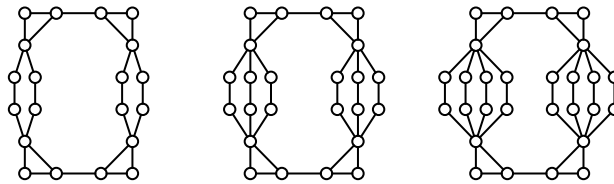
Our construction for the hard pattern graph uses similar ideas as the hard non-induced subgraph detection instances presented by Fischer et al. [28]. However, the pattern graphs they use a fairly dense and have treewidth higher than 2.

► **Theorem 13.** For any  $k \geq 2$ , there exists a pattern graph  $H_k$  of treewidth 2 such that CONGEST algorithm solving either  $H_k$ -detection or induced  $H_k$ -detection needs at least  $\Omega(n^{2-1/k})$  rounds.

Let  $k \geq 2$  be fixed. We construct the graph  $H_k$  as follows:

1. We start with four triangles  $A_1, A_2, B_1$  and  $B_2$  with nodes labelled by 1, 2 and 3.
2. Nodes 1 of  $A_1$  and  $A_2$  are connected by an edge, and nodes 1 of  $B_1$  and  $B_2$  are connected by an edge.
3. Nodes 2 of  $A_1$  and  $B_1$  are connected with  $k$  disjoint paths of length 3. Likewise, Nodes 2 of  $A_2$  and  $B_2$  are connected with  $k$  disjoint paths of length 3.

The graph  $H_k$  is a *series-parallel* graph, and thus has treewidth 2 [12].



► **Lemma 14.** Let  $k \geq 2$  be fixed. There exists a family of lower bound graphs for  $H_k$ -detection and induced  $H_k$ -detection with  $f_n = \text{DISJ}_{\Theta(n^2)}$  and  $C(n) = \Theta(n^{1/k})$ . (► See full version.)

Theorem 13 now follows immediately by Theorem 2.

## 4 Multicolored problems

In the *multicolored (induced) subgraph detection*, we are given a pattern graph  $H$  on  $k$  nodes and an input graph  $G$  with a (not necessarily proper)  $k$ -coloring, and the task is to find a (induced) copy of  $H$  that is multicolored, i.e. a copy where all nodes have different colors.

### 4.1 Reductions

We first prove that the complexities of multicolored  $k$ -clique and  $k$ -independent set are closely related to their standard versions also in the distributed setting. These results follow from standard fixed-parameter reductions [40, 27].

► **Theorem 15.** If multicolored  $k$ -clique can be solved in  $T(n)$  rounds in CONGEST, then  $k$ -clique can be solved  $O(k^2T(kn))$  rounds in CONGEST. If  $k$ -clique can be solved in  $T(n)$  rounds in CONGEST, then multicolored  $k$ -clique can be solved  $T(n)$  rounds in CONGEST. (► See full version.)

In the centralized setting, clique and independent set are equivalent, so the above reductions work also for independent set. However, in the distributed setting, only one direction works immediately, by essentially the same proof.

► **Theorem 16.** If multicolored  $k$ -independent set can be solved in  $T(n)$  rounds in CONGEST, then  $k$ -independent set can be solved  $O(k^2T(kn))$  rounds in CONGEST.

## 4.2 Lower bounds

Next, we prove some simple unconditional lower bounds for multicolored (induced) cycle detection and multicolored induced path detection.

► **Theorem 17.** *For any  $k \geq 4$ , any CONGEST algorithm solving multicolored (induced)  $k$ -cycle detection needs at least  $\Omega(n/\log n)$  rounds. (▷ See full version.)*

► **Theorem 18.** *For any  $k \geq 6$ , any CONGEST algorithm solving multicolored induced  $k$ -edge path detection needs at least  $\Omega(n/\log n)$  rounds. (▷ See full version.)*

## 5 Induced subgraph detection on bounded degeneracy graphs

### 5.1 Induced tree detection

We start by giving a parameterized distributed algorithm for detecting induced trees, parameterized by the degeneracy  $d = d(G)$  of the input graph. This result is based on the *random separation* algorithm of Cai et al. [13], adapted to distributed setting. For this result, we assume for convenience that all nodes are given the parameter  $d$  as input; we discuss at the end how to remove this dependence for the randomized version of the algorithm.

**Preliminaries.** Let  $G = (V, E)$  be a graph. We say that an orientation  $\sigma$  of the edges of  $G$  is an  $\alpha$ -bounded orientation, or simply  $\alpha$ -orientation, if every node  $v \in V$  has out-degree at most  $\alpha$  in  $\sigma$ , and  $\sigma$  is acyclic. A graph  $G$  is  $d$ -degenerate if and only if has an  $d$ -orientation; moreover, an  $O(d)$ -orientation can be computed fast in the CONGEST model:

► **Lemma 19** ([7]). *Let  $G$  be a  $d$ -degenerate graph, and let  $\varepsilon > 0$ . We can compute a  $(2 + \varepsilon)d$ -orientation of  $G$  in  $O(\log n)$  rounds in the CONGEST model, assuming  $d$  is known to all nodes. If  $d$  is not known, we instead can compute a  $(4 + \varepsilon)d$ -orientation of  $G$  in  $O(\log n)$  rounds.*

**Multicolored induced trees with orientation.** Let  $T$  be a tree on  $k$  nodes. We first to show how to solve a specific multicolored version of induced  $T$ -detection, given an acyclic orientation of  $G$  as input.

More precisely, let the graph  $G$ , let  $\sigma$  be an  $\alpha$ -bounded orientation of  $G$ , and let  $\chi: V \rightarrow \{0, 1, \dots, k\}$  be a (not necessarily proper)  $(k + 1)$ -coloring of  $G$ . Moreover, assume that the tree  $T$  is labelled in a bottom-up manner with  $1, 2, \dots, k$  with an arbitrary node as a root – that is, the root has label  $k$ , and each node has a smaller label than their parent. We say that an induced copy  $H$  of  $T$  in  $G$  is *proper* w.r.t  $\sigma$  and  $\chi$  if the node in  $H$  corresponding to node  $i$  in  $T$  has color  $i$ , and every node that is an out-neighbor of some node in  $H$  has color 0.

► **Lemma 20.** *Given a graph  $G = (V, E)$ , an orientation  $\sigma$  of  $G$ , and a coloring  $\chi$  as input, we can find a proper induced copy of a tree  $T$  in  $O(k)$  rounds using  $O(1)$ -bit messages in CONGEST model. (▷ See full version.)*

**Induced trees.** Using Lemma 20 as a subroutine, we now show how to detect induced copies of any tree  $T$ . We use random separation [13] and color-coding [5] techniques to reduce the general problem to detection of proper induced copies of  $T$ .

► **Theorem 21.** *Finding induced copy of a tree  $T$  on  $k$  nodes in a  $d$ -degenerate graph  $G$  can be done in  $k2^{O(dk)}k^k + O(\log n)$  rounds in the CONGEST model using a randomized algorithm. (▷ See full version.)*



**Derandomization.** Finally, we note that the algorithms can be derandomized using standard derandomization tools from fixed-parameter algorithms. Specifically, we use the derandomization of Alon and Gutner [4] to avoid incurring extra  $O(\log n)$  factor that would follow from the original derandomization of Cai et al. [13].

► **Theorem 22.** *Finding induced copy of a tree  $T$  on  $k$  nodes in a  $d$ -degenerate graph  $G$  can be done in  $f(d, k) + O(\log n)$  rounds in the CONGEST model using a deterministic algorithm for some function  $f$ , assuming  $d$  is known to all nodes.* (▷ See full version.)

**Unknown degeneracy.** The only part where the randomized algorithm uses the knowledge of  $d(G)$  is for deciding how many repeats of the random coloring it performs; Lemma 19 can be used without knowing  $d(G)$ . Without knowledge of  $d(G)$ , nodes can determine the largest out-degree in orientation  $\sigma$  in their radius- $k$  neighborhood and use that as a proxy for  $d(G)$  to determine how many repeats of the random coloring they should participate in; it is easy to verify that this still retains the correctness of the algorithm. The only caveat is that different nodes can terminate at different times, and cannot determine when all nodes have terminated.

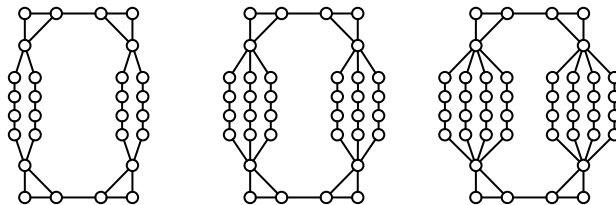
The deterministic algorithm, on the other hand, requires that all nodes know the degeneracy  $d(G)$ , or the same upper bound for this value. While we can compute an  $O(\kappa(G))$ -orientation  $\sigma$  for  $G$  in  $O(\log n)$  rounds, all nodes do not necessarily learn the largest out-degree in  $\sigma$ ; indeed, one can trivially see that having all nodes learn  $d(G)$  requires  $\Omega(D)$  rounds in the worst case.

## 5.2 Induced subgraph detection for bounded treewidth patterns

We now show that with slight modification, the hard treewidth 2 patterns presented in Section 3.4 can be adapted to bounded degeneracy setting. Recall that as mentioned in the introduction, any pattern graph on  $k$  nodes can be detected in  $O(kd(G)n)$  rounds by having all nodes gather full information about their distance- $k$  neighborhood; thus, the following lower bound is almost tight.

► **Theorem 23.** *For any  $k \geq 2$ , there exists a pattern graph  $H_k$  of treewidth 2 such that CONGEST algorithm solving either  $H$ -detection or induced  $H$ -detection on graphs of degeneracy 2 needs at least  $\Omega(n^{1-1/k})$  rounds.*

We use the same construction for  $k \geq 2$  for the pattern graph as in Lemma 13, but add paths of length 5 instead of paths of length 3 between triangles  $A_1$  and  $B_1$ , and triangles  $A_2$  and  $B_2$ . Let us denote the resulting graph by  $H'_k$ .



► **Lemma 24.** *Let  $k \geq 2$  be fixed. There exists a family of lower bound graphs of degeneracy 2 for  $H'_k$ -detection and induced  $H'_k$ -detection with  $f_n = \text{DISJ}_{\Theta(n)}$  and  $C(n) = \Theta(n^{1/k})$ .*

(▷ See full version.)

## 6 Bounded vertex cover number and MCIS

Finally, we consider induced subgraph detection parameterized by vertex cover number  $\tau(G)$ . Specifically, we show that a more general problem of *maximum common induced subgraph (MCIS)* can be solved in constant rounds on graphs of constant vertex cover number, which implies our results for induced subgraph detection.

**Maximum common induced subgraph.** In the centralized version of maximum common induced subgraph, we are given graphs  $G = (V_G, E_G)$  and  $H = (V_H, E_H)$  as input, and the task is to find the maximum-size graph  $G^*$  such that  $G^*$  appears as induced subgraph of both  $G$  and  $H$ . More precisely, the output should be a function  $f: V_G \rightarrow V_H \cup \{\perp\}$  such that for the set  $U_G = \{v \in V_G: f(v) \neq \perp\}$ , the function  $f$  restricted to  $U_G$  is an isomorphism between  $G[U_G]$  and  $H[f(U_G)]$ .

In this section, we consider MCIS parameterized by the sum of the vertex cover numbers  $\tau(G) + \tau(H)$ . Note that when  $H$  is complete graph and  $|G| = |H|$ , the problem is equivalent to maximum clique and hence NP-hard. It is W[1]-hard when parameterized by the solution size  $k$ , and W[1]-hard parameterized by the size of a minimum vertex cover of only one of the input graphs, even when restricted to bipartite graphs (see e.g. [2, 3] for more discussion).

**Distributed MCIS.** In the distributed version of the MCIS problem, the input graph  $G = (V_G, E_G)$  is the communication network, and full information about the second input graph  $H = (V_H, E_H)$  is given to every node as local input. Each node  $v$  needs to give a local output  $f(v) \in V_H \cup \{\perp\}$  such that the global function  $f$  satisfies the conditions of MCIS solution.

► **Theorem 25.** *Solving the maximum common induced subgraph problem on communication graph  $G$  and target graph  $H$  can be done in  $2^{O(\tau^2)}$  rounds in the CONGEST model deterministically, where  $\tau = \max(\tau(G), \tau(H))$ . (▷ See full version.)*

**Induced subgraph detection on bounded vertex cover number graphs.** As an immediate consequence of the MCIS algorithm, we obtain a parameterized distributed algorithm for detecting an induced copy of  $H$ , for any pattern graph  $H$ , as a graph  $H$  on  $k$  nodes has vertex cover number at most  $k$ .

► **Theorem 26.** *Let  $H$  be a pattern graph on  $k$  nodes. Finding induced copy  $H$  can be done in  $2^{O((\tau(G)+k)^2)}$  rounds in the CONGEST model deterministically.*

## 7 Conclusions and open problems

A central takeaway of this work is that centralized parameterized complexity offers both algorithmic techniques and perspectives for distributed computing. In particular, we believe that the study of structural graph parameters is a valuable paradigm for understanding sparse and structured networks in general. However, we note that there still remain open research directions related to topics studied in this paper:

- In terms of immediate open questions left by our work, we note that we currently do not have any systematic results on separation between the hardness of induced and non-induced subgraph detection for a given pattern  $H$ . For example, the induced cycle detection lower bound of Le Gall and Miyamoto [36] gives a near-linear – or super-linear, in case of even cycles – gap between induced and non-induced cycle detection, but it would be interesting to explore similar results for other pattern graphs in systematic fashion.

- More generally, we do not understand the complexity of subgraph detection type problems in distributed setting as well as in the centralized setting. For example, the complexity of  $k$ -independent set detection in CONGEST remains open, whereas in the centralized setting, it is equivalent to  $k$ -clique – a correspondence that does not hold in CONGEST.
- Besides degeneracy and vertex cover number, there are many other structural graph parameters commonly studied in parameterized complexity – for example, feedback vertex and edge sets, treewidth and pathwidth. Whereas Li [37] provides a framework for using treewidth for global optimization problems, it does not directly imply results for local problems such as subgraph detection; one might expect that considering something akin to *local* treewidth of a graph would be more appropriate for local graph problems. A secondary question is understanding what structural graph parameters are relevant from the perspective of real-world networks.

---

### References

- 1 Amir Abboud, Keren Censor-Hillel, and Seri Khoury. Near-linear lower bounds for distributed distance computations, even in sparse networks. In *Proc. 30th International Symposium on Distributed Computing (DISC 2016)*, 2016. doi:10.1007/978-3-662-53426-7\_3.
- 2 Faisal N. Abu-Khzam. Maximum common induced subgraph parameterized by vertex cover. *Information Processing Letters*, 114(3):99–103, 2014. doi:10.1016/j.ipl.2013.11.007.
- 3 Faisal N. Abu-Khzam, Édouard Bonnet, and Florian Sikora. On the complexity of various parameterizations of common induced subgraph isomorphism. *Theoretical Computer Science*, 697:69–78, 2017. doi:10.1016/j.tcs.2017.07.010.
- 4 Noga Alon and Shai Gutner. Linear time algorithms for finding a dominating set of fixed size in degenerated graphs. In *Proc. 13th International Computing and Combinatorics Conference (COCOON 2007)*, pages 394–405. Springer, 2007.
- 5 Noga Alon, Raphael Yuster, and Uri Zwick. Color-coding. *Journal of the ACM*, 42(4):844–856, 1995. doi:10.1145/210332.210337.
- 6 Nir Bacrach, Keren Censor-Hillel, Michal Dory, Yuval Efron, Dean Leitersdorf, and Ami Paz. Hardness of distributed optimization. In *Proc. ACM Symposium on Principles of Distributed Computing (PODC 2019)*, pages 238–247, 2019.
- 7 Leonid Barenboim and Michael Elkin. Sublogarithmic distributed MIS algorithm for sparse graphs using Nash-Williams decomposition. *Distributed Computing*, 22(5–6):363–379, 2010. doi:10.1007/s00446-009-0088-2.
- 8 Leonid Barenboim and Michael Elkin. *Distributed Graph Coloring: Fundamentals and Recent Developments*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool, 2013. doi:10.2200/S00520ED1V01Y201307DCT011.
- 9 Leonid Barenboim and Victor Khazanov. Distributed symmetry-breaking algorithms for congested cliques. In *International Computer Science Symposium in Russia*, pages 41–52. Springer, 2018.
- 10 Ran Ben-Basat, Ken ichi Kawarabayashi, and Gregory Schwartzman. Parameterized Distributed Algorithms. In *Proc. 33rd International Symposium on Distributed Computing (DISC 2019)*, 2019. doi:10.4230/LIPIcs.DISC.2019.6.
- 11 Andreas Björklund, Thore Husfeldt, Petteri Kaski, and Mikko Koivisto. Narrow sieves for parameterized paths and packings. *Journal of Computer and System Sciences*, 87:119–139, 2017. doi:10.1016/j.jcss.2017.03.003.
- 12 Andreas Brandstädt, Van Bang Le, and Jeremy P. Spinrad. *Graph classes: a survey*. SIAM Monographs on Discrete Mathematics and Applications. Society for Industrial and Applied Mathematics, 1999.
- 13 Leizhen Cai, Siu Man Chan, and Siu On Chan. Random separation: A new method for solving fixed-cardinality optimization problems. In *Proc. 2nd International Workshop on Parameterized and Exact Computation (IWPEC 2006)*, pages 239–250. Springer, 2006.

## 15:16 Beyond Distributed Subgraph Detection

- 14 Keren Censor-Hillel, Yi-Jun Chang, François Le Gall, and Dean Leitersdorf. Tight distributed listing of cliques. In *Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA 2021)*, 2021.
- 15 Keren Censor-Hillel, Orr Fischer, Tzlil Gonen, François Le Gall, Dean Leitersdorf, and Rotem Oshman. Fast Distributed Algorithms for Girth, Cycles and Small Subgraphs. In Hagit Attiya, editor, *Proc. 34th International Symposium on Distributed Computing (DISC 2020)*, volume 179, 2020. doi:10.4230/LIPIcs.DISC.2020.33.
- 16 Keren Censor-Hillel, Petteri Kaski, Janne H. Korhonen, Christoph Lenzen, Ami Paz, and Jukka Suomela. Algebraic methods in the congested clique. In *Proc. ACM Symposium on Principles of Distributed Computing (PODC 2015)*, pages 143–152, 2015.
- 17 Keren Censor-Hillel, Seri Khoury, and Ami Paz. Quadratic and near-quadratic lower bounds for the CONGEST model. In *Proc. 31st International Symposium on Distributed Computing (DISC 2017)*, 2017. doi:10.4230/LIPIcs.DISC.2017.10.
- 18 Yi-Jun Chang and Thatchaphol Saranurak. Improved distributed expander decomposition and nearly optimal triangle enumeration. In *Proc. ACM Symposium on Principles of Distributed Computing (PODC 2019)*, pages 66–73, 2019.
- 19 Yijia Chen and Jorg Flum. On parameterized path and chordless path problems. In *Proc. 22nd Annual IEEE Conference on Computational Complexity (CCC 2007)*, pages 250–263, 2007.
- 20 Marek Cygan, Fedor V Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. *Parameterized algorithms*. Springer, 2015. doi:10.1007/978-3-319-21275-3.
- 21 Artur Czumaj and Christian Konrad. Detecting cliques in congest networks. *Distributed Computing*, 2019. doi:10.1007/s00446-019-00368-w.
- 22 Mina Dalirrooyfard, Thuy Duong Vuong, and Virginia Vassilevska Williams. Graph pattern detection: Hardness for all induced patterns and faster non-induced cycles. In *Proc. 51st Annual ACM SIGACT Symposium on Theory of Computing (STOC 2019)*, pages 1167–1178, 2019. doi:10.1145/3313276.3316329.
- 23 Danny Dolev, Christoph Lenzen, and Shir Peled. “Tri, tri again”: Finding triangles and small subgraphs in a distributed setting. In *Proc. 26th International Symposium on Distributed Computing (DISC 2012)*, pages 195–209, 2012. doi:10.1007/978-3-642-33651-5\_14.
- 24 Rod G. Downey and Michael R. Fellows. Fixed-parameter tractability and completeness II: On completeness for W[1]. *Theoretical Computer Science*, 141(1):109–131, 1995. doi:10.1016/0304-3975(94)00097-3.
- 25 Andrew Drucker, Fabian Kuhn, and Rotem Oshman. On the power of the congested clique model. In *Proc. 33rd ACM Symposium on Principles of Distributed Computing (PODC 2014)*, pages 367–376, 2014. doi:10.1145/2611462.2611493.
- 26 Guy Even, Orr Fischer, Pierre Fraigniaud, Tzlil Gonen, Reut Levi, Moti Medina, Dennis Olivetti Pedro Montealegre, Rotem Oshman, Ivan Rapaport, and Ioan Todinca. Three notes on distributed property testing. In *Proc. 31st International Symposium on Distributed Computing (DISC 2017)*, 2017.
- 27 Michael R. Fellows, Danny Hermelin, Frances Rosamond, and Stéphane Vialette. On the parameterized complexity of multiple-interval graph problems. *Theoretical Computer Science*, 410(1):53–61, 2009. doi:10.1016/j.tcs.2008.09.065.
- 28 Orr Fischer, Tzlil Gonen, Fabian Kuhn, and Rotem Oshman. Possibilities and impossibilities for distributed subgraph detection. In *Proc. 30th Symposium on Parallelism in Algorithms and Architectures (SPAA 2018)*, pages 153–162, 2018.
- 29 Pierre Fraigniaud and Dennis Olivetti. Distributed detection of cycles. In *Proc. 29th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2017)*, pages 153–162, 2017. doi:10.1145/3087556.3087571.

- 30 Silvio Frischknecht, Stephan Holzer, and Roger Wattenhofer. Networks cannot compute their diameter in sublinear time. In *Proc. 23rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2012)*, 2012.
- 31 Mohsen Ghaffari and Ali Sayyadi. Distributed arboricity-dependent graph coloring via all-to-all communication. In *Proc. 46th International Colloquium on Automata, Languages, and Programming (ICALP 2019)*, 2019.
- 32 Janne H. Korhonen and Amir Nikabadi. Beyond distributed subgraph detection: Induced subgraphs, multicolored problems and graph parameters, 2021. [arXiv:2109.06561](https://arxiv.org/abs/2109.06561).
- 33 Janne H. Korhonen and Joel Rybicki. Deterministic subgraph detection in broadcast CONGEST. In *Proc. 21st International Conference on Principles of Distributed Systems (OPODIS 2017)*, 2017. doi:10.4230/LIPIcs.OPODIS.2017.4.
- 34 Janne H. Korhonen and Jukka Suomela. Towards a complexity theory for the congested clique. In *Proc 30th on Symposium on Parallelism in Algorithms and Architectures (SPAA 2018)*, pages 163–172, 2018.
- 35 Eyal Kushilevitz and Noam Nisan. *Communication complexity*. Cambridge University Press, 1997.
- 36 François Le Gall and Masayuki Miyamoto. Lower bounds for induced cycle detection in distributed computing. In *Proc. 32nd International Symposium on Algorithms and Computation (ISAAC 2021)*, 2021. [arXiv:2110.00741](https://arxiv.org/abs/2110.00741).
- 37 Jason Li. Distributed treewidth computation, 2018. [arXiv:1805.10708](https://arxiv.org/abs/1805.10708).
- 38 Burkhard Monien. How to find long paths efficiently. *North-Holland Mathematics Studies*, 109:239–254, 1985.
- 39 Jaroslav Nešetřil and Svatopluk Poljak. On the complexity of the subgraph problem. *Commentationes Mathematicae Universitatis Carolinae*, 26(2):415–419, 1985.
- 40 Krzysztof Pietrzak. On the parameterized complexity of the fixed alphabet shortest common supersequence and longest common subsequence problems. *Journal of Computer and System Sciences*, 67(4):757–771, 2003. doi:10.1016/S0022-0000(03)00078-3.
- 41 Alexander A. Razborov. On the distributional complexity of disjointness. *Theoretical Computer Science*, 106(2):385–390, 1992. doi:10.1016/0304-3975(92)90260-M.
- 42 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*, 41:1235–1265, 2012. doi:10.1137/11085178X.
- 43 Sebastian Siebertz and Alexandre Vigny. Parameterized distributed complexity theory: A logical approach. *arXiv preprint*, 2019. [arXiv:1903.00505](https://arxiv.org/abs/1903.00505).
- 44 Virginia Vassilevska. *Efficient algorithms for path problems in weighted graphs*. PhD thesis, Carnegie Mellon University, 2008.
- 45 Ryan Williams. Finding paths of length  $k$  in  $O^*(2^k)$  time. *Information Processing Letters*, 109(6):315–318, 2009.
- 46 Raphael Yuster and Uri Zwick. Finding even cycles even faster. *SIAM Journal on Discrete Mathematics*, 10(2):209–222, 1997.

## **A** Induced short paths

Induced paths with two edges can be detected in  $O(1)$  rounds, in contrast to the situation with e.g. triangle detection. The proof follows the centralized algorithm of Vassilevska [44].

► **Theorem 27.** *Given a graph  $G$  on  $n$  nodes, detecting an induced path of length 2 on  $G$  can be done in  $O(1)$  rounds in the broadcast CONGEST model.*

## 15:18 Beyond Distributed Subgraph Detection

**Proof.** As the first step, we assign a label  $\ell(v)$  for each node as follows. First, each node  $v \in V$  broadcast its identifier to all its neighbors  $N(v)$ , and then each node  $v$  picks the label  $\ell(v)$  to be the smallest identifier from the set that it received, or its own identifier if that is smaller. The nodes then broadcast their label  $\ell(v)$  and their degree  $\deg(v)$  to all their neighbors.

Each node  $v$  then checks the following conditions, and reports that induced 2-path exists if at least one of them is satisfied:


1. There exists a neighbor  $u \in N(v)$  with  $\deg(u) \neq \deg(v)$ .
2. There exists neighbors  $u, w \in N(v)$  with  $\ell(u) \neq \ell(w)$ .

For the correctness of the algorithm, we first observe that a graph does not contain an induced 2-path if and only if each connected component is a clique. If none of the nodes report an induced 2-path, then by conditions (a) and (b), each connected component is clique. Likewise, if  $G$  consists of disjoint cliques, no node will report an induced 2-path. Finally, we note that the algorithm takes 3 rounds in CONGEST. ◀

# An Improved Random Shift Algorithm for Spanners and Low Diameter Decompositions

Sebastian Forster  

University of Salzburg, Austria

Martin Grösbacher 

University of Salzburg, Austria

Tijn de Vos  

University of Salzburg, Austria

---

## Abstract

Spanners have been shown to be a powerful tool in graph algorithms. Many spanner constructions use a certain type of clustering at their core, where each cluster has small diameter and there are relatively few spanner edges between clusters. In this paper, we provide a clustering algorithm that, given  $k \geq 2$ , can be used to compute a spanner of stretch  $2k - 1$  and expected size  $O(n^{1+1/k})$  in  $k$  rounds in the CONGEST model. This improves upon the state of the art (by Elkin, and Neiman [TALG'19]) by making the bounds on both running time and stretch independent of the random choices of the algorithm, whereas they only hold with high probability in previous results. Spanners are used in certain synchronizers, thus our improvement directly carries over to such synchronizers. Furthermore, for keeping the *total* number of inter-cluster edges small in low diameter decompositions, our clustering algorithm provides the following guarantees. Given  $\beta \in (0, 1]$ , we compute a low diameter decomposition with diameter bound  $O\left(\frac{\log n}{\beta}\right)$  such that each edge  $e \in E$  is an inter-cluster edge with probability at most  $\beta \cdot w(e)$  in  $O\left(\frac{\log n}{\beta}\right)$  rounds in the CONGEST model. Again, this improves upon the state of the art (by Miller, Peng, and Xu [SPAA'13]) by making the bounds on both running time and diameter independent of the random choices of the algorithm, whereas they only hold with high probability in previous results.

**2012 ACM Subject Classification** Theory of computation  $\rightarrow$  Sparsification and spanners; Theory of computation  $\rightarrow$  Distributed algorithms

**Keywords and phrases** Spanner, low diameter decomposition, synchronizer, distributed graph algorithms

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2021.16

**Related Version** *Full Version:* <https://arxiv.org/abs/2111.08975>

**Funding** Supported by the Austrian Science Fund (FWF): P 32863-N.

**Acknowledgements** The first author would like to thank Merav Parter for mentioning that there was still some room for improvement in the spanner construction.

## 1 Introduction

Clustering has become an essential tool in dealing with large data sets. The goal of clustering data is to identify disjoint, dense regions such that the space between them is sparse. When working with graphs, this translates to partitioning the vertex set into clusters with relatively few edges between clusters such that the clusters satisfy a particular property. One can for example demand that the clusters have low diameter [4, 2, 9, 31], high conductance [22, 24, 36, 12, 13, 35, 14], or low effective resistance diameter [1]. In this paper, we focus on the low diameter decomposition and its connection to spanners. Low diameter decompositions are formally defined as follows.



© Sebastian Forster, Martin Grösbacher, and Tijn de Vos;  
licensed under Creative Commons License CC-BY 4.0

25th International Conference on Principles of Distributed Systems (OPODIS 2021).

Editors: Quentin Bramas, Vincent Gramoli, and Alessia Milani, Article No. 16; pp. 16:1–16:17

Leibniz International Proceedings in Informatics



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

## 16:2 Spanners and Low Diameter Decompositions

► **Definition 1.** Let  $G = (V, E)$  be a weighted graph. A probabilistic  $(\beta, \delta)$ -low diameter decomposition of  $G$  is a partition of the vertex set  $V$  into subsets  $V_1, \dots, V_l$ , called clusters, such that

- each cluster  $V_i$  has strong diameter at most  $\delta$ , i.e.,  $d_{G[V_i]}(u, v) \leq \delta$  for all  $u, v \in V_i$ <sup>1</sup>;
- the probability that an edge  $e \in E$  is an inter-cluster edge is at most  $\beta \cdot w(e)$ , i.e., for  $e = (u, v)$ , the probability that  $u \in V_i$  and  $v \in V_j$  for  $i \neq j$  is at most  $\beta \cdot w(u, v)$ .

In an unweighted graph, another typical definition of the low diameter decomposition replaces the second condition with an upper bound on the number of inter-cluster edges [31]. In this fashion, a probabilistic low diameter decomposition has  $O(\beta m)$  inter-cluster edges in expectation.

Originally, low diameter decompositions were developed for distributed models, where they have been proven useful by reducing communication significantly in certain situations [4, 6]. Later, they also have shown to be fruitful in other models; they have been applied in shortest path approximations [15], cut sparsifiers [27], and tree embeddings with low stretch [2, 9, 10].

The clustering technique used for computing low diameter decompositions has implicitly been used to develop sparse spanners [11, 30, 17] and synchronizers [4, 7]. The main idea is to create the clusters, and add some, but not all, of the inter-cluster edges. In a sense, the inter-cluster edges are sparsified. We formalize this concept as follows.

► **Definition 2.** Let  $G = (V, E)$  be an unweighted graph. A sparsified  $(\zeta, \delta)$ -low diameter decomposition of  $G$  is a partition of the vertex set  $V$  into subsets  $V_1, \dots, V_l$ , called clusters, together with a set of edges  $F \subseteq E$  such that

- each cluster  $V_i$  has strong diameter at most  $\delta$ , i.e.,  $d_{G[V_i]}(u, v) \leq \delta$  for all  $u, v \in V_i$ ;
- for every edge, one of its endpoints has an edge from  $F$  into the cluster of the other endpoint, i.e.,  $\forall e = (u, v) \in E$ , we have either  $(u', v) \in F$  for some  $u' \in C_u$  or  $(u, v') \in F$  for some  $v' \in C_v$ <sup>2</sup>.
- $|F| \leq \zeta$ .

Moreover, we say that a sparsified  $(\zeta, \delta)$ -low diameter decomposition is tree-supported if for each cluster  $V_i$  we have a cluster center  $c_i \in V_i$  and a tree of height at most  $\delta/2$  spanning the cluster. All these trees together are called the support-forest.

Our main result is a clustering algorithm that produces a sparsified low diameter decomposition.

► **Theorem 3.** There exists an algorithm, such that for each unweighted graph  $G = (V, E)$  and parameter  $k \geq 2$  it outputs a tree-supported sparsified  $(\zeta, 2k - 2)$ -low diameter decomposition, with  $\zeta = O(n^{1+1/k})$  in expectation. The algorithm runs in  $k$  rounds in the CONGEST model, and in  $O(k \log^* n)$  depth and  $O(m)$  work in the PRAM model.

An important feature of this result is that the bounds on the strong diameter and number of rounds are not probabilistic; they are independent of the random choices in the algorithm. We show two applications of this theorem: constructing spanners and constructing synchronizers.

<sup>1</sup> For  $U \subseteq V$ , we write  $G[U]$  for the graph induced by  $U$ , i.e.,  $G[U] := (U, U \times U \cap E)$ .

<sup>2</sup> For  $v \in V$ , we write  $C_v$  for the cluster containing  $v$ .



**Spanners.** Given a graph  $G = (V, E)$ , we say that  $H \subseteq G$  is a *spanner* of stretch  $\alpha$ , if  $d_H(u, v) \leq \alpha \cdot d_G(u, v)$ , for every  $u, v \in V$ . It is straightforward that a tree-supported sparsified  $(\zeta, 2k - 2)$ -low diameter decomposition gives a spanner of size  $\zeta + n$  and stretch  $2k - 1$ , for details we refer to Section 3.1. This gives us the following corollary.

► **Corollary 4.** *There exists an algorithm, such that for each unweighted graph  $G = (V, E)$  and parameter  $k \geq 2$  it outputs a spanner  $H$  of stretch  $2k - 1$ . The expected size of  $H$  is at most  $O(n^{1+1/k})$ . The algorithm runs in  $k$  rounds in the CONGEST model, and in  $O(k \log^* n)$  depth and  $O(m)$  work in the PRAM model.*

Spanners themselves have been useful in computing approximate shortest paths [5, 16], distance oracles and labeling schemes [38, 32], and routing [33]. A simple greedy algorithm [3] gives a spanner of stretch  $2k - 1$  and of size  $O(n^{1+1/k})$ , which is an optimal trade-off under the girth conjecture [18]. However, its fastest known implementation in the RAM model takes  $O(kn^{2+1/(2k+1)})$  time [34]. Halperin and Zwick [23] gave a linear-time algorithm to construct spanners with an optimal trade-off for unweighted graphs in the RAM model. However, this algorithm does not adapt well to distributed and parallel models of computation. This problem can be overcome by exploiting the aforementioned relation with sparsified low diameter decompositions. This was (implicitly) done by Baswana and Sen [11], who provide an algorithm that computes a spanner of stretch  $2k - 1$  and of size  $O(kn^{1+1/k})$  in  $O(k)$  rounds. The state of the art is by Elkin and Neiman [17], which builds off [30], and is also based on low diameter decompositions. They provide an algorithm that with probability  $1 - 1/c$  computes a  $(2k - 1)$ -spanner of expected size  $O(c^{1/k}n^{1+1/k})$  in  $k$  rounds. Standard techniques of boosting the failure probability to something inverse polynomial (or “with high probability”) will require a logarithmic overhead. Alternatively, one can view the algorithm of Elkin and Neiman as an algorithm that outputs an  $\alpha$ -spanner of expected size  $O(c^{1/k}n^{1+1/k})$  in  $O(\alpha)$  rounds, such that with probability  $1 - 1/c$  we have that  $\alpha = 2k - 1$ .

Corollary 4 improves on the result of Elkin and Neiman by making the bounds on the stretch and the running time independent of the random choices in the algorithm. In particular, the algorithm of Elkin-Neiman involves sampling vertex values from an exponential distribution. The exponential distribution introduces an (as we show) unnecessary amount of randomness; we demonstrate that the geometric distribution suffices. We replace the extra random bits the exponential distribution provides by a tie-breaking rule on the vertex IDs, which we believe contributes to a more intuitive construction.

**Synchronizers.** The second application of Theorem 3 is in constructing synchronizers in the CONGEST model. A synchronizer gives a procedure to run a synchronous algorithm on an asynchronous network. More precisely, the goal is to run any synchronous  $R(n)$ -round  $M(n)$ -message complexity CONGEST model algorithm on an asynchronous network with minimal time and message overhead. The first results on synchronizers are by Awerbuch [4], called synchronizers  $\alpha$ ,  $\beta$ , and  $\gamma$ . Subsequently, these results were improved by Awerbuch and Peleg [8], and Awerbuch et al. [7], both having  $O(R(n) \log^3 n)$  time and  $O(M(n) \log^3 n)$  message complexity.

The synchronizer  $\gamma$  from [4] essentially consists of running a combination of the simple synchronizers  $\alpha$  and  $\beta$  on a sparsified  $(\zeta, \delta)$ -low diameter decompositions. In that case, the bound  $\zeta$  on the sparsified inter-cluster edges goes into the bound for the communication overhead of the synchronizer and the strong diameter bound  $\delta$  goes into the bound for the time overhead of the synchronizer. Applying synchronizer  $\gamma$  on our clustering, we obtain the following result.

► **Theorem 5.** *There exists an algorithm that, given  $k \geq 2$ , can run any synchronous  $R(n)$ -round  $M(n)$ -message complexity CONGEST model algorithm on an asynchronous CONGEST network. In expectation, the algorithm uses a total of  $O(M(n) + R(n)n^{1+1/k})$  messages. Provided that each message incurs a delay of at most one time unit, it takes  $O(R(n)k)$  rounds. The initialization phase takes  $O(k)$  time, using  $O(km)$  messages.*

The running time claimed in this theorem is independent of the random choices in our algorithm, which is a direct result of Theorem 3. The previous sparsified low diameter decompositions (implicit in [17]) would provide similar bounds on the running time, but only with constant probability.

**Low Diameter Decompositions.** Perhaps unsurprisingly, we show that, with the right choice of parameters, our clustering algorithm can also compute *unsparsified* low diameter decompositions.

► **Theorem 6.** *There exists an algorithm, such that for each graph  $G = (V, E)$  with integer weights  $w: E \rightarrow \{1, \dots, W\}$  and parameter  $\beta \in (0, 1]$  it outputs a low diameter decomposition, whose components are clusters of strong diameter of at most  $O\left(\frac{\log n}{\beta}\right)$ . Moreover, each edge is an inter-cluster edge with probability at most  $\beta \cdot w(u, v)$ . The algorithm runs in  $O\left(\frac{\log n}{\beta}\right)$  rounds in the CONGEST model, and in  $O\left(\frac{\log n \log^* n}{\beta}\right)$  depth and  $O(m)$  work in the PRAM model.*

Similar to our spanner algorithm, the bounds on the running time and strong diameter hold independent of the random choices within the algorithm, as opposed to the previous state of the art [31], where they only hold with high probability. In the low diameter decomposition as discussed above, the trade-off between  $\beta$  and diameter bound  $O\left(\frac{\log n}{\beta}\right)$  is essentially optimal [9].

## Technical Overview

Our clustering algorithm follows an approach known as ball growing, related to the probabilistic partitions of [9, 10]. In a sequential setting, this consists of picking a vertex, and repeatedly adding the neighbors of the current vertices to the ball. This stops when a certain bound is reached, such as a bound on the diameter of the ball or on the number of edges between the current ball and the remainder of the graph. The algorithm repeats this procedure with the remainder of the graph until this is empty. Miller, Peng, and Xu [31] showed that this can be parallelized by letting each vertex create its own ball, but after a certain start time delay. In [31], this has been done by sampling the delays from the exponential distribution, which leads to the aforementioned probabilistic diameter guarantee, as the exponential distribution can take infinitely high values – albeit with small probability. Furthermore, multiple authors (see e.g. [19, 31]) argue that one can round the sampled values from the exponential distribution for most of the algorithm and solely use that the fractional values of the sampled value induce a random permutation of the nodes. In this paper, we show that even fewer random bits are needed: we do not require a random permutation of the nodes. We demonstrate that a tie-breaking rule based on the IDs is enough.

We sample with a capped-off geometric distribution, also used in [28, 7]. As opposed to the standard geometric distribution, the capped-off version can only take a finite number of values. We believe this leads to a more direct proof of the spanner algorithm of [17] and of the decomposition algorithm of [31]. Moreover, by making the sparsifier low diameter

decomposition explicit, the application to synchronizers is almost immediate. In the remainder of this paper, we will not think of the sampled values as start time delays, but as the distance to some conceptual source  $s$ , similar to the view in [31]. The rest of the clustering algorithm then consists of computing a shortest path tree rooted at  $s$ , which is easily calculated, both in the CONGEST and PRAM model. The clusters consists of the trees that remain when we disconnect the shortest path tree by removing the root  $s$ .

As an anonymous reviewer pointed out, in the case of low diameter decompositions, the algorithm of Miller et al. [31] admits an alternative approach. We can exploit the fact that the exponential delays are bounded with high probability. In case the delays exceed the bound, we could return a sub-optimal clustering, without any central communication. As this only happens with low probability, it does not impact the expected number of inter-cluster edges. Note however, that the spanner construction of Elkin and Neiman [17] is not in this high-probability regime, therefore this straightforward approach would not work. We additionally believe that, beyond the result itself, our algorithm provides a more streamlined view.

## 2 The Clustering Algorithm

Let  $G = (V, E)$  be a graph with integer weights  $w: E \rightarrow \{1, \dots, W\}$ . Let  $p \in (0, 1)$  and  $r \in \mathbb{N}$  be parameters, to be chosen according to the application of our algorithm. In the following, we provide an algorithm for computing a clustering, where the strong diameter of these clusters will be  $2r$ . In particular, we will show that each cluster is tree-supported by a tree of height  $r$ . The number of inter-cluster edges depends on both  $p$  and  $r$ , and can be bounded in two ways. The first approach, detailed in Section 3, shows we have a sparsified low diameter decomposition. Here, for each vertex we compute the expected number of edges in the sparsified set of inter-cluster edges, which gives a bound that does not depend on  $m$ , but only on  $n, p$  and  $r$ . The second approach, detailed in Section 4, shows we have a probabilistic low diameter decomposition, by computing the probability that any edge is an inter-cluster edge.

### 2.1 Construction

First we conceptually add a node  $s$  to the graph  $G$  to form the graph  $G'$ . The node  $s$  will function as an artificial source for a shortest path tree. Each vertex will have a distance to  $s$  in  $G'$  depending on some random offset. Hereto, each vertex samples a value  $\delta_v$  from the *capped exponential distribution*  $\text{GeomCap}(p, r)$ , defined by

$$\mathbb{P}[\text{GeomCap}(p, r) = i] = \begin{cases} p(1-p)^i & \text{if } 0 \leq i \leq r-1; \\ (1-p)^r & \text{if } i = r; \\ 0 & \text{else.} \end{cases}$$

This distribution corresponds to the model where we repeat at most  $r$  Bernoulli trials, and measure how many trials occur (strictly) before the first success, or whether there is no success in the first  $r$  trials. We check that  $\text{GeomCap}$  is indeed a probability distribution on  $\{0, 1, \dots, r\}$ :

$$\sum_{i=0}^r \mathbb{P}[\text{GeomCap}(p, r) = i] = \sum_{i=0}^{r-1} p(1-p)^i + (1-p)^r = p \frac{1 - (1-p)^r}{1 - (1-p)} + (1-p)^r = 1.$$

As the intuition suggests, GeomCap has a memoryless property as long as the cap is not reached, i.e.,  $\mathbb{P}[\text{GeomCap}(p, r) = i \mid \text{GeomCap}(p, r) \geq i] = p$  for  $i \leq r - 1$ . The proof is completely analogous to the proof of the memoryless property of the geometric distribution.

For each vertex  $v \in V$ , we conceptually add an edge  $(s, v)$  to  $G'$ , with weight  $w(s, v) := r - \delta_v$ . We define  $d^{(u)}(s, x) := w(s, u) + d_G(u, x)$ , which is the minimal path length, for a path from  $s$  to  $x$  over  $u$ . Now we have that the distance from  $s$  to  $x$  equals  $d_{G'}(s, x) = \min_{u \in V} \{d^{(u)}(s, x)\}$ . We call this the *level* of  $x$ , ranging from 0 (closest to  $s$ ) to  $r$  (furthest from  $s$ ). Moreover, we define  $p_u(x)$  to be the predecessor of  $x$  on an arbitrary but fixed shortest path from  $u$  to  $x$ . Next, we construct a shortest path tree  $T$  rooted at  $s$ . When necessary, we do tie-breaking according to IDs: let  $v$  be such that  $d_{G'}(s, x) = d^{(v)}(s, x)$  and  $\text{ID}(v) < \text{ID}(u)$  for all  $u \in V$  satisfying  $d_{G'}(s, x) = d^{(u)}(s, x)$ . Then we connect  $x$  to the shortest path tree using the edge  $(p_v(x), x)$ . Moreover, we add  $x$  to the cluster of  $v$  and write  $c_x := v$  for the corresponding cluster center. Intuitively, the clusters correspond to the connected components that remain when we remove the source  $s$  from the created shortest path tree. The formal argument for this can be found in the proof of Lemma 7. The computation of this shortest path tree is model-specific, we provide details in Section 2.3.

The algorithm outputs the shortest path tree  $T$ , and for each  $x \in V$ , the center of its cluster center  $c_x$  and its level. The knowledge of cluster centers immediately gives a clustering, where – by the remark above – each cluster has radius at most  $r$ . In Section 3, we show how to construct a set of edges  $F \subseteq E$  from the cluster centers and levels, such that  $H := T \cup F$  is a spanner.

In the above, we only need an arbitrary ordering of the vertices. If we assume that each vertex  $v \in V$  has a unique identifier,  $\text{ID}(v) \in \{1, \dots, N\}$ , we can provide an alternative way of constructing the same shortest path tree. We construct a graph where  $w(s, v) = r - \delta_v + \text{ID}(v)/(N + 1)$ , and compute a shortest path tree rooted at  $s$ . This embeds the tie-breaking rule in the weight of the added edges, and thus in the distances. For generality – and suitable implementation in distributed models with limited bandwidth – the remainder of this paper relies on the former characterization using the tie-breaking rule.

## 2.2 Tree-Support

Next, we will show that the created clusters are tree-supported by a tree of height  $r$ . We have already chosen cluster centers, and we will show that we can identify trees rooted at these centers that satisfy the tree-support condition.

► **Lemma 7.** *Each cluster is tree-supported by a tree of height at most  $r$ .*

**Proof.** Let  $v \in V$  be a vertex, which is part of the cluster centered at  $c_v$ . We show that there is a path from  $c_v$  to  $v$  contained in this cluster, which has length at most  $d_G(c_v, v) \leq r$ . We proceed to show by induction on  $d_G(c_v, v)$  that there is a path from  $c_v$  to  $v$  contained in their cluster, which has length at most  $d_G(c_v, v)$ . The base case,  $v = c_v$ , is trivial. Let  $u$  be the predecessor of  $v$  on some path from  $c_v$  to  $v$  of length  $d_G(c_v, v)$ . It suffices to show that  $u$  is in the same cluster, then the result follows from the induction hypothesis. By definition of  $c_v$ , we have that

$$d_{G'}(s, v) = d_{G'}(s, c_v) + d_G(c_v, v) = d_{G'}(s, c_v) + d_G(c_v, u) + w(u, v) = d^{(c_v)}(s, u) + w(u, v).$$

By the triangle inequality we have  $d_{G'}(s, v) \leq d_{G'}(s, u) + w(u, v)$ . Combining this, we see  $d^{(c_v)}(s, u) \leq d_{G'}(s, u)$ . As the distance  $d_{G'}(s, u)$  is minimal, by definition we have  $d^{(c_v)}(s, u) = d_{G'}(s, u)$ . Now suppose that  $u$  is part of some cluster  $c_u$ . Then we have  $d^{(c_u)}(s, u) = d_{G'}(s, u)$

and  $\text{ID}(c_u) \leq \text{ID}(c_v)$ . However, this implies that  $d^{(c_u)}(s, v) \leq d^{(c_u)}(s, u) + w(u, v) = d^{(c_v)}(s, u) + w(u, v) = d^{(c_v)}(s, v)$ . Hence by the tie-breaking for  $v$  we have  $\text{ID}(c_v) \leq \text{ID}(c_u)$  and thus  $c_u = c_v$ .  $\blacktriangleleft$

As an immediate corollary, we obtain a bound on the strong diameter of the clusters.

► **Corollary 8.** *Each cluster has a strong diameter of  $2r$ .*

## 2.3 Implementation and Running Time

For the RAM model, the implementation is straightforward and can be done in linear time [37]. The implementation in distributed and parallel models requires a little more attention. For both models, the computational aspect is very similar to prior work [31, 17].

### 2.3.1 Distributed Model

The algorithm as presented, can be implemented efficiently both in the LOCAL and in the CONGEST model. It runs in  $r + 1$  rounds as follows. In the initialization phase, each vertex  $v$  samples its value  $\delta_v$  and sets its initial distance to the conceptual vertex  $s$  as  $r - \delta_v$ . In the first round of communication,  $v$  sends the tuple  $(r - \delta_v, \text{ID}(v))$  to its neighbors. In each round,  $v$  updates its distance to  $s$  according to received messages. It then broadcasts the tuple of its updated distance and the ID corresponding to the first vertex on the path from  $s$  to  $v$ . Note that at the end of the algorithm, each node knows its own level and cluster center, and the level and cluster center of each of its neighbors.

When the algorithm is applied with  $r = O(n)$  (if  $r \geq n$ , we can simply return the connected components of the graph as clusters), we maintain a bound on the message size of  $O(\log n)$ , so there are no digit precision consideration for the CONGEST model. Moreover, each vertex  $v$  has distance at most  $r - \delta_v \leq r$  to  $s$ , the algorithm terminates within  $r + 1$  rounds.<sup>3</sup>

### 2.3.2 PRAM Model

The implementation in the PRAM model is slightly different to the CONGEST model. Instead of broadcasts by each vertex in each round, a vertex  $v$  updates its distance only once: either after one of its neighbors updated its distance, or after time  $r + 1 - \delta_v$  it sets its distance to  $r - \delta_v$ . The total required depth differs on the exact model of parallelism, it is  $O(r \log^* n)$  in the CRCW model of parallel computation. To show this, we follow the general lines of [25], but we have to be careful: during the shortest path computation, we might need to apply our tie-breaking rule, i.e., finding the minimum ID among all options. Note that in the PRAM model, we can assume without loss of generality that the IDs are labeled 1 to  $n$  in the adjacency list representation. Finding the minimum can be done with high probability in  $O(\log^* n)$  depth and  $O(n)$  work, as we can sort a list of integers between 1 and  $n$  in  $O(\log^* n)$  depth and  $O(n)$  work [21]. If we exceed the  $O(\log^* n)$  depth bound, we stop and output the trivial clustering consisting of singletons. This clustering clearly satisfies the diameter bound, and as we only output it with low probability, it has no effect on the expected number of inter-cluster edges. So we can conclude that the additional sorting overhead for the tie-breaking is a factor  $O(\log^* n)$ . The algorithm has total work  $O\left(m + \frac{n}{1-p}\right)$ , where the contribution of  $O\left(\frac{n}{1-p}\right)$  comes from sampling from the geometric distribution. In this paper, this factor vanishes as we always have  $p$  such that  $O\left(\frac{n}{1-p}\right) = O(m)$ .

<sup>3</sup> The “+1” appears, as nodes in the lowest level have distance 0 to the source  $s$ .

### 3 Constructing a Sparsified Low Diameter Decomposition

In this section, we show how the clustering algorithm leads to a sparsified low diameter decomposition. The procedure is as follows: given  $k \geq 2$ , we set  $r = k - 1$ ,  $p = 1 - n^{-1/k}$ , and compute a clustering using the algorithm of Section 2. We denote  $F \subseteq E$  for the sparsified set of inter-cluster edges. Intuitively, for each vertex  $v \in V$ , we add an edge to  $F$  for each cluster in which we have a neighbor on one level below, or a neighbor on the same level as  $v$  with the ID of the cluster center smaller than the ID of center of the cluster of  $v$ .

► **Lemma 9.** *There exists a set of edges  $F \subseteq E$  of expected size  $O(n^{1+1/k})$ , such that for every edge, one of its endpoints has an edge from  $F$  into the cluster of the other endpoint.*

**Proof.** We define,  $F := \bigcup_{x \in V} C(x)$ , where  $C(x)$  consists of the following edges

$$C(x) := \{(x, p_u(x)) : d^{(u)}(s, x) = d_{G'}(s, x)\} \\ \cup \{(x, p_u(x)) : d^{(u)}(s, x) = d_{G'}(s, x) + 1 \text{ and } \text{ID}(u) < \text{ID}(c_x)\}.$$

First, we show that  $F$  satisfies the property stated in the lemma, then we consider its size. Let  $(x, y) \in E$ , without loss of generality, we assume  $d_{G'}(s, x) \geq d_{G'}(s, y)$ , and in case of equality we assume  $\text{ID}(c_x) > \text{ID}(c_y)$ . We will show that there is an edge  $(x, p_{c_y}(x)) \in C(x)$  to the cluster of  $y$ . First of all, notice that  $d_{G'}(s, y) \geq d_{G'}(s, x) - 1$  by the triangle inequality. If  $d_{G'}(s, y) = d_{G'}(s, x) - 1$ , then  $d^{(c_y)}(s, x) \leq d_{G'}(s, x)$ . Because of minimality of  $d_{G'}(s, x)$ , we have  $d^{(c_y)}(s, x) = d_{G'}(s, x)$ , and thus  $(x, p_{c_y}(x)) \in C(x)$  by definition of  $C(x)$ . If  $d_{G'}(s, y) = d_{G'}(s, x)$ , we have  $\text{ID}(c_x) > \text{ID}(c_y)$ . Moreover, we have  $d^{(c_y)}(s, x) \leq d_{G'}(s, x) + 1$ . So again it follows that  $(x, p_{c_y}(x)) \in C(x)$  by definition of  $C(x)$ .

Now, we turn to the expected size of  $F$ . By linearity of expectation, we have  $\mathbb{E}[F] = \sum_{x \in V} \mathbb{E}[C(x)]$ . We will show that for each  $x \in V$  the expected size of  $C(x)$  is at most  $2n^{1/k}$ . For each  $u \in V$ , we potentially add an edge  $(x, p_u(x))$  to  $C(x)$ . First, we calculate the probability that at least  $t$  such vertices  $u$  contribute an edge. Hereto, we look at the random variables  $X_u = d^{(u)}(s, x) = k - \delta_u + d_G(u, x)$ . According to these random variables, we order all vertices:  $V = \{u_1, u_2, \dots, u_n\}$ , such that for  $i < j$  we satisfy one of the following properties

- $X_{u_i} < X_{u_j}$ ;
- $X_{u_i} = X_{u_j}$  and  $\text{ID}(u_i) < \text{ID}(u_j)$ .

We calculate  $\mathbb{P}[|C(X)| \geq t]$ , i.e., the probability that  $\{(x, p_{u_1}(x)), \dots, (x, p_{u_t}(x))\} \subseteq C(x)$ . We do this by conditioning on  $u_t = v$ . We observe

$$\mathbb{P}[\{(x, p_{u_1}(x)), \dots, (x, p_{u_t}(x))\} \subseteq C(x) \mid u_t = v] = \\ \prod_{i=1}^{t-1} \mathbb{P}[(x, p_{u_i}(x)) \in C(x) \text{ and } (x, p_v(x)) \in C(x) \mid u_t = v].$$

So we calculate  $\mathbb{P}[(x, p_{u_i}(x)) \in C(x) \text{ and } (x, p_v(x)) \in C(x) \mid u_t = v]$  for  $i < t$ . By definition of  $C(x)$ , this can only hold if either  $x$ 's closest neighbors in the clusters centered at  $u_i$  and  $v$  are on the same level (in which case we have  $\text{ID}(u_i) < \text{ID}(v)$ , as  $v = u_t$  and  $i < t$ ) or the neighbor from the cluster centered at  $u_i$  is at a level lower and  $\text{ID}(v) < \text{ID}(u_i)$ . Note that the level of the closest neighbor in the cluster of  $u_i$  or  $v$  corresponds to the distance  $d^{(u_i)}(s, x) = X_{u_i}$  or  $d^{(v)}(s, x)$  respectively. As the allowed distances depend on the ID of  $u_i$ , we split the vertices according to ID:

$$V_{<} := \{u \in V : \text{ID}(u) < \text{ID}(v)\}; \\ V_{>} := \{u \in V : \text{ID}(u) > \text{ID}(v)\}.$$

If  $u_i \in V_{<}$ , then we know  $X_{u_i} \leq d^{(v)}(s, x)$ . If both  $(x, p_{u_i}(x))$  and  $(x, p_v(x))$  are in  $C(x)$ , we must have  $X_{u_i} = d^{(v)}(s, x)$ . So for every  $i < t$ , we are looking at

$$\begin{aligned} & \mathbb{P}[(x, p_{u_i}(x)) \in C(x) \text{ and } (x, p_v(x)) \in C(x) \mid u_i \in V_{<} \text{ and } u_t = v] \\ & \leq \mathbb{P}\left[X_{u_i} = d^{(v)}(s, x) \mid u_i \in V_{<} \text{ and } u_t = v\right], \\ & = \mathbb{P}\left[X_{u_i} = d^{(v)}(s, x) \mid u_i \in V_{<} \text{ and } X_{u_i} \leq d^{(v)}(s, x)\right], \end{aligned}$$

where the last equality holds as we only rewrote the condition using the order of the  $u_j$ 's. We fill in the definition of  $X_{u_i}$  and use that the probability of the event we are looking at is independent of  $\text{ID}(u_i)$ :

$$\begin{aligned} & \mathbb{P}\left[X_{u_i} = d^{(v)}(s, x) \mid u_i \in V_{<} \text{ and } X_{u_i} \leq d^{(v)}(s, x)\right] \\ & = \mathbb{P}\left[k - \delta_{u_i} + d_G(u_i, x) = d^{(v)}(s, x) \mid u_i \in V_{<} \text{ and } k - \delta_{u_i} + d_G(u_i, x) \leq d^{(v)}(s, x)\right] \\ & = \mathbb{P}\left[k - \delta_{u_i} + d_G(u_i, x) = d^{(v)}(s, x) \mid k - \delta_{u_i} + d_G(u_i, x) \leq d^{(v)}(s, x)\right] \\ & = \mathbb{P}\left[\delta_{u_i} = k + d_G(u_i, x) - d^{(v)}(s, x) \mid \delta_{u_i} \geq k + d_G(u_i, x) - d^{(v)}(s, x)\right]. \end{aligned}$$

When  $k + d_G(u_i, x) - d^{(v)}(s, x) \leq k - 2$ , this equals  $p$ , by the memoryless property of the geometric distribution. To distinguish this, we partition the vertices  $u$  with  $\text{ID}(u) < \text{ID}(v)$  into two set

$$\begin{aligned} V_{<,1} & := \{u \in V : \text{ID}(u) < \text{ID}(v) \text{ and } k + d_G(u_i, x) - d^{(v)}(s, x) \leq k - 2\}; \\ V_{<,2} & := \{u \in V : \text{ID}(u) < \text{ID}(v) \text{ and } k + d_G(u_i, x) - d^{(v)}(s, x) = k - 1\}. \end{aligned}$$

Now for  $u_i \in V_{>}$ , we obtain by the same reasoning

$$\begin{aligned} & \mathbb{P}[(x, p_{u_i}(x)) \in C(x) \text{ and } (x, p_v(x)) \in C(x) \mid u_i \in V_{>} \text{ and } u_t = v] \\ & = \mathbb{P}\left[\delta_{u_i} = k + d_G(u_i, x) - d^{(v)}(s, x) + 1 \mid \delta_{u_i} \geq k + d_G(u_i, x) - d^{(v)}(s, x) + 1\right]. \end{aligned}$$

As before, when  $k + d_G(u_i, x) - d^{(v)}(s, x) + 1 \leq k - 2$ , this equals  $p$ , by the memoryless property of the geometric distribution. And again, we partition  $V_{>}$  into two sets

$$\begin{aligned} V_{>,1} & := \{u \in V : \text{ID}(u) > \text{ID}(v) \text{ and } k + d_G(u_i, x) - d^{(v)}(s, x) + 1 \leq k - 2\}; \\ V_{>,2} & := \{u \in V : \text{ID}(u) > \text{ID}(v) \text{ and } k + d_G(u_i, x) - d^{(v)}(s, x) + 1 = k - 1\}. \end{aligned}$$

If we define  $V_1 = V_{<,1} \cup V_{>,1}$  and  $V_2 = V_{<,2} \cup V_{>,2}$ , we can summarize our results as

$$\mathbb{P}[(x, p_{u_i}(x)) \in C(x) \text{ and } (x, p_v(x)) \in C(x) \mid u_i \in V_1 \text{ and } u_t = v] \leq p.$$

Next, we split the expected value of  $C(x)$  depending on  $V_1$  and  $V_2$ :

$$\mathbb{E}[|C(x)| \mid u_t = v] = \mathbb{E}[|C(x) \cap V_1| \mid u_t = v] + \mathbb{E}[|C(x) \cap V_2| \mid u_t = v].$$

We bound the first summand with  $n^{1/k}$ , independent of  $v$ . Hereto, we observe that for any non-negative discrete random variable  $X$  we have

$$\mathbb{E}[X] = \sum_{s=1}^{\infty} s \mathbb{P}[X = s] = \sum_{s=1}^{\infty} \sum_{t=1}^s \mathbb{P}[X = s] = \sum_{t=1}^{\infty} \sum_{s=t}^{\infty} \mathbb{P}[X = s] = \sum_{t=1}^{\infty} \mathbb{P}[X \geq t].$$

## 16:10 Spanners and Low Diameter Decompositions

Using this, we obtain

$$\begin{aligned}
& \mathbb{E}[|C(x) \cap V_1| \mid u_t = v] \\
&= \sum_{t=1}^n \mathbb{P}[|C(x) \cap V_1| \geq t \mid u_t = v] \\
&\leq \sum_{t=1}^n \prod_{i=1}^{t-1} \mathbb{P}[(x, p_{u_i}(x)) \in C(x) \text{ and } (x, p_v(x)) \in C(x) \mid u_i \in V_1 \text{ and } u_t = v] \\
&\leq \sum_{t=1}^n p^{t-1} \\
&\leq \sum_{t=0}^{\infty} p^t \\
&= \frac{1}{1-p} \\
&= n^{1/k},
\end{aligned}$$

where the last equality holds by definition of  $p$ . For the second summand, we look at all  $v$  simultaneously.

$$\begin{aligned}
\sum_{v \in V} \mathbb{E}[|C(x) \cap V_2| \mid u_t = v] \mathbb{P}[u_t = v] &\leq \sum_{v \in V} \mathbb{E}[|V_2| \mid u_t = v] \mathbb{P}[u_t = v] \\
&\leq \sum_{v \in V} \mathbb{E}[|\{u \in V : \delta_u = k-1\}| \mid u_t = v] \mathbb{P}[u_t = v] \\
&= \mathbb{E}[|\{u \in V : \delta_u = k-1\}|],
\end{aligned}$$

where the last equality holds by the law of total probability. We bound this as follows

$$\mathbb{E}[|\{u \in V : \delta_u = k-1\}|] \leq n \mathbb{P}[\delta_u = k-1] = n(1-p)^{k-1} = n^{1/k},$$

where the last equality holds by definition of  $p$ . In total, this gives us  $\mathbb{E}[|C(x)|] \leq 2n^{1/k}$ . ◀

Together Lemma 7 and Lemma 9 imply the following theorem.

► **Theorem 3 (Restated).** *There exists an algorithm, such that for each unweighted graph  $G = (V, E)$  and parameter  $k \geq 2$  it outputs a tree-supported sparsified  $(\zeta, 2k-2)$ -low diameter decomposition, with  $\zeta = O(n^{1+1/k})$  in expectation. The algorithm runs in  $k$  rounds in the CONGEST model, and in  $O(k \log^* n)$  depth and  $O(m)$  work in the PRAM model.*

### 3.1 Constructing a Spanner

Now, we can construct a spanner from the tree supported low diameter decomposition in the following manner. Let  $T$  denote the support forest, and let  $F$  denote the set as given in Lemma 9. We define the spanner  $H := (V, T \cup F)$ . As any forest has at most  $n-1$  edges, the expected size of  $H$  is at most  $O(n^{1+1/k})$ . Actually, one could also show that  $T \subseteq F$  in our construction of  $F$ , but this would not impact the asymptotic size bound. To show that  $H$  is a spanner, we show its of limited stretch.

► **Lemma 10.**  *$H$  is a spanner of stretch  $2k-1$ .*



**Proof.** We will show that for every edge  $(x, y) \in E$ , there exists a path from  $x$  to  $y$  in  $H$  of length at most  $2k - 1$ . Consequently we have that  $d_H(x, y) \leq (2k - 1)d_G(x, y)$  for every  $x, y \in V$ , hence  $H$  is a spanner of stretch  $2k - 1$ .

Let  $(x, y) \in E$ . By definition of  $F$ , one of the endpoints has an edge in  $F$  into the cluster of the other endpoint. Without loss of generality, let there be an edge  $(x, z) \in F$  with  $z$  in the cluster of  $y$ . By Corollary 8, there is path of length at most  $2(k - 1)$  from  $z$  to  $y$ , so in total we have a path of length at most  $2(k - 1) + 1 = 2k - 1$  from  $x$  to  $z$  to  $y$ . ◀

Now, the following corollary follows from Theorem 3 and Lemma 10.

► **Corollary 11 (Restated).** *There exists an algorithm, such that for each unweighted graph  $G = (V, E)$  and parameter  $k \geq 2$  it outputs a spanner  $H$  of stretch  $2k - 1$ . The expected size of  $H$  is at most  $O(n^{1+1/k})$ . The algorithm runs in  $k$  rounds in the CONGEST model, and in  $O(k \log^* n)$  depth and  $O(m)$  work in the PRAM model.*

### 3.2 Constructing a Synchronizer

Suppose we are given a synchronous CONGEST model algorithm, but we want to run it on an asynchronous CONGEST network. That is, the messages sent in the network can now have arbitrary delays and, in an event-driven manner, nodes become active each time they receive a message. For the purpose of analyzing the time complexity of the algorithm, it is often assumed that the delay is at most one unit of time, however, the algorithm should behave correctly under *any* finite delays. In this situation, a node should start simulating its next (synchronous) round when it has received all the messages from the previous round from its neighbors. The problem is that it cannot tell the difference between the situation if a message from a particular neighbor has not arrived yet or if this same neighbor is not sending any message in that round at all. We say that a node is *safe* if all the messages it has sent have arrived at their destination. In order to determine whether all neighboring nodes are safe, additional messages are sent. The procedure governing these additional messages is called the *synchronizer*. There are two things to take into account when analyzing synchronizers. First, the time overhead: how much time is needed to send the additional messages for each synchronous round. Second, the message-complexity (or communication) overhead: how many additional messages are sent. For more details on synchronizers see e.g. [29, 26].

Let us first consider two simple synchronizers: synchronizer  $\alpha$  and synchronizer  $\beta$ , see [4]. In synchronizer  $\alpha$ , when a node receives a message from a neighbor, it sends back an “acknowledge” message. When a node has received acknowledge messages for all its sent messages, it marks itself safe and reports this to all its neighbors. The synchronizer  $\alpha$  uses, for each simulated synchronous round, additional  $O(1)$  time, and  $O(m)$  messages.

Synchronizer  $\beta$  will produce a different trade off between time and message overhead. It uses an initialization phase in which it creates a rooted spanning tree, where the root is declared the leader. Now after sending messages of a certain synchronous round, again nodes that receive messages reply with an acknowledge message to each. Nodes that are safe, and whose children in the constructed tree are also safe communicate this to their parent in the tree. Eventually the leader will learn that the whole graph is safe, and will broadcast this along the spanning tree. Synchronizer  $\beta$  uses  $O(D)$  time and  $O(n)$  messages per synchronous round.

Now we are ready to consider a little more involved example, called synchronizer  $\gamma$ , see [4]. This synchronizer makes use of clustering, where within each cluster synchronizer  $\beta$  is used and between clusters synchronizer  $\alpha$  is used. In the LOCAL model, the cluster centers can simply select a communication link for each neighboring cluster to communicate

individually with the neighboring cluster centers [4]. However, in the CONGEST model, communicating information about neighboring clusters to the cluster center might lead to congestion problems. Using a slightly more careful analysis, the procedure can be adapted to the CONGEST model.

► **Lemma 11** (Implicit in [4]). *Given a  $T(n)$ -round synchronous CONGEST model algorithm for constructing a sparsified  $(\zeta, \delta)$ -low diameter decomposition, any synchronous  $R(n)$ -round  $M(n)$ -message complexity CONGEST model algorithm can be run on an asynchronous CONGEST network with a total of  $O(M(n) + R(n)(\zeta + n))$  messages, and, provided that each message incurs a delay of at most one time unit, in time  $O(R(n)\delta)$ . The initialization phase takes  $O(T(n))$  time, using  $O((T(n) + \delta)m)$  messages.*

For a sketch of the algorithm, we refer to Appendix A. If we plug in our clustering, we obtain the following theorem.

► **Theorem 5** (Restated). *There exists an algorithm that, given  $k \geq 2$ , can run any synchronous  $R(n)$ -round  $M(n)$ -message complexity CONGEST model algorithm on an asynchronous CONGEST network. In expectation, the algorithm uses a total of  $O(M(n) + R(n)n^{1+1/k})$  messages. Provided that each message incurs a delay of at most one time unit, it takes  $O(R(n)k)$  rounds. The initialization phase takes  $O(k)$  time, using  $O(km)$  messages.*

## 4 Constructing a Low Diameter Decomposition

In this section, we show that for an integer weighted graph the computed clustering is a probabilistic low diameter decomposition. To be precise, if we set  $p = \frac{\beta}{4}$ , and  $r = \left\lceil \frac{1}{p} \ln \left( \frac{n^2}{p} \right) + \frac{1}{4p} \right\rceil$  we obtain a  $(\beta, O(\frac{\log n}{\beta}))$ -low diameter decomposition. By Corollary 8, we know that each of the clusters has a strong diameter of at most  $2r = 2 \left\lceil \frac{1}{p} \ln \left( \frac{n^2}{p} \right) + \frac{1}{4p} \right\rceil = O\left(\frac{\log n}{\beta}\right)$ . Now, we show that the probability that an edge  $e \in E$  is an inter-cluster edge is at most  $4p \cdot w(e) = \beta \cdot w(e)$ . We use a general proof structure from [31], but make it more streamlined; we avoid an artificially constructed “midpoint” on the edge  $(u, v)$ . Further, our proof borrows the idea of conditioning on the event  $E_{u', v', \alpha}$  from Xu [39], which we adapt to our situation.

► **Lemma 12.** *For  $(u, v) \in E$ , the probability that  $u$  and  $v$  belong to different clusters is at most  $4p \cdot w(u, v)$ .*

**Proof.** Suppose  $(u, v)$  is an inter-cluster edge. Without loss of generality, we assume  $d_{G'}(s, v) \leq d_{G'}(s, u)$ . By the triangle inequality, we have  $d_{G'}(s, u) \leq d_{G'}(s, v) + w(u, v)$ , hence we have  $d_{G'}(s, u) - d_{G'}(s, v) \leq w(u, v)$ . Using this, we can upper bound the probability that an edge  $(u, v)$  is an inter-cluster edge by the probability that this inequality holds. Note that we can assume  $4p \cdot w(u, v) < 1$ , otherwise the statement is trivially true.

We want to condition on the cluster center  $v'$  that satisfied  $d^{(v')}(s, u) = d_{G'}(s, v)$ , and on the cluster center  $u' \neq v'$  that minimizes  $d^{(u')}(s, u)$ . Moreover, we ask that these cluster respect the tie-breaking rule, i.e., both have minimal ID among all centers with equal distance. Finally, we condition on the value of  $d_{G'}(s, u)$ , which we set to  $\alpha$ . We call this event  $E_{u', v', \alpha}$ , which we formally define to hold when the following four conditions are satisfied

1.  $d^{(v')}(s, u) \leq \alpha$ ;
2. for  $w' \in V \setminus \{v'\}$  we either have  $d^{(w')}(s, v) > d^{(v')}(s, v)$ , or we have  $d^{(w')}(s, v) = d^{(v')}(s, v)$  and  $\text{ID}(v') < \text{ID}(w')$ ;

3.  $d^{(u')}(s, v) = \alpha$ ;
4. for  $w' \in V \setminus \{u', v'\}$  we either have  $d^{(w')}(s, u) > d^{(u')}(s, u)$ , or we have  $d^{(w')}(s, u) = d^{(u')}(s, u)$  and  $\text{ID}(u') < \text{ID}(w')$ .

Now, we condition on  $E_{u', v', \alpha}$  and use the law of total probability:

$$\begin{aligned}
& \mathbb{P}[(u, v) \text{ is an inter-cluster edge}] \\
&= \sum_{u' \in V} \sum_{v' \in V \setminus \{u'\}} \sum_{\alpha} \mathbb{P}[(u, v) \text{ is an inter-cluster edge} \mid E_{u', v', \alpha}] \mathbb{P}[E_{u', v', \alpha}] \\
&\leq \sum_{u' \in V} \sum_{v' \in V \setminus \{u'\}} \sum_{\alpha} 2 \mathbb{P} \left[ d^{(u')}(s, u) - d^{(v')}(s, v) \leq w(u, v) \mid E_{u', v', \alpha} \right] \mathbb{P}[E_{u', v', \alpha}].
\end{aligned}$$

For simplicity, we omit the bounds for the sum over  $\alpha$ , which is a finite sum as we always have  $\alpha \leq r + m$ . The factor two appears because the event assumes  $d_{G'}(s, v) \leq d_{G'}(s, u)$ , hence we gain a factor two by symmetry of  $u$  and  $v$ . We look at the first probability more closely. We can loosen some of the event's restrictions, and just maintain  $d^{(v')}(s, v) \leq \alpha$ , as the event we examine is independent of conditions 2, 3, and 4 of the event  $E_{u', v', \alpha}$ . We obtain

$$\begin{aligned}
& \mathbb{P} \left[ d^{(u')}(s, u) - d^{(v')}(s, v) \leq w(u, v) \mid E_{u', v', \alpha} \right] \\
&= \mathbb{P} \left[ \alpha - d^{(v')}(s, v) \leq w(u, v) \mid d^{(v')}(s, v) \leq \alpha \right] \\
&= \mathbb{P} [\delta_{v'} \leq r + d_G(v', v) - \alpha + w(u, v) \mid \delta_{v'} \geq r + d_G(v', v) - \alpha],
\end{aligned}$$

where the last equality holds by definition of  $d^{(v')}(s, v)$ . Now if  $r + d_G(v', v) - \alpha + w(u, v) \leq r$  (or equivalently,  $\alpha \geq d_G(v', v) + w(u, v)$ ), we stay away from our cap on the geometric distribution, and hence we can apply the memoryless property of the geometric distribution to obtain

$$\begin{aligned}
& \mathbb{P} [\delta_{v'} \leq r + d_G(v', v) - \alpha + w(u, v) \mid \delta_{v'} \geq r + d_G(v', v) - \alpha] \\
&= 1 - (1 - p)^{w(u, v)} \\
&\leq pw(u, v),
\end{aligned}$$

where the last step holds by Bernoulli's inequality. If we have  $r + d_G(v', v) - \alpha + w(u, v) > r$  (or equivalently,  $\alpha < d_G(v', v) + w(u, v)$ ), we show that the probability  $\mathbb{P}[E_{u', v', \alpha}]$  of the event taking place is already small:

$$\begin{aligned}
\mathbb{P}[E_{u', v', \alpha}] &\leq \mathbb{P} \left[ d^{(v')}(s, v) \leq \alpha \text{ and } d^{(u')}(s, u) = \alpha \right] \\
&= \mathbb{P} [\delta_{v'} \geq r + d_G(v', v) - \alpha] \mathbb{P} \left[ d^{(u')}(s, u) = \alpha \right] \quad (\text{since the events are independent}) \\
&\leq \mathbb{P} [\delta_{v'} \geq r - w(u, v)] \mathbb{P} \left[ d^{(u')}(s, u) = \alpha \right],
\end{aligned}$$

where the last equality holds as  $r + d_G(v', v) - \alpha + w(u, v) > r$ . We bound this probability as follows:

$$\mathbb{P} [\delta_{v'} \geq r - w(u, v)] \mathbb{P} \left[ d^{(u')}(s, u) = \alpha \right] \leq (1 - p)^{r - w(u, v)} \mathbb{P} \left[ d^{(u')}(s, u) = \alpha \right].$$

## 16:14 Spanners and Low Diameter Decompositions

Now we use that  $r = \left\lceil \frac{1}{p} \ln \left( \frac{n^2}{p} \right) + \frac{1}{4p} \right\rceil$ , to obtain

$$\begin{aligned} (1-p)^{r-w(u,v)} &\leq (1-p)^{\frac{1}{p} \ln \left( \frac{n^2}{p} \right) + \frac{1}{4p} - w(u,v)} \\ &\leq \left( (1-p)^{1/p} \right)^{\ln \left( \frac{n^2}{p} \right)} \quad (\text{as } 4p \cdot w(u,v) \leq 1) \\ &\leq \frac{p}{n^2}. \end{aligned}$$

Combining all of this, we obtain

$$\begin{aligned} &\mathbb{P}[(u, v) \text{ is an inter-cluster edge}] \\ &\leq 2 \sum_{u' \in V} \sum_{v' \in V \setminus \{u'\}} \sum_{\alpha \geq d_G(v', v) + w(u, v)} \mathbb{P} \left[ d^{(u')}(s, u) - d^{(v')}(s, v) \leq w(u, v) \mid E_{u', v', \alpha} \right] \mathbb{P} [E_{u', v', \alpha}] \\ &\quad + 2 \sum_{u' \in V} \sum_{v' \in V \setminus \{u'\}} \sum_{\alpha < d_G(v', v) + w(u, v)} \mathbb{P} \left[ d^{(u')}(s, u) - d^{(v')}(s, v) \leq w(u, v) \mid E_{u', v', \alpha} \right] \mathbb{P} [E_{u', v', \alpha}] \\ &\leq 2 \sum_{u' \in V} \sum_{v' \in V \setminus \{u'\}} \sum_{\alpha \geq d_G(v', v) + w(u, v)} p \cdot w(u, v) \mathbb{P} [E_{u', v', \alpha}] \\ &\quad + 2 \sum_{u' \in V} \sum_{v' \in V \setminus \{u'\}} \sum_{\alpha < d_G(v', v) + w(u, v)} \frac{p}{n^2} \mathbb{P} \left[ d^{(u')}(s, u) = \alpha \right] \\ &\leq 2p \cdot w(u, v) \sum_{u' \in V} \sum_{v' \in V \setminus \{u'\}} \sum_{\alpha} \mathbb{P} [E_{u', v', \alpha}] + 2 \frac{p}{n^2} \sum_{u' \in V} \sum_{v' \in V \setminus \{u'\}} \sum_{\alpha} \mathbb{P} \left[ d^{(u')}(s, u) = \alpha \right]. \end{aligned}$$

Next, we notice that all events  $E_{u', v', \alpha}$  are disjoint by design, so

$$\sum_{u' \in V} \sum_{v' \in V \setminus \{u'\}} \sum_{\alpha} \mathbb{P} [E_{u', v', \alpha}] = 1.$$

Clearly we have  $\sum_{\alpha} \mathbb{P} \left[ d^{(u')}(s, u) = \alpha \right] = 1$ , as this is just a sum over all possible values of  $d^{(u')}(s, u)$ . Filling both in, we conclude

$$\mathbb{P}[(u, v) \text{ is an inter-cluster edge}] \leq 2p \cdot w(u, v) + 2 \sum_{u' \in V} \sum_{v' \in V \setminus \{u'\}} \frac{p}{n^2} \leq 4p \cdot w(u, v). \quad \blacktriangleleft$$

Together with Corollary 8, this gives us the following theorem.

► **Theorem 6 (Restated).** *There exists an algorithm, such that for each graph  $G = (V, E)$  with integer weights  $w: E \rightarrow \{1, \dots, W\}$  and parameter  $\beta \in (0, 1]$  it outputs a low diameter decomposition, whose components are clusters of strong diameter of at most  $O\left(\frac{\log n}{\beta}\right)$ . Moreover, each edge is an inter-cluster edge with probability at most  $\beta \cdot w(u, v)$ . The algorithm runs in  $O\left(\frac{\log n}{\beta}\right)$  rounds in the CONGEST model, and in  $O\left(\frac{\log n \log^* n}{\beta}\right)$  depth and  $O(m)$  work in the PRAM model.*

## 5 Conclusion

We have presented an algorithm that computes a clustering, more precisely, a tree-supported sparsified low diameter decomposition. This directly leads to a sparse spanner and can be applied to compute a synchronizer for the CONGEST model. Moreover, we show that we also improve upon the state-of-the-art for low diameter decompositions. By showing

that clustering can be done using a capped geometric distribution, we improve on existing algorithms for spanners and low diameter decompositions in two ways. First, we obtain bounds on the diameter/stretch and running time that are independent of the random choices of the algorithm. Second, the discreteness of the geometric distribution fits the discrete nature of graph theoretical problems better than a continuous distribution. We believe this leads to a more intuitive algorithm.

A natural question that remains is whether it would be possible to give a with-high-probability bound on the total number of inter-cluster edges or the size of the spanner rather than an in-expectation bound. A more ambitious goal is to develop a completely deterministic algorithm with the same bounds, improving on the work of Ghaffari and Kuhn [20].

---

## References

- 1 Vedat Levi Alev, Nima Anari, Lap Chi Lau, and Shayan Oveis Gharan. Graph clustering using effective resistance. In *Proc. of the Innovations in Theoretical Computer Science Conference (ITCS)*, pages 41:1–41:16, 2018.
- 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 Ingo Althöfer, Gautam Das, David Dobkin, Deborah Joseph, and José Soares. On sparse spanners of weighted graphs. *Discrete & Computational Geometry*, 9(1):81–100, 1993.
- 4 Baruch Awerbuch. Complexity of network synchronization. *Journal of the ACM*, 32(4):804–823, 1985.
- 5 Baruch Awerbuch, Bonnie Berger, Lenore Cowen, and David Peleg. Near-linear cost sequential and distributed constructions of sparse neighborhood covers. In *Proc. of the Conference on Foundations of Computer Science (FOCS)*, pages 638–647, 1993.
- 6 Baruch Awerbuch, Andrew V Goldberg, Michael Luby, and Serge A Plotkin. Network decomposition and locality in distributed computation. In *Proc. of the Conference on Foundations of Computer Science (FOCS)*, volume 30, pages 364–369, 1989.
- 7 Baruch Awerbuch, Boaz Patt-Shamir, David Peleg, and Michael Saks. Adapting to asynchronous dynamic networks. In *Proc. of the Symposium on Theory of Computing (STOC)*, pages 557–570, 1992.
- 8 Baruch Awerbuch and David Peleg. Network synchronization with polylogarithmic overhead. In *Proc. of the Symposium on Foundations of Computer Science (FOCS)*, pages 514–522, 1990.
- 9 Yair Bartal. Probabilistic approximation of metric spaces and its algorithmic applications. In *Proc. of the Conference on Foundations of Computer Science (FOCS)*, pages 184–193, 1996.
- 10 Yair Bartal. On approximating arbitrary metrics by tree metrics. In *Proc. of the Symposium on Theory of Computing (STOC)*, pages 161–168, 1998.
- 11 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.
- 12 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 *Proc. of the Symposium on Theory of Computing (STOC)*, pages 961–970, 2012.
- 13 Yi-Jun Chang, Seth Pettie, and Hengjie Zhang. Distributed triangle detection via expander decomposition. In *Proc. of the Symposium on Discrete Algorithms (SODA)*, pages 821–840, 2019.
- 14 Yi-Jun Chang and Thatchaphol Saranurak. Deterministic distributed expander decomposition and routing with applications in distributed derandomization. In *Proc. of the Symposium on Foundations of Computer Science (FOCS)*, pages 377–388, 2020.
- 15 Edith Cohen. Polylog-time and near-linear work approximation scheme for undirected shortest paths. In *Proc. of the Symposium on Theory of Computing (STOC)*, pages 16–26, 1994.

## 16:16 Spanners and Low Diameter Decompositions

- 16 Edith Cohen. Fast algorithms for constructing  $t$ -spanners and paths with stretch  $t$ . *SIAM Journal on Computing*, 28(1):210–236, 1998.
- 17 Michael Elkin and Ofer Neiman. Efficient algorithms for constructing very sparse spanners and emulators. *ACM Transactions on Algorithms (TALG)*, 15, 2019.
- 18 Arnold Filtser and Shay Solomon. The greedy spanner is existentially optimal. In *Proc. of the Symposium on Principles of Distributed Computing (PODC)*, pages 9–17, 2016.
- 19 Sebastian Forster and Gramoz Goranci. Dynamic low-stretch trees via dynamic low-diameter decompositions. In *Proc. of the Symposium on Theory of Computing (STOC)*, pages 377–388, New York, NY, USA, 2019.
- 20 Mohsen Ghaffari and Fabian Kuhn. Derandomizing distributed algorithms with small messages: Spanners and dominating set. In *Proc. of the Symposium on Distributed Computing (DISC)*, pages 29:1–29:17, 2018.
- 21 Joseph Gil, Yossi Matias, and Uzi Vishkin. Towards a theory of nearly constant time parallel algorithms. In *Proc. of the Symposium of Foundations of Computer Science (FOCS)*, pages 698–710, 1991.
- 22 Oded Goldreich and Dana Ron. A sublinear bipartiteness tester for bounded degree graphs. *Combinatorica*, 19(3):335–373, 1999.
- 23 Shay Halperin and Uri Zwick. Personal communication, 1993.
- 24 Ravi Kannan, Santosh Vempala, and Adrian Vetta. On clusterings: Good, bad and spectral. *Journal of the ACM*, 51(3):497–515, 2004.
- 25 Philip N Klein and Sairam Subramanian. A randomized parallel algorithm for single-source shortest paths. *Journal of Algorithms*, 25(2):205–220, 1997.
- 26 Ajay D Kshemkalyani and Mukesh Singhal. *Distributed computing: principles, algorithms, and systems*. Cambridge University Press, 2011.
- 27 Tom Leighton and Satish Rao. Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. *Journal of the ACM*, 46(6):787–832, 1999.
- 28 Nathan Linial and Michael Saks. Low diameter graph decompositions. *Combinatorica*, 13(4):441–454, 1993.
- 29 Nancy A Lynch. *Distributed algorithms*. Elsevier, 1996.
- 30 Gary L Miller, Richard Peng, Adrian Vladu, and Shen Chen Xu. Improved parallel algorithms for spanners and hopsets. In *Proc. Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 192–201, 2015.
- 31 Gary L Miller, Richard Peng, and Shen Xu. Parallel graph decompositions using random shifts. *Proc. of the Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 196–203, 2013.
- 32 David Peleg. Proximity-preserving labeling schemes. *Journal of Graph Theory*, 33(3):167–176, 2000.
- 33 David Peleg and Eli Upfal. A trade-off between space and efficiency for routing tables. *Journal of the ACM*, 36(3):510–530, 1989.
- 34 Liam Roditty and Uri Zwick. On dynamic shortest paths problems. In *Proc. of the European Symposium on Algorithms (ESA)*, pages 580–591, 2004.
- 35 Thatchaphol Saranurak and Di Wang. Expander decomposition and pruning: Faster, stronger, and simpler. In *Proc. of the Symposium on Discrete Algorithms (SODA)*, pages 2616–2635, 2019.
- 36 Daniel A Spielman and Shang-Hua Teng. Spectral sparsification of graphs. *SIAM Journal on Computing*, 40(4):981–1025, 2011.
- 37 Mikkel Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *Journal of the ACM*, 46(3):362–394, 1999.
- 38 Mikkel Thorup and Uri Zwick. Approximate distance oracles. *Journal of the ACM*, 52(1):1–24, 2005.
- 39 Shen Chen Xu. *Exponential Start Time Clustering and its Applications in Spectral Graph Theory*. PhD thesis, Carnegie Mellon University, Pittsburgh, 2017.

## A Using Sparsified Low Diameter Decompositions for Synchronization

In the following, we turn to the algorithm realizing Lemma 11, i.e., we show how we can run a synchronous CONGEST algorithm on an asynchronous CONGEST network, using a sparsified low diameter decomposition. Hereto, we present an implementation of the synchronizer  $\gamma$  in the CONGEST model, using sparsified low diameter decompositions for the communication. We refer to [4] for a proof of correctness of the synchronizer  $\gamma$ .

The initialization phase consists of three steps. First, we compute the sparsified  $(\zeta, \delta)$ -low diameter decomposition. To do this in the asynchronous CONGEST model, we use the synchronizer  $\alpha$  (for details see [4], or textbooks, e.g., [29, 26]). Hence this takes  $O(T(n))$  time, and  $O(T(n)m)$  messages. Second, we pick a cluster center for each cluster and construct a tree rooted at the cluster center spanning the cluster. We can do this in  $O(\delta)$  time, using  $O(\delta m)$  messages, again using the synchronizer  $\alpha$ . Note that if the computed decomposition was tree-supported, these trees are already given. As a third and final step of the initialization phase, each vertex needs to be aware of its incident sparsified inter-cluster edges, as it will use these to communicate to neighboring clusters. This might be already determined during the construction of the clustering. It could also be the case that for each sparsified inter-cluster edge, only one of the two incident vertices knows this. In  $O(1)$  time, using  $O(m)$  messages, this can be communicated using the synchronizer  $\alpha$ . In total, we use  $O(T(n))$  time for the initialization phase, and  $O((T(n) + \delta)m)$  messages.

Now we are set up for the simulation of the  $R(n)$ -round,  $M(n)$ -message complexity synchronous CONGEST model algorithm. In each simulation of a synchronous round of this algorithm, vertices respond to each message with an acknowledge message, same as in the synchronizers  $\alpha$  and  $\beta$ . When a vertex has received acknowledge messages for each sent message, it declares itself safe. If a vertex and all its children in the cluster tree are safe, it notifies its parent in the cluster tree. Once the cluster center has received confirmation that the whole cluster is safe, it down-casts this information to the whole cluster. Each vertex communicates that its cluster is safe over its sparsified inter-cluster edges. Once a vertex received a message of being safe over each sparsified inter-cluster edge, it declares itself *ready*. When a vertex and all its children in the cluster tree are ready, it sends a ready-message to its parent in the cluster tree. Once a cluster center received ready-messages from the whole cluster, it down-casts a message “cluster ready” to all cluster vertices.

Assuming that each message incurs a delay of at most one time unit, we need at most  $O(\delta)$  time for this procedure, as we send information along the trees of height  $\delta$  for a total of four times. See [4] for the argument explaining why confirmation that neighboring clusters are done suffices. Moreover, the only communication links participating in this procedure, are the edges from the sparsified low diameter decomposition (consisting of at most  $\zeta$  inter-cluster edges and  $n$  tree edges). Each of these edges sends up to four messages in total, giving a total bound on the number of messages of  $O(R(n)(\zeta + n))$  for the synchronization. This gives a total time bound of  $O(R(n)\delta)$ , and message complexity bound of  $O(M(n) + R(n)(\zeta + n))$ .





# Distributed CONGEST Approximation of Weighted Vertex Covers and Matchings

Salwa Faour ✉

University of Freiburg, Germany

Marc Fuchs ✉

University of Freiburg, Germany

Fabian Kuhn ✉

University of Freiburg, Germany

---

## Abstract

We provide CONGEST model algorithms for approximating the minimum weighted vertex cover and the maximum weighted matching problem. For bipartite graphs, we show that a  $(1 + \varepsilon)$ -approximate weighted vertex cover can be computed deterministically in  $\text{poly}\left(\frac{\log n}{\varepsilon}\right)$  rounds. This generalizes a corresponding result for the unweighted vertex cover problem shown in [Faour, Kuhn; OPODIS '20]. Moreover, we show that in general weighted graph families that are closed under taking subgraphs and in which we can compute an independent set of weight at least  $\lambda \cdot w(V)$  (where  $w(V)$  denotes the total weight of all nodes) in polylogarithmic time in the CONGEST model, one can compute a  $(2 - 2\lambda + \varepsilon)$ -approximate weighted vertex cover in  $\text{poly}\left(\frac{\log n}{\varepsilon}\right)$  rounds in the CONGEST model. Our result in particular implies that in graphs of arboricity  $a$ , one can compute a  $(2 - 1/a + \varepsilon)$ -approximate weighted vertex cover problem in  $\text{poly}\left(\frac{\log n}{\varepsilon}\right)$  rounds in the CONGEST model.

For maximum weighted matchings, we show that a  $(1 - \varepsilon)$ -approximate solution can be computed deterministically in time  $2^{O(1/\varepsilon)} \cdot \text{poly} \log n$  in the CONGEST model. We also provide a randomized algorithm that with arbitrarily good constant probability succeeds in computing a  $(1 - \varepsilon)$ -approximate weighted matching in time  $2^{O(1/\varepsilon)} \cdot \text{poly} \log(\Delta W) \cdot \log^* n$ , where  $W$  denotes the ratio between the largest and the smallest edge weight. Our algorithm generalizes results of [Lotker, Patt-Shamir, Pettie; SPAA '08] and [Bar-Yehuda, Hillel, Ghaffari, Schwartzman; PODC '17], who gave  $2^{O(1/\varepsilon)} \cdot \log n$  and  $2^{O(1/\varepsilon)} \cdot \frac{\log \Delta}{\log \log \Delta}$ -round randomized approximations for the unweighted matching problem.

Finally, we show that even in the LOCAL model and in bipartite graphs of degree  $\leq 3$ , if  $\varepsilon < \varepsilon_0$  for some constant  $\varepsilon_0 > 0$ , then computing a  $(1 + \varepsilon)$ -approximation for the unweighted minimum vertex cover problem requires  $\Omega\left(\frac{\log n}{\varepsilon}\right)$  rounds. This generalizes a result of [Göös, Suomela; DISC '12], who showed that computing a  $(1 + \varepsilon_0)$ -approximation in such graphs requires  $\Omega(\log n)$  rounds.

**2012 ACM Subject Classification** Theory of computation  $\rightarrow$  Distributed algorithms

**Keywords and phrases** distributed graph algorithms, minimum weighted vertex cover, maximum weighted matching, distributed optimization, CONGEST model

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2021.17

**Related Version** *Full Version*: <https://arxiv.org/abs/2111.10577>

## 1 Introduction and Related Work

Maximum matching (MM) and minimum vertex cover (MVC) are two classic optimization problems that have been studied intensively in the context of distributed graph algorithms (e.g., [1–5, 7–11, 14, 15, 21, 23, 25, 28, 32–35, 38, 40, 45–47, 49, 50, 52, 56]). The problems are closely related to each other: the fractional relaxations of the unweighted variants of the problem are linear programming (LP) duals of each other. The problems are however also fundamentally different. While a maximum (weighted) matching can be found in polynomial time in all graphs [18, 19], for the minimum (weighted) vertex cover problem, this is only true for bipartite graphs [20, 42]. In general graphs, even for the unweighted MVC problem, the



© Salwa Faour, Marc Fuchs, and Fabian Kuhn;  
licensed under Creative Commons License CC-BY 4.0

25th International Conference on Principles of Distributed Systems (OPODIS 2021).

Editors: Quentin Bramas, Vincent Gramoli, and Alessia Milani; Article No. 17; pp. 17:1–17:20

Leibniz International Proceedings in Informatics



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

best polynomial-time approximation algorithms have an approximation ratio of  $2 - o(1)$  [41]. The MVC problem is known to be APX-hard [17, 36], and if the unique games conjecture holds, the current  $(2 - o(1))$ -approximation algorithms are essentially best possible [43].

In the distributed context, most prominently, the problems have been studied in the standard message passing models in graphs, in the LOCAL model and the CONGEST model [53]. In both models, the graph  $G = (V, E)$  on which we want to solve some graph problem also represents the network and it is assumed that the nodes  $V$  of  $G$  can communicate with each other in synchronous rounds by exchanging messages over the edges  $E$  of  $G$ . In the LOCAL model, the size of those messages is not restricted, whereas in the CONGEST model, it is assumed that each message has to consist of at most  $O(\log n)$  bits, where  $n = |V|$  is the number of nodes of the network graph  $G$ . In the following discussion of existing work on distributed matching and vertex cover algorithms, we concentrate on polylogarithmic-time distributed algorithms that also work for the weighted variants of the problems.

**Distributed Complexity of Weighted Matchings.** While the unweighted versions of both problems can be approximated within a factor of 2 by computing a maximal matching, a little more work is needed for weighted matchings and vertex covers. The first polylogarithmic-time distributed algorithm for computing a constant approximation for the maximum weighted matching (MWM) problem was presented in [56]. This algorithm was then improved in [50] and in [49], where it is shown that a  $(1/2 - \varepsilon)$ -approximation for MWM can be computed in  $O(\log(1/\varepsilon) \log n)$  rounds in the CONGEST model. In [7], it was further shown that can one compute a  $1/2$ -approximation for MWM in time  $O(\log W \cdot T_{\text{MIS}})$  in the CONGEST model, where  $W$  is the ratio between the largest and smallest edge weight and where  $T_{\text{MIS}}$  is the time for computing a maximal independent set. The paper also shows that for constant  $\varepsilon$ , a  $(1/2 - \varepsilon)$ -approximation can be computed in only  $O\left(\frac{\log \Delta}{\log \log \Delta}\right)$  rounds. Note that as shown in [47], this time complexity is best possible for any constant approximation algorithm, even in the LOCAL model. All the above algorithms are randomized. In [25], Fischer gave a deterministic CONGEST algorithm to compute a  $(1/2 - \varepsilon)$ -approximate weighted matching with a round complexity of  $O(\log^2 \Delta \cdot \log 1/\varepsilon + \log^* n)$ . This algorithm was refined in [2], where it was shown that in time  $O\left(\frac{\log^2(\Delta/\varepsilon) + \log^* n}{\varepsilon} + \frac{\log(\Delta W)}{\varepsilon^2}\right)$ , it is even possible to deterministically compute a  $(2/3 - \varepsilon)$ -approximation for the MWM problem in general graphs and a  $(1 - \varepsilon)$ -approximation for the MWM problem in bipartite graphs, in the CONGEST model. To the best of our knowledge, this is the only existing polylogarithmic-time CONGEST algorithm to obtain an approximation ratio that is better than  $1/2$ . It has been observed already in [45, 49, 52] that in the LOCAL model, better approximations for maximum weighted matching can be computed efficiently. In particular, [49, 52] show that even in general graphs, a  $(1 - \varepsilon)$ -approximation can be computed in  $\text{poly}\left(\frac{\log n}{\varepsilon}\right)$  rounds. It has later been shown that this can also be achieved deterministically [31]. The best known LOCAL MWM approximation algorithms are by Harris [35], who shows that a  $(1 - \varepsilon)$ -approximation can be computed in randomized time  $\tilde{O}\left(\frac{\log \Delta}{\varepsilon^3}\right) + \text{poly log}\left(\frac{\log \log n}{\varepsilon}\right)$  and in deterministic time  $\tilde{O}\left(\frac{\log^2 \Delta}{\varepsilon^4} + \frac{\log^* n}{\varepsilon}\right)$ . Those algorithms are based on computing large matchings in hypergraphs defined by paths of length  $O(1/\varepsilon)$  and they unfortunately cannot directly turned into efficient CONGEST algorithms. To the best of our knowledge, even for constant  $\varepsilon > 0$ , the only efficient CONGEST algorithms are for the unweighted maximum matching problem. Lotker, Patt-Shamir, and Pettie [49] give an algorithm to compute a  $(1 - \varepsilon)$ -approximation for the *unweighted* maximum matching problem in time only  $2^{O(1/\varepsilon)} \cdot \log n$  in the randomized CONGEST model. In [7] (full version), this was improved to  $2^{O(1/\varepsilon)} \cdot \frac{\log \Delta}{\log \log \Delta}$ . We obtain similar algorithms for the weighted matching problem. Obtaining a  $(1 - \varepsilon)$ -approximation

in poly( $\frac{\log n}{\varepsilon}$ ) CONGEST rounds is one of the key open questions in understanding the distributed complexity of maximum matching. Fischer, Mitrović, and Uitto [26] recently settled a related problem for unweighted matchings in the streaming model and in the latest version of their paper, they even obtain a poly( $\frac{\log n}{\varepsilon}$ )-round CONGEST algorithm for the unweighted matching problem.

**Distributed Complexity of Weighted Vertex Covers.** The first distributed constant-factor approximation algorithm for the minimum weighted vertex cover (MWVC) problem is due to Khuller, Vishkin, and Young [44]. They describe a simple deterministic algorithm to obtain a  $(2 + \varepsilon)$ -approximation for MWVC. The algorithm can directly be implemented in  $O(\log(n) \cdot \log(1/\varepsilon))$  rounds in the CONGEST model. The time for computing a  $(2 + \varepsilon)$ -approximation has subsequently been improved to  $O(\log(\Delta)/\text{poly}(\varepsilon))$  in [47] and to  $O(\log \Delta / \log \log \Delta)$  in [9–11] (with a very minor dependency on  $\varepsilon$  in [11]). Note that as for maximum matching, this dependency on  $\Delta$  is optimal for any constant-factor approximations [46]. The algorithm of [11] can also be used to compute a 2-approximate weighted vertex cover in time  $O(\log n)$ . Other polylogarithmic-time algorithms to compute 2-approximations for MWVC appeared in [34, 44, 45]. In the LOCAL model, one can use generic techniques from [30, 54] (or the techniques from this paper) to deterministically compute a  $(1 + \varepsilon)$ -approximate minimum weighted vertex cover in time poly( $\frac{\log n}{\varepsilon}$ ). Further, in [32], it was shown that even on bipartite graphs with maximum degree 3, there exists a constant  $\varepsilon_0 > 0$  such that computing a  $(1 + \varepsilon_0)$ -approximate (unweighted) vertex cover requires  $\Omega(\log n)$  rounds, even in the LOCAL model and even when using randomization. We generalize this result and show that for computing a  $(1 + \varepsilon)$ -approximation, one requires  $\Omega(\log(n)/\varepsilon)$  rounds. While for maximum matching, there are several CONGEST algorithms that achieve approximation ratios that are better than 1/2, for the minimum vertex cover problem, efficiently achieving an approximation ratio significantly below 2 in general graphs might be a hard problem. In this case, computing an exact solution even has a lower bound of  $\tilde{\Omega}(n^2)$  rounds in the CONGEST model and it is therefore basically as hard as any graph problem can be in this model [13]. To what extent we can achieve approximation ratios below 2 in the CONGEST model for variants of the minimum vertex cover problem is an interesting open question. There recently has been some progress. In [12], it is shown that the minimum vertex cover problem (and also the maximum matching problem) can be solved more efficiently if the optimal solution is small. In particular, if the size of an optimal vertex cover is at most  $k$ , a minimum vertex cover can be computed deterministically in time  $O(k^2)$  and a  $(2 - \varepsilon)$ -approximate solution can be computed deterministically in time  $O(k + (\varepsilon k)^2)$  (and slightly more efficiently with randomization). This was the first efficient CONGEST algorithm that achieves an approximation ratio below 2 for the minimum vertex cover problem for some graphs. In [23], it was shown that in bipartite graphs, a  $(1 + \varepsilon)$ -approximation can be computed in time poly( $\frac{\log n}{\varepsilon}$ ). One of the main results of this paper is a generalization of this result to the weighted vertex cover problem. Further, it has recently been shown that on the square graph  $G^2$ , it is possible to compute a  $(1 + \varepsilon)$ -approximate (unweighted) vertex cover in time  $O(n/\varepsilon)$  in the CONGEST model (on  $G$ ) [8].

## 1.1 Our Contributions

We next state our main contributions in detail. We prove new CONGEST upper bounds for approximating minimum weighted vertex cover and maximum weighted matching (MWM). We start by describing our results for the vertex cover problem. In [23], it was shown that in bipartite graphs, the unweighted vertex cover problem can be  $(1 + \varepsilon)$ -approximated in poly log( $\frac{\log n}{\varepsilon}$ ) time in the CONGEST model. The following theorem is a generalization of the result of [23] to weighted graphs.

► **Theorem 1.** *For every  $\varepsilon \in (0, 1]$ , there is a deterministic CONGEST algorithm to compute a  $(1 + \varepsilon)$ -approximation for the minimum weighted vertex cover problem in bipartite graphs in time  $\text{poly}\left(\frac{\log n}{\varepsilon}\right)$ .*

The next theorem shows that in graph families that are closed under taking (induced) subgraphs and in which we can efficiently compute large (or heavy) independent sets, we can efficiently approximate minimum (weighted) vertex cover with an approximation ratio that is better than 2.

► **Theorem 2.** *Let  $\mathcal{G}$  be a family of weighted graphs that is closed under taking induced subgraphs and such that for some  $\lambda \in (0, 1/2]$  and any  $n$ -node graph  $G = (V, E, w)$  of  $\mathcal{G}$ , there is a  $T_\lambda(n)$ -round CONGEST algorithm to compute an independent set  $S$  of weight  $w(S) \geq \lambda w(V)$ . Then, there is  $T_\lambda(n) + \text{poly}\left(\frac{\log n}{\varepsilon}\right)$ -round CONGEST algorithm to compute a  $(2 - 2\lambda + \varepsilon)$ -approximate weighted vertex cover for graphs of  $\mathcal{G}$ . If the independent set algorithm is deterministic, then also the vertex cover algorithm is deterministic.*

Note that the algorithm of Theorem 2 uses the bipartite vertex cover algorithm of Theorem 1 as a subroutine. Theorem 2 in particular implies that for graphs for which we can compute a coloring with a small number of colors, we can efficiently compute a non-trivial vertex cover approximation.

► **Corollary 3.** *Let  $\mathcal{G}$  be a family of weighted graphs such that for some non-negative integer  $C$ , for any  $n$ -node graph  $G = (V, E, w)$  of  $\mathcal{G}$ , there is a  $T_C(n)$ -round CONGEST algorithm to compute a vertex coloring of  $G$  with  $C$  colors. Then, there is a  $T_C(n) + \text{poly}\left(\frac{\log n}{\varepsilon}\right)$ -round CONGEST algorithm to compute a  $(2 - 2/C + \varepsilon)$ -approximation of the minimum weighted vertex cover problem for graphs of  $\mathcal{G}$ . If the coloring algorithm is deterministic, then also the vertex cover algorithm is deterministic.*

In order to efficiently compute an independent set  $S$  of weight  $w(S) \geq w(V)/C$  from a  $C$ -coloring, we need the graph to be of small diameter. However, by using standard clustering techniques (which we anyways need to apply also for our bipartite vertex cover algorithm), one can reduce the minimum (weighted) vertex cover problem on general  $n$ -node graphs to graphs of diameter  $\text{poly}\left(\frac{\log n}{\varepsilon}\right)$ . In particular, in graphs of arboricity  $a$ , we can (deterministically) compute a  $(2 + \varepsilon)a$ -coloring in time  $O(\log^3 a \cdot \log n)$  [29]. As a consequence, we get a deterministic  $\text{poly}\left(\frac{\log n}{\varepsilon}\right)$ -round CONGEST algorithm for computing a  $(2 - 1/a + \varepsilon)$ -approximation of minimum weighted vertex cover in graphs of arboricity  $a$ .

In addition to our CONGEST algorithms for approximating minimum weighted vertex cover, we also provide new CONGEST algorithms for approximating maximum weighted matching. The following theorem can be seen as a generalization of Theorem 3.15 in [49] and of Theorem B.12 in [7] (full version).

► **Theorem 4.** *For every  $\varepsilon, \delta \in (0, 1]$ , there is a randomized CONGEST algorithm that with probability at least  $1 - \delta$  computes a  $(1 - \varepsilon)$ -approximation to the maximum weighted matching problem in  $2^{O(1/\varepsilon)} \cdot (\log(W\Delta) + \log^2 \Delta + \log^* n) \cdot \log^3(1/\delta)$  rounds. Further, there is a deterministic CONGEST algorithm to compute a  $(1 - \varepsilon)$ -approximation for the minimum weighted matching problem in time  $2^{O(1/\varepsilon)} \cdot \text{poly} \log n$ .*

Note that except for the  $\log^* n$  term, for constant error probability  $\delta$ , the round complexity of our randomized algorithm is independent of the number of nodes  $n$ . Moreover, for constant  $\varepsilon$  and  $\delta$ , in bounded-degree graphs with bounded weights, the round complexity of the randomized algorithm is only  $O(\log^* n)$ . For unweighted matchings, a round complexity that is completely independent of  $n$  was obtained by [7]. Interestingly, Göös and Suomela in [32]

showed that such a result is not possible for the minimum vertex cover problem, even in the LOCAL model. They show that even for bipartite graphs of maximum degree 3, there exists a constant  $\varepsilon_0 > 0$  such that any randomized distributed  $(1 + \varepsilon_0)$ -approximation algorithm for the (unweighted) minimum vertex cover problem requires  $\Omega(\log n)$  rounds. As our last contribution, we generalize the result of [32] to computing  $(1 + \varepsilon)$ -approximate solutions for any sufficiently small  $\varepsilon > 0$ .

► **Theorem 5.** *There exists a constant  $\varepsilon_0 > 0$  such that for every  $\varepsilon \in (0, \varepsilon_0]$ , any randomized LOCAL model algorithm to compute a  $(1 + \varepsilon)$ -approximation for the (unweighted) minimum vertex cover problem in bipartite graphs of maximum degree 3 requires  $\Omega\left(\frac{\log n}{\varepsilon}\right)$  rounds.*

Theorem 5 is obtained by a relatively simple reduction to the  $(1 + \varepsilon_0)$ -approximation lower bound proven in [32] for bipartite graphs of maximum degree 3. We note that if we only require the approximation factor to hold in expectation, as discussed at the end of Section 3, in the LOCAL model the theorem is tight even for general graphs and even for the weighted vertex cover problem.

**Organization of the paper.** The remainder of the paper is organized as follows. In Section 2, we define the communication model and we introduce all the necessary mathematical notations and definitions. In Section 4, we give an overview over our minimum weighted vertex cover algorithms and in Section 5, we discuss the most important ideas to derive our maximum weighted matching result. Many of the technical details regarding our algorithms have to be omitted in this extended abstract and they only appear in the full version [22] of this paper. In Section 6, we prove the lower bound on approximating vertex cover in bipartite graphs in the LOCAL model. Finally, in Appendix A, we give some basic algorithmic tools that we need for our algorithms.

## 2 Model and Preliminaries

### 2.1 Mathematical Notation

Let  $G = (V, E, w)$  be an undirected weighted graph, where  $w$  is a non-negative weight function. We will use node and edge weights in the paper and depending on the context, we will use  $w$  to assign weights to nodes and/or edges. Generally for a set  $X$  of nodes and/or edges, we use  $w(X)$  to denote the sum of the weights of all nodes/edges in  $X$ . For example, if we have node weights,  $w(V)$  denotes the sum of the weights of all the nodes. Throughout the paper, we assume that all weights are integers that are polynomially bounded in the number of nodes of the graph. However, as long as we can communicate a single weight in a single message, all our algorithms can be adapted to also work at no significant additional asymptotic cost for more general weight assignments. We further use the following notation for graphs. For a node  $v \in V$ , we use  $N(v) \subseteq V$  to denote the set of neighbors of  $v$  and we use  $E(v) \subseteq E$  to denote the set of edges that are incident to  $v$ .

For a graph  $G = (V, E)$ , the *bipartite double cover* is defined as 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  $\{(u, 1), (v, 0)\}$ . Moreover, if  $G$  is a weighted graph with weight function  $w$ , we assume that the bipartite double cover  $G_2$  is also weighted and that the corresponding nodes and/or edges have the same weight as in  $G$ . That is, in case of node weights, for every  $u \in V$ , we define  $w((u, 0)) = w((u, 1)) = w(u)$  and in case of edge weights, for every  $\{u, v\} \in E$ , we define  $w(\{(u, i), (v, 1 - i)\}) = w(\{u, v\})$  for  $i \in \{0, 1\}$ .

## 2.2 Problem Definitions

In this paper, we consider the *minimum weighted vertex cover (MWVC)* and the *maximum weighted matching (MWM)* problems. Formally, in the MWVC problem, we are given a weighted graph  $G = (V, E, w)$  with positive node weights. A vertex cover of  $G$  is a set  $S \subseteq V$  of nodes such that for every edge  $\{u, v\} \in E$ ,  $S \cap \{u, v\} \neq \emptyset$ . The goal of the MWVC problem is to find a vertex cover  $S$  of minimum total weight  $w(S)$ . In the MWM problem, we are given a weighted graph  $G = (V, E, w)$  with positive edge weights. A matching of  $G$  is a set  $M \subseteq E$  of edges such that no two edges in  $M$  are adjacent. The goal of the MWM problem is to find a matching  $M$  of maximum total weight  $w(M)$ . The unweighted versions of the two problems are closely related to each other as their natural fractional linear programming (LP) relaxations are duals of each other. In the paper, we will also use the fractional relaxation of the MWVC problem and its dual problem. In the fractional MWVC problem on  $G$ , every node  $u \in V$  is assigned a value  $x_u \in [0, 1]$  such that for every edge  $\{u, v\} \in E$ ,  $x_u + x_v \geq 1$  and such that the sum  $\sum_{u \in V} w(u) \cdot x_u$  is minimized. The dual LP of this problem, which for a given weight function  $w$ , we in the following call the *fractional  $w$ -matching* problem, is defined as follows. Every edge  $e \in E$  is assigned a (fractional) value  $y_e \geq 0$  such that for every node  $u \in V$ , we have  $\sum_{e: u \in e} y_e \leq w(u)$  and such that the sum  $\sum_{e \in E} y_e$  is maximized. We use the vector  $\mathbf{y}$  to refer to a fractional solution that assigns a fractional value  $y_e$  to every edge. Further for convenience, for a set of edges  $F$ , we also use the short notation  $y(F) := \sum_{e \in F} y_e$ . LP duality directly implies that the value of any fractional  $w$ -matching cannot be larger than the weight of any vertex cover:

► **Lemma 6.** *Let  $G = (V, E, w)$  be a node-weighted graph and let  $\mathbf{y}$  be a fractional  $w$ -matching of  $G$ . It then holds that  $y(E) \leq w(S)$  for every vertex cover  $S$  of  $G$ .*

**Proof.** We have

$$w(S) = \sum_{v \in S} w(v) \geq \sum_{v \in S} \sum_{e: v \in e} y_e \geq \sum_{e \in E} y_e = y(E).$$

The first inequality holds because the values  $y_e$  form a valid fractional  $w$ -matching and the second inequality holds because  $S$  is a vertex cover. ◀

The *approximation ratio* of an approximation algorithm for the MWVC or MWM problem is defined as the worst-case ratio between the total weight of a vertex cover or matching computed by the algorithm over the total weight of an optimal vertex cover or matching. That is, we define the approximation ratio such that it is  $\geq 1$  for minimization and  $\leq 1$  for maximization problems.

## 2.3 Low-Diameter Clustering

Many of our algorithms have some components that require global communication in the network. In order to achieve a polylogarithmic round complexity, we therefore need a graph with polylogarithmic diameter. We achieve this by applying standard clustering techniques. Formally, we use the clusterings as in [23] described in the following. Let  $G = (V, E, w)$  be a weighted graph with non-negative node and edge weights. A *clustering* of  $G$  is a collection  $\{S_1, \dots, S_k\}$  of disjoint node sets  $S_i \subseteq V$ . For  $\lambda \in [0, 1]$ , a clustering  $\{S_1, \dots, S_k\}$  is called  $\lambda$ -dense if the total weight of all nodes and edges in the induced subgraphs  $G[S_i]$  for  $i \in \{1, \dots, k\}$  is at least  $\lambda(w(V) + w(E))$ . Further, for an integer  $h \geq 1$ , a clustering  $\{S_1, \dots, S_k\}$  is called  $h$ -hop separated if for any two clusters  $S_i$  and  $S_j$  ( $i \neq j$ ) and any pair of nodes  $(u, v) \in S_i \times S_j$ , we have  $d_G(u, v) \geq h$ , where  $d_G(u, v)$  denotes the hop-distance

between  $u$  and  $v$ . Further for two integers  $c, d \geq 1$ , a clustering  $\{S_1, \dots, S_k\}$  is defined to be  $(c, d)$ -routable if we are given a collection of  $T_1, \dots, T_k$  trees in  $G$  such that for each  $i \in \{1, \dots, k\}$ , the nodes  $S_i$  are contained in  $T_i$ , every tree  $T_i$  has diameter at most  $d$ , and every edge  $e \in E$  of  $G$  is contained in at most  $c$  of the trees  $T_1, \dots, T_k$ . Note that this implies that each cluster of a  $(c, d)$ -routable clustering has weak diameter at most  $d$  and if the nodes of  $T_i$  are all contained in  $S_i$ , it implies that the strong diameter of cluster  $S_i$  is at most  $d$ .

## 2.4 Communication Model

Throughout the paper, we assume a standard synchronous message passing model on graphs. That is, the network is modeled as an undirected  $n$ -node graph  $G = (V, E)$ . Each node is equipped with a unique  $O(\log n)$ -bit identifier. The nodes  $V$  communicate in synchronous rounds over the edges  $E$  such that in each round, every node can send an arbitrary message to each of its neighbors. Internal computations at the nodes are free. Initially, the nodes do not know anything about the topology of the network. When computing a vertex cover or a matching, at the end of the algorithm, every node must output if it is in the vertex cover or which of its edges belong to the matching. The time or round complexity of an algorithm is defined as the number of rounds that are needed until all nodes terminate. If the size of the messages is not restricted, this model is known as the LOCAL model [53]. In the more restrictive CONGEST model, all messages must consist of at most  $O(\log n)$  bits [53]. In several of our algorithms, we will first compute a clustering as defined above in Section 2.3 and we afterwards run CONGEST algorithms on the clusters. If we are given a  $(c, d)$ -routable clustering, we are only guaranteed that the diameter of each cluster  $S_i$  is small if we add the nodes and edges of the tree  $T_i$  to the cluster. For running our algorithms on individual clusters, we therefore need an extension of the classic CONGEST model, which has been introduced as the SUPPORTED CONGEST model in [27, 55]. In the SUPPORTED CONGEST model, we are given two graphs, a communication graph  $H = (V_H, E_H)$  and a logical graph  $G = (V, E)$ , which is a subgraph of  $H$ . When solving a graph problem such as MWVC or MWM in the SUPPORTED CONGEST model, we need to solve the graph problem on the logical graph  $G$ , we can however use CONGEST algorithms on the underlying communication graph  $H$  to do so. Note that if we are given a  $(c, d)$ -routable clustering, we can define  $G_i := G[S_i]$  and  $H_i$  as the union of the graph  $G_i$  and the tree  $T_i$  for each cluster and we can then in parallel run 1 round of a SUPPORTED CONGEST algorithm on each cluster in  $c$  CONGEST rounds on  $G$ .

## 3 Reducing to Small Diameter

We start the presentation of our results by describing how we can use clusterings to reduce the problem of approximating MWVC or MWM on general graphs to the case of approximating the same problems on graphs of small diameter. The high-level idea that we use to reduce the diameter is a classic one. We find a disjoint and sufficiently separated collection of low-diameter clusters such that only a small fraction of the graph is outside of the clusters (see, e.g., [6, 48, 51, 53, 54] for constructions of such clusterings). We then compute a good approximation for a given problem inside each cluster and we use a coarse approximation to extend the solution to the parts of the graph outside of the clusters. If we want this to work in general graphs, we have to adapt the standard clustering constructions such that the part of the graph that is outside of the clusters contains only a *small fraction of a solution to the actual problem* that we want to approximate, rather than simply a small fraction of the number of nodes and/or edges of the graph. A generic way to achieve this in the LOCAL

model has been described in [30] and a method that can also be used in the CONGEST model has recently been described in [23] for the unweighted minimum vertex cover problem. The following theorem shows how to extend the approach of [23] to work also for weighted vertex cover and matching. The theorem shows that at the cost of a  $(1 + \varepsilon)$ -factor, the problems of approximating MWVC and MWM can efficiently be reduced to approximating the problems in the SUPPORTED CONGEST model with a small-diameter communication graph. We here use the SUPPORTED CONGEST model because we compute clusters by adapt a network decomposition algorithm of [54], which only creates clusters of weak diameter. Due to lack of space, the proof of the following theorem is omitted and deferred to the full version of this paper [22].

► **Theorem 7 (Diameter Reduction).** *Let  $T_{\text{SC}}^\alpha(n, D)$  be the time required for computing an  $\alpha$ -approximation for the MWVC or the MWM problem in the SUPPORTED CONGEST model with a communication graph of diameter  $D$ . Then, for every  $\varepsilon \in (0, 1]$ , there is a poly  $(\frac{\log n}{\varepsilon}) + O(\log n \cdot T_{\text{SC}}^\alpha(n, O(\frac{\log^3 n}{\varepsilon})))$ -round CONGEST algorithm to compute a  $(1 - \varepsilon)\alpha$ -approximation of MWM or an  $(1 + \varepsilon)\alpha$ -approximation of MWVC in the CONGEST model. If the given SUPPORTED CONGEST model algorithm is deterministic, then the resulting CONGEST model algorithm is also deterministic. Also, if we want to solve MWVC or MWM in the CONGEST model on a bipartite graph, then it is sufficient to have a SUPPORTED CONGEST model algorithm that works for a bipartite communication (and thus also logical) graph.*

► **Remark.** We note that by using a variant of the randomized clustering algorithm of Miller, Peng, and Xu [51], one can compute a  $(1, O(\frac{\log n}{\varepsilon}))$ -routable, 3-hop separated clustering with expected density  $1 - \varepsilon$  in time  $O(\frac{\log n}{\varepsilon})$  (also cf. [23]). In combination with the argument in the above proof, this in particular implies that it in  $O(\frac{\log n}{\varepsilon})$  rounds in the LOCAL model, it is possible to compute weighted matchings and weighted vertex covers with expected approximation ratios  $1 - \varepsilon$  and  $1 + \varepsilon$ , respectively.

## 4 Weighted Vertex Cover Algorithms

We next describe our results on approximating the minimum weighted vertex cover. We start by describing a distributed approximation scheme for bipartite graphs.

### 4.1 Basic Weighted Vertex Cover Algorithm

It is well-known that in bipartite graphs, the size of a maximum matching is equal to the size of a minimum vertex cover (this is known as König's theorem [16, 42]). The theorem was also independently discovered in [20] by Egerváry, who also more generally proved that on node-weighted bipartite graphs, the total value of an optimal (fractional)  $w$ -matching is equal to the weight of a minimum weighted vertex cover, where a fractional  $w$ -matching of a node-weighted graph  $G = (V, E, w)$  is an assignment of fractional values  $y_e \geq 0$  to all edges such that the edges of each node  $v$  sum up to at most  $w(v)$ . In both cases, the theorem can be proven in a constructive way. Given a maximum matching or more generally a maximum fractional  $w$ -matching, there is a simple (and efficient) algorithm to compute a vertex cover of the same size or weight.

Moreover as shown in [24], if we are given a good approximate matching or  $w$ -matching with some additional properties, the constructive proof of [20, 42] can be adapted to obtain a good approximate (weighted) vertex cover. For the unweighted case, this method is at the core of the CONGEST model bipartite vertex cover algorithms of [23]. We next describe how to use this technique to approximate MWVC in the CONGEST model.



Let  $G = (V, E, w)$  be a node-weighted graph, where  $w$  is a non-negative node weight function. For a node  $v \in V$  and a given fractional  $w$ -matching  $y_e$  for  $e \in E$ , we define the *slack* of  $v$  as  $s(v) := w(v) - \sum_{e:v \in e} y_e$ . Given a fractional  $w$ -matching  $y_e$  for  $e \in E$ , an *augmenting path* is an odd length path  $P = (v_0, \dots, v_{2k+1})$  where the end nodes  $v_0$  and  $v_{2k+1}$  have positive slack  $s(v_0), s(v_{2k+1}) > 0$ , and for  $i \in \{1, 2, \dots, k\}$ , each even edge  $e = (v_{2i-1}, v_{2i})$  has a positive fractional value  $y_e > 0$ . Assume that we are given a bipartite graph  $G = (A \cup B, E, w)$  and that we are given a fractional  $w$ -matching  $\mathbf{y}$  of  $G$  such that there are no augmenting paths of length at most  $2k - 1$  for some integer  $k \geq 1$ . We can then apply the following algorithm to compute a vertex cover  $S$  of  $G$ . The algorithm computes disjoint sets  $A_0, A_1, \dots, A_k \subseteq A$  and disjoint sets  $B_1, \dots, B_k \subseteq B$ . In the following for a set of nodes  $X$ , we use  $N_{y>0}(X)$  to denote the set of nodes that are connected to a node in  $X$  through an edge  $e$  with  $y_e > 0$ .

■ **Algorithm 1** Basic Approximate Weighted Vertex Cover Algorithm.

1. Define  $A_0 := \{v \in A : s(v) > 0\}$  as the nodes in  $A$  with positive slack and  $B_0 := \emptyset$ .
2. For every  $i \in \{1, \dots, k\}$ , define  $B_i := \left\{ v \in B \setminus \bigcup_{j=0}^{i-1} B_j : v \in N(A_{i-1}) \right\}$ .
3. For every  $i \in \{1, \dots, k\}$ , define  $A_i := \left\{ v \in A \setminus \bigcup_{j=0}^{i-1} A_j : v \in N_{y>0}(B_i) \right\}$ .
4. Define  $i^* := \arg \min_{i \in \{1, \dots, k\}} w(B_i)$ .
5. Output  $S := \bigcup_{i=1}^{i^*} B_i \cup (A \setminus \bigcup_{i=0}^{i^*-1} A_i)$ .

That is, the sets  $A_0, B_1, A_1, B_2, A_2 \dots$  are the levels of a BFS traversal of the graph starting at the nodes in  $A_0$  and where steps from  $B_i$  to  $A_i$  have to be over an edge  $e$  with positive fractional value  $y_e > 0$ .

► **Lemma 8.** *Given a weighted bipartite graph  $G = (A \cup B, E, w)$ , an integer  $k \geq 1$ , and a fractional  $w$ -matching of  $G$  with no augmenting paths of length at most  $2k - 1$ , the above algorithm computes a  $(1 + 1/k)$ -approximate weighted vertex cover of  $G$ . Further, the above algorithm can be deterministically implemented in  $O(D + k)$  rounds in the SUPPORTED CONGEST model if the communication graph is also bipartite and has diameter at most  $D$ .*

**Proof.** We first prove that  $S$  is a valid vertex cover. For this, we need to show that there is no edge between  $A \setminus S$  and  $B \setminus S$ . We have  $A \setminus S = \bigcup_{j=0}^{i^*-1} A_j$  and  $B \setminus S = \bigcup_{j=i^*+1}^k B_j \cup \bar{B}$ , where  $\bar{B} = B \setminus \bigcup_{i=1}^k B_i$ . We therefore need to show that there cannot be an edge between a set  $A_j$  for  $j < i^*$  and  $\sum_{j=i^*+1}^k B_j \cup \bar{B}$ . However, by construction, all neighbors of nodes in  $A_j$  are in  $B_1, \dots, B_{i^*}$  and we can thus conclude that  $S$  is a vertex cover.

We next show that  $S$  is a  $(1 + 1/k)$ -approximate weighted vertex cover of  $G$ . First observe that for all  $i \in \{1, 2, \dots, k\}$ , all nodes  $v$  in  $B_i$  are saturated nodes with zero slack, i.e.,  $w(v) = \sum_{e \in E(v)} y_e$ . From the construction of the sets  $A_0, B_1, A_1, \dots$ , we otherwise get an augmenting path of length at most  $2k - 1$ . Moreover since the sets of edges incident to the sets  $B_1, \dots, B_k$  are disjoint, the sum of the fractional values of all those edges is at most  $y(E) = \sum_{e \in E} y_e$ . By the choice of  $i^*$ , we therefore know  $w(B_{i^*}) = \sum_{v \in B_{i^*}} \sum_{e \in E(v)} y_e \leq y(E)/k$ . Moreover, from the BFS construction of the sets  $A_i$  and  $B_i$  it follows that the only edges  $e$  with  $y_e > 0$  for which both nodes are in  $S$  are edges that are incident to nodes in  $B_{i^*}$ . We can therefore conclude that  $w(S) \leq y(E) + w(B_{i^*}) \leq (1 + 1/k) \cdot y(E)$ . The approximation ratio now follows from LP duality (i.e., from Lemma 6).

Finally, we discuss how the above algorithm can be efficiently implemented in time  $O(D + k)$  rounds in the SUPPORTED CONGEST model, where  $D$  is the diameter of the communication graph  $H$ . In  $O(D)$  rounds, one can compute a BFS spanning tree of  $H$  and use it to compute the bipartition of the nodes of  $H$  and thus of  $G$  into sets  $A$  and  $B$ . Then in  $O(k)$  rounds, the algorithm constructs the sets  $A_i$  and  $B_i$  for  $i \in \{1, 2, \dots, k\}$  by running the first  $2k$  iterations of parallel BFS on  $G$  starting from set  $A_0$  (where edges from  $B_i$  to  $A_i$  need to have positive fractional values). Finally in another  $O(D + k)$  rounds, we use the precomputed BFS spanning tree on  $H$  and a standard pipelining scheme for the root to compute the weights of all the sets  $B_i$ , determine the index  $i^*$  of the smallest weight amongst them, and broadcast it to all nodes in  $G$ . ◀

We remark that although the above algorithm only requires a BFS traversal from all nodes in  $A_0$  for  $k$  levels, the algorithm still requires time  $D$  for two reasons. First, the algorithm needs to know the bipartition  $A \cup B$  of  $G$  and computing the bipartition requires  $\Omega(D)$  time. Second, even if the bipartition is given initially, the algorithm still needs  $\Omega(D)$  time to determine the optimal level  $i^*$ .

## 4.2 Getting Rid of Short Augmenting Paths

The basic MWVC algorithm described in Section 4.1 basically converts a good approximation of fractional  $w$ -matching into a good MWVC approximation. However the algorithm needs a fractional  $w$ -matching with the additional property that there are no short augmenting paths. For the unweighted setting, there exists a randomized CONGEST algorithm to compute an integral matching with no short augmenting paths [49]. It is however not clear if the algorithm of [49] can be generalized to the fractional  $w$ -matching problem. Further, even in the unweighted case, we do not have a deterministic CONGEST algorithm to compute such a matching. As in the deterministic, unweighted MVC algorithm of [23], we therefore use a different approach. In the unweighted setting, we first compute a  $(1 - \delta)$ -approximate matching that can potentially have short augmenting paths. We then get rid of those short augmenting paths by removing at least one unmatched node or both nodes of a matching edge from the graph. The removed nodes are at the end added to the vertex cover to make sure that all edges are covered. The selection of a smallest possible number of unmatched nodes and matching edges that hit all short augmenting paths can be phrased as a minimum set cover problem, which we can approximate efficiently in the CONGEST model. In the weighted case, we use a generalization of this approach. Because of the weights, the process and its analysis however becomes more subtle and we have to be more careful.

Assume that we want to compute a  $(1 + O(\varepsilon))$ -approximate weighted vertex cover for a node-weighted bipartite graph  $G = (A \cup B, E, w)$ . In a first step, we compute a  $(1 - \delta)$ -approximate fractional  $w$ -matching  $\mathbf{y} := \{y_e : e \in E\}$  of  $G$  for some parameter  $\delta \ll \varepsilon$ . We can do this efficiently by using Theorem 12. The fractional  $w$ -matching  $\mathbf{y}$  however might have short augmenting paths. In a second step, we then convert our graph  $G$  and the fractional  $w$ -matching  $\mathbf{y}$  such that we obtain an instance with no short augmenting paths and that we can thus apply Lemma 8. More concretely, we decrease some of the weights  $w(v)$  and some of the fractional values  $y_e$  such that for the resulting weights  $w'(v)$  and the resulting  $w'$ -matching  $\mathbf{y}'$ , the graph  $G$  has no short augmenting paths and such that a  $(1 + \varepsilon)$ -approximate weighted vertex cover of  $G$  with the weights  $w'(v)$  is a  $(1 + O(\varepsilon))$ -approximate weighted vertex cover of  $G$  for the original weights. We next describe the main ideas of this transformation.

Formally, the conversion can be defined by a set  $X \subseteq \{v \in A \cup B : s(v) > 0\}$  of nodes with positive slack and a set  $F \subseteq \{e \in E : y_e > 0\}$  of edges with positive fractional value. The new fractional values  $y'_e$  and the new weights  $w'(v)$  are defined as follows:

$$y'_e := \begin{cases} 0 & \text{if } e \in F \\ y_e & \text{if } e \notin F \end{cases}, \quad w'(v) := \begin{cases} w(v) - s(v) - y(E(v) \cap F) & \text{if } v \in X \\ w(v) - y(E(v) \cap F) & \text{if } v \notin X \end{cases} \quad (1)$$

► **Lemma 9.** *Any augmenting path of  $G$  w.r.t. the weight function  $w'$  and the fractional  $w'$ -matching  $\mathbf{y}'$  is also an augmenting path w.r.t. the original weight function  $w$  and the original fractional  $w$ -matching  $\mathbf{y}$ .*

**Proof.** First note that whenever we decrease a value  $y_e$  to  $y'_e < y_e$  for some edge  $e = \{u, v\}$ , we also decrease the weights of  $u$  and  $v$  by the same amount. Therefore, the slack of a node  $v$  w.r.t.  $w'$  and  $\mathbf{y}'$  cannot be larger than the slack of  $v$  w.r.t.  $w$  and  $\mathbf{y}$ . Therefore any odd-length path  $P$  in  $G$  that starts and ends at a node with positive slack w.r.t.  $w'$  and  $\mathbf{y}'$  also starts and ends at a node with positive slack w.r.t.  $w$  and  $\mathbf{y}$ . Further, if every even edge  $e$  of such a path  $P$  has a positive fractional value  $y'_e > 0$ , then it also holds that  $y_e > 0$ . ◀

Note that the definition of  $w'$  and  $\mathbf{y}'$  in (1) guarantees that all nodes  $v \in X$  have slack 0 w.r.t. the new weights  $w'$  and the new fractional values  $\mathbf{y}'$ . Consider some augmenting path  $P = (v_0, \dots, v_{2\ell+1})$  of  $G$  w.r.t.  $w$  and  $\mathbf{y}$ . If we have  $v_0 \in X$  or  $v_{2\ell+1} \in X$  or if we have  $e \in F$  for one of the even edges  $\{v_{2i-1}, v_{2i}\}$  of  $P$ , then  $P$  is not an augmenting path of  $G$  w.r.t.  $w'$  and  $\mathbf{y}'$ . In order to get rid of all short augmenting paths, we therefore need to choose one of the end nodes or one of the even edges of each such path and add them to  $X$  or  $F$ . We can then use Lemma 8 to efficiently compute a good vertex cover approximation for  $G$  w.r.t. the new weights  $w'$ . The quality of such a vertex cover w.r.t. the original weights  $w$  can be bounded as follows.

► **Lemma 10.** *Let  $S^*$  be an optimal weighted vertex cover of  $G = (V, E)$  w.r.t. the weights  $w$  and assume that for some  $\alpha \geq 1$ ,  $S$  is an  $\alpha$ -approximate weighted vertex cover of  $G$  w.r.t. the weights  $w'$ . It then holds that*

$$w(S) \leq \alpha \cdot w(S^*) + s(X) + y(F), \quad \text{where } s(X) := \sum_{v \in X} s(v).$$

**Proof.** Let  $S'$  be an optimal weighted vertex cover of  $G$  w.r.t. the weights  $w'$ . Because any vertex cover must contain at least one node of every edge in  $F$ , we have  $w'(S') \leq w(S^*) - y(F)$ . We therefore have

$$w(S) \leq w'(S) + s(X) + 2y(F) \leq \alpha w'(S') + s(X) + 2y(F) \leq \alpha w(S^*) + s(X) + y(F). \quad \blacktriangleleft$$

In order to optimize the approximation, we thus need to determine the sets  $X$  and  $F$  such that  $s(X) + y(F)$  is as small as possible and such that we “cover” all short augmenting paths. The problem of finding the best possible sets  $X$  and  $F$  can naturally be phrased as a weighted set cover problem. One can further show that if the parameter  $\delta$  that determines the quality of the fractional  $w$ -matching  $\mathbf{y}$  is chosen sufficiently small (but still as  $\delta = \text{poly}(\varepsilon / \log n)$ ), even a logarithmic approximation to this weighted set cover instance guarantees that  $s(X) + y(F) = O(\varepsilon \cdot w(S^*))$ . Further, by sequentially going over the possible short augmenting path lengths and adapting existing algorithms of [49] and [23], a variant of the greedy algorithm for this weighted set cover instance can be implemented efficiently in the CONGEST model on  $G$ . Given an efficient algorithm to find appropriate sets  $X$  and  $F$ , the claim of Theorem 1 then follows almost immediately by combining with Theorem 7 and Lemma 8. The technical details appear in the full version [22].

### 4.3 Generalization to Non-Bipartite Graphs

For approximating the MWVC problem in general graphs, we employ a standard approach that is for example described in [37]. We describe the details for completeness. The minimum fractional (weighted) vertex cover problem is the natural LP relaxation of the minimum (weighted) vertex cover problem. That is, a fractional vertex cover of a graph  $G = (V, E)$  is an assignment of values  $x_v \in [0, 1]$  to all nodes such that for every edge  $\{u, v\}$ ,  $x_u + x_v \geq 1$ . While the integrality gap of the (weighted and unweighted) vertex cover can be arbitrarily close to 2, it is well-known that there are optimal fractional solutions that are *half-integral*. That is, there are optimal fractional solutions such that for all nodes  $v$ ,  $x_v \in \{0, 1/2, 1\}$ . Given such a fractional solution, let  $S_x$  be the set of nodes  $v$  with  $x_v = x$  for  $x \in \{0, 1/2, 1\}$ . Let  $I_{1/2}$  be an independent set of the induced subgraph  $G[S_{1/2}]$  of the half-integral nodes. It is not hard to see that  $S := S_1 \cup S_{1/2} \setminus I_{1/2}$  is a vertex cover of  $G$  and if  $w(I_{1/2}) \geq \lambda \cdot w(S_{1/2})$ , then the set  $S$  is a  $(2 - 2\lambda)$ -approximate solution for the MWVC problem on  $G$  with weights  $w$ . If the half-integral fractional solution is only an  $\alpha$ -approximate fractional weighted vertex cover, the resulting approximation is  $(2 - 2\lambda) \cdot \alpha$ .

The core to proving Theorem 2 is therefore to first compute a  $(1 + \varepsilon)$ -approximate half-integral fractional solution for a given weighted graph  $G = (V, E, w)$ . The following lemma uses a standard approach to achieve this by first computing an approximate solution to the (integral) MWVC problem for the bipartite double cover  $G_2$  of  $G$  (see definition in Section 2).

► **Lemma 11.** *Let  $G = (V, E, w)$  be a weighted graph and let  $G_2 = (V_2, E_2)$  be the bipartite double cover of  $G$ , where each node  $(v, i) \in V_2$  for  $v \in V$  gets assigned weight  $w(v)$ . Let  $S$  be a vertex cover of  $G_2$  and define  $x_v := |\{(v, 0), (v, 1)\} \cap S|/2$  for every  $v \in V$ . If  $S$  is an  $\alpha$ -approximate weighted vertex cover of  $G_2$  for some  $\alpha \geq 1$ , then  $\mathbf{x} = \{x_v : v \in V\}$  is a half-integral  $\alpha$ -approximate fractional weighted vertex cover of  $G$ .*

**Proof.** We first show that the weight  $\sum_{v \in V} w(v) \cdot z_v$  of an optimal fractional weighted vertex cover  $\mathbf{z}$  of  $G = (V, E, w)$  is exactly half the weight of an optimal fractional weighted vertex cover of  $G_2$  with weights assigned as defined by the claim of the lemma. To see this, let  $\mathbf{z}$  be a fractional vertex cover of  $G$ . We can then get a valid fractional vertex cover of  $G_2$  by setting  $z_{(v,0)} = z_{(v,1)} = z_v$  for every node  $v \in V$  of  $G$  and the corresponding nodes  $(v, 0)$  and  $(v, 1)$  in  $G_2$ . In the other direction, for a fractional vertex cover  $\mathbf{z}$  in  $G_2$ , we obtain a fractional vertex cover of  $G$  by setting  $z_v := (z_{(v,0)} + z_{(v,1)})/2$ . Note that because  $G_2$  is a bipartite graph, an optimal (integral) weighted vertex cover of  $G_2$  is also an optimal fractional weighted vertex cover of  $G_2$  and therefore the vertex cover  $S$  is an  $\alpha$ -approximate fractional weighted vertex cover of  $G_2$ . The fractional vertex cover  $\mathbf{x}$  of  $G$  as given by the lemma statement is of half the weight of  $S$  in  $G_2$  and it therefore is an  $\alpha$ -approximate fractional weighted vertex cover of  $G$ . Clearly,  $\mathbf{x}$  is half-integral. ◀

We can now prove Theorem 2 and Corollary 3.

**Proof of Theorem 2.** As discussed, given a weighted graph  $G = (V, E, w)$ , we first construct the bipartite double cover  $G_2 = (V_2, E_2, w)$ , where the weight function  $w$  is extended to  $G_2$  in the obvious way (and as described in Section 4.3). Note that since every node of  $G$  only needs to simulate 2 nodes in  $G_2$  and every edge of  $G$  is replaced by 2 edges in  $G_2$ , CONGEST algorithms on  $G_2$  can be simulated on  $G$  with only constant overhead. By using Theorem 1, we can therefore compute a  $(1 + \varepsilon)$ -approximation of MWVC on  $G_2$  in time  $\text{poly}\left(\frac{\log n}{\varepsilon}\right)$ . By Lemma 11, this can directly be turned into a half-integral  $(1 + \varepsilon)$ -approximate fractional weighted vertex cover of  $G$ . Let  $S_1$  be the nodes of  $G$  that have a fractional vertex cover

value of 1 and let  $S_{1/2}$  be the nodes of  $G$  that have a fractional vertex cover value of  $1/2$ . Assume further that  $I_{1/2}$  is an independent set of the induced subgraph  $G[S_{1/2}]$ . Clearly,  $S := S_1 \cup S_{1/2} \setminus I_{1/2}$  is a vertex cover of  $G$ . If the weight  $w(I_{1/2})$  is  $w(I_{1/2}) \geq \lambda \cdot w(S_{1/2})$ , then the weight of the vertex cover  $S$  can be bounded as

$$\begin{aligned} w(S) &= w(S_1) + w(S_{1/2}) - w(I_{1/2}) \\ &\leq w(S_1) + (1 - \lambda) \cdot w(S_{1/2}) \\ &\leq (2 - 2\lambda) \cdot \left( w(S_1) + \frac{1}{2} \cdot w(S_{1/2}) \right) \\ &\leq (2 - 2\lambda) \cdot (1 + \varepsilon) \cdot w(S^*), \end{aligned}$$

where  $S^*$  is an optimal weighted vertex cover of  $G$ . The claim of the theorem now follows. ◀

**Proof of Corollary 3.** By Theorem 7, we can first reduce the diameter of the communication graph in time  $\text{poly}\left(\frac{\log n}{\varepsilon}\right)$  while only losing a  $(1 + \varepsilon/2)$ -factor in the approximation. Now assume that we are given a weighted graph  $G$  with a  $C$ -coloring, that we have a communication graph  $H$  of diameter  $D$ , and that we can use the SUPPORTED CONGEST model. In time  $O(D)$  in the SUPPORTED CONGEST model, we can then use the  $C$ -coloring to compute an independent set of weight at least  $w(V(G))/C$  by just keeping the heaviest color class. The corollary therefore follows directly by combining Theorem 7 and Theorem 2. ◀

## 5 Weighted Matching Approximation

We provide a randomized and a deterministic CONGEST algorithm to approximate the MWM problem. Both algorithms are based on the following key idea that was developed for the unweighted maximum matching problem by Lotker, Patt-Shamir, and Pettie [49] and that was extended by Bar-Yehuda et al. [7]. We iteratively adapt an initial matching  $M_0$  of a given weighted graph  $G = (V, E, w)$  as follows. We repeatedly sample bipartite subgraphs of  $G$  and we then find a good matching on this bipartition to improve the matching of  $G$ . In [7, 49], the matching is improved by finding short augmenting paths in the sampled bipartite subgraph and augmenting the existing matching along those paths. While, as discussed below, also for maximum weighted matching, it is in principle possible to find augmenting paths and cycles, in the weighted case, we do not know how to do this efficiently in the CONGEST model. Instead, we use the bipartite MWM approximation algorithm of [2] to find a good matching in each sampled bipartite graph. Unlike when using augmenting paths, this approach can potentially also lead to a worse matching (if the existing matching is already a very good matching of the sampled bipartite graph). We will however see that when computing a sufficiently good approximation in each sampled bipartition, we can use the approach to improve the matching of  $G$  sufficiently often.

Before explaining our algorithms in more detail, we need to introduce the notion of augmenting paths and cycles for weighted matchings. Given a matching  $M$ , a path or cycle in which the edges alternate between edges  $\in M$  and edges  $\notin M$  is called an *alternating path or cycle* w.r.t. matching  $M$ . An alternating path or cycle is called an *augmenting path or cycle* w.r.t.  $M$  if swapping the matching edges with the non-matching edges increases the weight of the matching. This increase is termed the *gain* of an augmenting path/cycle w.r.t.  $M$  (we will omit the qualification 'w.r.t.' if it is clear from the context). By definition, the gain of an augmenting path is positive. Note that alternating cycles (and therefore also augmenting cycles) are always of even length.

Let  $M^*$  be a maximum weighted matching in  $G$ . Consider the symmetric difference  $F = M^* \triangle M$  of  $M^*$  and some arbitrary matching  $M$ . The set  $F$  consists of (vertex-disjoint) alternating paths and cycles where the edges alternate between  $M$  and  $M^*$ . All of those alternating paths and cycles are either augmenting paths/cycles, or their  $M$ -edges have exactly the same weight as their  $M^*$ -edges. The total gain of all the (vertex-disjoint) augmenting paths and cycles induced by  $F$  is therefore exactly  $w(M^*) - w(M)$ .

Dealing with augmenting paths and cycles in the context of edge-weighted graphs is much more challenging than in the unweighted case. Every augmenting path in unweighted graphs improves the matching by exactly one and augmenting cycles do not exist. Further, the classic results of Hopcraft and Karp [39] imply that if the shortest augmenting path is of length  $\ell$ , augmenting over an augmenting path of length  $\ell$  cannot create new augmenting paths of length  $\leq \ell$  and after augmenting over a maximal set of vertex-disjoint augmenting paths of length  $\ell$ , one gets a matching with a shortest augmenting path length  $\geq \ell + 2$ . If in some matching  $M$ , the shortest augmenting path has length  $\ell = 2k - 1 \geq 1$ , we further know that  $M$  already is a  $(1 - \frac{1}{k})$ -approximation of the optimal matching. The  $(1 - \varepsilon)$ -approximation algorithm for unweighted matching in [49] heavily relies on all those properties.

Some of the properties of augmenting paths for unweighted matchings also carry over to the weighted case. In the full version [22] we show that there exists a set of vertex-disjoint augmenting paths and cycles of length at most  $\ell$  for some  $\ell = O(1/\varepsilon)$  such that augmenting over all those paths/cycles improves the current matching by at least  $\frac{\varepsilon}{4} \cdot w(M^*)$ . Basically, the existence of this collection of short augmenting paths can be proven by breaking long augmenting paths and cycles in the symmetric difference  $F = M \triangle M^*$  into short augmenting paths. However, while in the unweighted case, a large set of vertex-disjoint short augmenting paths can be computed efficiently in the CONGEST model (see [7, 49]), it is not clear how to efficiently compute such a set of augmenting paths/cycles for weighted matchings in the CONGEST model, even in bipartite graphs (there are efficient algorithms in the LOCAL model and this has also been exploited in the literature, e.g., in [31, 49, 52]). Fortunately, there still is an efficient CONGEST algorithm for computing a  $(1 - \varepsilon)$ -approximate weighted matching in bipartite graphs [2]. Unlike the existing CONGEST algorithms that are based on the Hopcroft/Karp framework, the algorithm of [2] is even deterministic. It is however not based on augmenting along short augmenting paths or cycles. Instead, it is based on linear programming and on a deterministic rounding scheme that was introduced in [25]. As a result, the matching computed by the algorithm of [2] does not have the nice structural properties of the matchings computed by algorithms based on the Hopcroft/Karp framework (such as not having any short augmenting paths or cycles).

**Our randomized weighted matching algorithm.** The general idea of our approach is as simple as the algorithm for the unweighted case in [49].<sup>1</sup> We first describe the randomized version of our algorithm. We start with an initial matching  $M_0$  of the given weighted graph  $G = (V, E, w)$  and it then consists of iterations  $i = 1, 2, \dots$ . In each iteration, we update the given matching such that at the end of iteration  $i$ , we have matching  $M_i$ . In each iteration  $i$ , we sample a bipartite subgraph  $H_i = (\hat{V}_i, \hat{E}_i)$  of  $G$  as follows. Every node  $v \in V$  colors itself black or white independently with probability  $1/2$ . An edge is called *monochromatic* if both of its endpoints have the same color, otherwise, we call the edge *bichromatic*. To preserve good intermediate results, we keep all the matching edges of the previous matching not occurring in the bipartition, i.e., we keep monochromatic matching edges. We call a node *free* regarding to matching  $M$  if none of its incident edges are  $\in M$ .

<sup>1</sup> Our algorithm is essentially the same one as the one in [7, 49]. We just replace the bipartite matching algorithm used as a subroutine. The analysis is then however different from the analysis in [7, 49].

■ **Algorithm 2** Construct bipartite subgraph  $H_i = (\hat{V}_i, \hat{E}_i)$  of  $G$  based on matching  $M_{i-1}$ .

1. Color each node black or white.
2.  $\hat{V}_i := \{u \in V \mid u \text{ is free or } \exists \{u, v\} \in M_{i-1} \text{ s.t. } \{u, v\} \text{ is bichromatic}\}$ .
3.  $\hat{E}_i := \{\{u, v\} \in E \mid u, v \in \hat{V}_i \text{ and } \{u, v\} \text{ is bichromatic}\}$ .

After sampling the bipartite subgraph  $H_i$ , we use the algorithm of [2] to compute a  $(1 - \lambda)$ -approximate weighted matching of  $H_i$  for a sufficiently small parameter  $\lambda > 0$  ( $\lambda$  will be exponentially small in  $1/\varepsilon$ ). We then update the existing matching  $M_{i-1}$  by replacing the bichromatic matching edges with the matching edges of the newly computed matching of  $H_i$ . Because there is a collection of short augmenting paths and cycles with total gain  $\Theta(w(M^*) - w(M_{i-1}))$  (where  $M^*$  is an optimal matching of  $G$ ), we have a reasonable chance of sampling such paths and cycles so that the matching on  $H_i$  can potentially be improved by a sufficiently large amount. Note however that our algorithm does not guarantee that the quality of the matching improves monotonically during the algorithm. However, because the algorithm of [2] has deterministic guarantees if we choose  $\lambda$  sufficiently small, we have the guarantee that  $w(M_i)$  cannot be much worse than  $w(M_{i-1})$ . With the right choice of parameters, it turns out that  $2^{O(1/\varepsilon)}$  iterations are sufficient to obtain a  $(1 - \varepsilon)$ -approximate matching with at least constant probability.

**Our deterministic weighted matching algorithm.** The basic idea of our deterministic algorithm is the same as for the randomized algorithm. However, we now have to compute the bipartition into black and white nodes in each iteration deterministically. For this purpose, we prove in Lemma 13 that for some  $T = 2^{O(1/\varepsilon)} \cdot \ln n$ , there exist a collection of bipartitions  $H_1, \dots, H_T$  such that every path/cycle of length at most  $O(1/\varepsilon)$  (and thus also every augmenting path/cycle of this length) of  $G$  appears in at least one of these bipartitions. Of course, after changing the matching, also the set of augmenting paths and cycles changes and one can therefore not just iterate over all bipartitions  $H_1, \dots, H_T$ , improve the matching for each bipartition by using a generic MWM approximation algorithm, and guarantee that at the end the resulting matching is a sufficiently good approximation. However, the property of  $H_1, \dots, H_T$  guarantees that for a fixed initial matching  $M$ , when going over all  $T$  bipartitions, there exists one bipartition that can improve the weight of  $M$  by  $\Theta(w(M)/T)$ . We therefore proceed as follows. We iterate  $O(T)$  times through the sequence  $H_1, \dots, H_T$  of bipartitions. For each bipartition, we use the (deterministic) algorithm of [2] to compute a  $(1 - \lambda)$ -approximate weighted matching of the current bipartite graph. We then however only switch to the new matching if it improves the old one by a sufficiently large amount. Checking if a given bipartition leads to a sufficiently large improvement of the current matching can be done efficiently by first applying the diameter reduction technique given by Theorem 7.

## 6 Vertex Cover Lower Bound

Göös and Suomela in [32] showed that there exists a constant  $\varepsilon_0 > 0$  such that computing a  $(1 + \varepsilon_0)$ -approximation of minimum (unweighted) vertex cover in bipartite graphs of maximum degree 3 requires  $\Omega(\log n)$  rounds even in the LOCAL model. We next describe how to extend this lower bound to show that for  $\varepsilon \leq \varepsilon_0$ , computing a  $(1 + \varepsilon)$ -approximate vertex cover for any  $\varepsilon > 0$ . That is, we next prove Theorem 5, i.e., we prove that even in bipartite graph of maximum degree 3, there exists a constant  $\varepsilon_0 > 0$  such that for  $\varepsilon \in (0, \varepsilon_0]$ , computing a  $(1 + \varepsilon)$ -approximate vertex cover requires  $\Omega(\frac{\log n}{\varepsilon})$  rounds in the LOCAL model.

**Proof of Theorem 5.** in [32], Gös and Suomela showed that there exists a bipartite graph  $G = (V_G, E_G)$  with maximum degree 3 and a constant  $\varepsilon_0 > 0$  such that no randomized distributed algorithm with running time  $o(\log n)$  can find a  $(1 + \varepsilon_0)$ -approximate vertex cover on  $G$ . To extend this proof to smaller approximation ratios, we proceed as follows. Given a positive integer parameter  $k$ , we construct a new lower bound graph  $H$  as follows. Graph  $H$  is obtained from graph  $G$  by replacing every edge  $e$  of  $G$  by a path  $P_e$  of length  $2k + 1$ .

We first describe a method to transform a given vertex cover  $S_H$  of  $G$  into a vertex cover  $S'_H$  of  $H$  such that  $|S'_H| \leq |S_H|$  and such that  $S'_H$  has the following form. For each edge  $e$  of  $G$ ,  $S'_H$  contains exactly  $k$  of the inner nodes of the path  $P_e$  in  $H$  and it contains at least one of the end nodes of  $P_e$ . The transformation is done independently for each path  $P_e$  as follows. Let  $P_e = (v_0, v_1, \dots, v_{2k+1})$  be the path that replaces edge  $e = \{v_0, v_{2k+1}\}$  in  $G$ . If  $S_H \cap \{v_0, v_{2k+1}\} \neq \emptyset$ , we add the nodes  $S_H \cap \{v_0, v_{2k+1}\}$  also to  $S'_H$ . If  $S_H \cap \{v_0, v_{2k+1}\} = \emptyset$ , we arbitrarily add either  $v_0$  or  $v_{2k+1}$  to  $S'_H$ . Further  $S'_H$  contains exactly  $k$  of the inner nodes  $v_1, \dots, v_{2k}$  of  $P_e$  in such a way that every edge of  $P_e$  is covered by some node in  $S'_H$ . If  $v_0 \in S'_H$ , we add all  $v_{2i}$  for  $i \in \{1, \dots, k\}$  and otherwise we add all  $v_{2i-1}$  for  $i \in \{1, \dots, k\}$ . Clearly  $S'_H$  is a vertex cover of  $H$ . To see that  $|S'_H| \leq |S_H|$ , observe that in order to cover all  $2k + 1$  edges of  $P_e$ , every vertex cover of  $H$  must contain at least  $k + 1$  of the nodes of  $P_e$  and it also must contain at least  $k$  of the inner nodes of  $P_e$ . If  $S_H \cap \{v_0, v_{2k+1}\} \neq \emptyset$ ,  $S'_H$  only differs in terms of the inner nodes of  $P_e$  from  $S_H$  and we know that also  $S_H$  must contain at least  $k$  inner nodes of  $P_e$ . If  $S_H \cap \{v_0, v_{2k+1}\} = \emptyset$ , we add either  $v_0$  or  $v_{2k+1}$  to  $S'_H$ . However in this case,  $S_H$  contains at least  $k + 1$  inner nodes of  $P_e$  and  $S'_H$  only contains  $k$  inner nodes of  $P_e$ . The transformation from  $S_H$  to  $S'_H$  can be done independently for each of the paths  $P_e$  of length  $2k + 1$  and it can therefore be done in  $O(k)$  rounds in the LOCAL model.

Let  $e_G$  be the number of edges of  $G$  and let  $s_G$  be the size of an optimal vertex cover of  $G$ . Because the maximum degree of  $G$  is 3 and we have an optimal vertex cover of  $G$ , we get that  $e_G = c \cdot s_G$  for some constant  $c \leq 3$ . By the observation above, any vertex cover  $S_H$  of  $H$  can be transformed into an equally good vertex cover  $S'_H$  with a nice structure. Note that  $S'_H$  consists of exactly  $k$  inner nodes of each path  $P_e$  for  $e \in E_G$  and it consists of at least one of the end nodes of each such path (i.e., of a vertex cover of  $G$ ). We therefore obtain that there is an optimal vertex cover of  $H$  that consists of an optimal vertex cover of  $G$  and of  $k$  inner nodes of each  $(2k + 1)$ -hop path  $P_e$  replacing an edge  $e$  of  $G$ . The size  $s_H$  of an optimal vertex cover of  $H$  is therefore exactly  $s_H = s_G + k \cdot e_G = (1 + ck) \cdot s_G$ .

Assume now that we have a  $T$ -round algorithm to compute a  $(1 + \varepsilon)$ -approximate vertex cover  $S_H$  on graph  $H$  for some  $\varepsilon \leq \varepsilon_0 / (1 + 3k) \leq \varepsilon_0 / (1 + ck)$ . By the above observation, in  $O(k)$  rounds, we can transform this vertex cover into a vertex cover  $S'_H$ , which contains  $k \cdot e_G = ck \cdot s_G$  inner path nodes and at least one of the end nodes of each  $(2k + 1)$ -hop path replacing an edge of  $G$  in  $H$ . The vertex cover  $S'_H$  of  $H$  therefore induces a vertex cover of  $G$  of size  $|S'_H| - k \cdot e_G = |S'_H| - ck \cdot s_G$ . Because we assumed that  $\varepsilon \leq \varepsilon_0 / (1 + ck)$ , we have  $|S'_H| \leq (1 + \varepsilon) \cdot (1 + ck) \cdot s_G \leq (1 + \varepsilon_0) s_G + ck \cdot s_G$ . The vertex cover  $S'_H$  of  $H$  therefore induces a  $(1 + \varepsilon_0)$ -approximate vertex cover of  $G$ . We next show that this implies an  $O(1 + T/k)$ -round algorithm to compute a  $(1 + \varepsilon_0)$ -approximate vertex cover of  $G$ . Assume that we want to compute a vertex cover of  $G$ . To do this, the nodes of  $G$  can simulate graph  $H$  by adding  $2k$  virtual nodes on each edge of  $G$ . An  $R$ -round algorithm on  $H$  can then be run in  $O(\lceil R/k \rceil) = O(1 + R/k)$  rounds on  $G$ . We can therefore compute the vertex cover  $S'_H$  on the virtual graph  $H$  in  $O(1 + (T + k)/k) = O(1 + T/k)$  rounds on  $G$ . Since  $k = \Theta(1/\varepsilon)$ , the lower bound of [32] implies a lower bound of  $\Omega(\frac{\log n}{\varepsilon})$  on the time  $T$  for computing a  $(1 + \varepsilon)$ -approximate vertex cover on  $H$ . Finally note that since  $G$  is bipartite, then  $H$  is also bipartite and if the maximum degree of  $G$  is 3, then the maximum degree of  $H$  is also 3. ◀



## References

- 1 M. Ahmadi and F. Kuhn. Distributed maximum matching verification in CONGEST. In *Proc. 34th Symp. on Distributed Computing (DISC)*, pages 37:1–37:18, 2020.
- 2 M. Ahmadi, F. Kuhn, and R. Oshman. Distributed approximate maximum matching in the CONGEST model. In *Proc. 32nd Symp. on Distr. Computing (DISC)*, pages 6:1–6:17, 2018.
- 3 S. Assadi, M. Bateni, A. Bernstein, V. Mirrokni, and C. Stein. Coresets meet EDCS: Algorithms for matching and vertex cover on massive graphs. In *Proc. 30th ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 1616–1635, 2019.
- 4 M. Åstrand, P. Floréen, V. Polishchuk, J. Rybicki, J. Suomela, and J. Uitto. A local 2-approximation algorithm for the vertex cover problem. In *Proc. 23rd Symp. on Distributed Computing (DISC)*, pages 191–205, 2009.
- 5 M. Åstrand and J. Suomela. Fast distributed approximation algorithms for vertex cover and set cover in anonymous networks. In *Proc. 22nd ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*, pages 294–302, 2010.
- 6 B. Awerbuch and D. Peleg. Sparse partitions. In *Proc. 31st IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 503–513, 1990.
- 7 R. Bar-Yehuda, K. Censor-Hillel, M. Ghaffari, and G. Schwartzman. Distributed approximation of maximum independent set and maximum matching. In *Proc. 36th ACM Symp. on Principles of Distributed Computing (PODC)*, full version: [arXiv:1708.00276](https://arxiv.org/abs/1708.00276), pages 165–174, 2017.
- 8 R. Bar-Yehuda, K. Censor-Hillel, Y. Maus, S. Pai, and S. V. Pemmaraju. Distributed approximation on power graphs. In *Proc. 39th ACM Symp. on Principles of Distributed Computing (PODC)*, pages 501–510, 2020.
- 9 R. Bar-Yehuda, K. Censor-Hillel, and G. Schwartzman. A distributed  $(2+\varepsilon)$ -approximation for vertex cover in  $O(\log \Delta/\varepsilon \log \log \Delta)$  rounds. In *Proc. 35th ACM Symp. on Principles of Distributed Computing (PODC)*, pages 3–8, 2016.
- 10 R. Ben-Basat, G. Even, Kawarabayashi K, and G. Schwartzman. A deterministic distributed 2-approximation for weighted vertex cover in  $O(\log N \log \Delta / \log^2 \log \Delta)$  rounds. In *Proc. 25th Coll. on Structural Information and Comm. Complexity (SIROCCO)*, pages 226–236, 2018.
- 11 R. Ben-Basat, G. Even, K. Kawarabayashi, and G. Schwartzman. Optimal distributed covering algorithms. In *Proc. 33rd Symp. on Distributed Computing (DISC)*, pages 5:1–5:15, 2019.
- 12 Ran Ben-Basat, Ken ichi Kawarabayashi, and Gregory Schwartzman. Parameterized distributed algorithms. In *Proc. 33rd In. Symp. on Distributed Computing (DISC)*, pages 6:1–6:16, 2019.
- 13 K. Censor-Hillel, S. Khoury, and A. Paz. Quadratic and near-quadratic lower bounds for the CONGEST model. In *Proc. 31st Symp. on Distr. Computing (DISC)*, pages 10:1–10:16, 2017.
- 14 A. Czygrinow and M. Hańkowiak. Distributed algorithm for better approximation of the maximum matching. In *9th Annual Int. Computing and Combinatorics Conf. (COCOON)*, pages 242–251, 2003.
- 15 A. Czygrinow, M. Hańkowiak, and E. Szymanska. A fast distributed algorithm for approximating the maximum matching. In *Proceedings of 12th Annual European Symp. on Algorithms (ESA)*, pages 252–263, 2004.
- 16 R. Diestel. *Graph Theory*, chapter 2.1. Springer, Berlin, 3rd edition, 2005.
- 17 I. Dinur and S. Safra. On the hardness of approximating vertex cover. *Annals of Mathematics*, 162(1):439–485, 2005.
- 18 J. Edmonds. Maximum matching and a polyhedron with 0, 1-vertices. *J. of Res. the Nat. Bureau of Standards*, 69 B:125–130, 1965.
- 19 J. Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17(3):449–467, 1965.
- 20 J. Egerváry. Matrixok kombinatorius tulajdonságairól [On combinatorial properties of matrices]. *Matematikai és Fizikai Lapok (in Hungarian)*, 38(16–28), 1931.
- 21 G. Even, M. Medina, and D. Ron. Distributed maximum matching in bounded degree graphs. In *Proceedings of the 2015 Int. Conf. on Distributed Computing and Networking (ICDCN)*, pages 18:1–18:10, 2015.

- 22 S. Faour, M. Fuchs, and F. Kuhn. Distributed CONGEST approximation of weighted vertex covers and matchings. *CoRR*, abs/2111.10577, 2021. [arXiv:2111.10577](#).
- 23 S. Faour and F. Kuhn. Approximating bipartite minimum vertex cover in the CONGEST model. In *Proc. 24th Conf. on Principles of Distr. Systems (OPODIS)*, pages 29:1–29:16, 2020.
- 24 U. Feige, Y. Mansour, and R.É. Schapire. Learning and inference in the presence of corrupted inputs. In *Proc. 28th Conf. on Learning Theory (COLT)*, pages 637–657, 2015.
- 25 M. Fischer. Improved deterministic distributed matching via rounding. In *Proc. 31st Symp. on Distributed Computing (DISC)*, pages 17:1–17:15, 2017.
- 26 M. Fischer, S. Mitrovic, and J. Uitto. Deterministic  $(1 + \varepsilon)$ -approximate maximum matching with  $\text{poly}(1/\varepsilon)$  passes in the semi-streaming model. *CoRR*, abs/2106.04179, 2021. [arXiv:2106.04179](#).
- 27 K.-T. Foerster, J. H. Korhonen, J. Rybicki, and S. Schmid. Brief announcement: Does preprocessing help under congestion? In *Proc. 38th ACM Symp. on Principles of Distributed Computing (PODC)*, pages 259–261, 2019.
- 28 M. Ghaffari, C. Jin, and D. Nilis. A massively parallel algorithm for minimum weight vertex cover. In *Proc. 32nd ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*, pages 259–268, 2020.
- 29 M. Ghaffari and F. Kuhn. Deterministic distributed vertex coloring: Simpler, faster, and without network decomposition. In *Proc. 62nd IEEE Symp. on Foundations of Computer Science (FOCS)*, 2021.
- 30 M. Ghaffari, F. Kuhn, and Y. Maus. On the complexity of local distributed graph problems. In *Proc. 39th ACM Symp. on Theory of Computing (STOC)*, pages 784–797, 2017.
- 31 M. Ghaffari, F. Kuhn, Y. Maus, and J. Uitto. Deterministic distributed edge-coloring with fewer colors. In *Proc. 50th ACM Symp. on Theory of Comp. (STOC)*, pages 418–430, 2018.
- 32 M. Göös and J. Suomela. No sublogarithmic-time approximation scheme for bipartite vertex cover. *Distributed Computing*, 27(6):435–443, 2014.
- 33 F. Grandoni, J. Könemann, and A. Panconesi. Distributed weighted vertex cover via maximal matchings. *ACM Trans. Algorithms*, 5(1):6:1–6:12, 2008.
- 34 F. Grandoni, J. Könemann, A. Panconesi, and M. Sozio. A primal-dual bicriteria distributed algorithm for capacitated vertex cover. *SIAM J. Comput.*, 38(3):825–840, 2008.
- 35 D. G. Harris. Distributed local approximation algorithms for maximum matching in graphs and hypergraphs. *SIAM J. Computing*, 49(4):711–746, 2020.
- 36 J. Håstad. Some optimal inapproximability results. *J. of the ACM*, 48(4):798–859, 2001.
- 37 D. Hochbaum, editor. *Approximation Algorithms for NP-hard Problems*. PWS Publishing Company, 1997.
- 38 J.H. Hoepman, S. Kutten, and Z. Lotker. Efficient distributed weighted matchings on trees. In *Proc 13th Coll. on Structural Inf. and Comm. Complexity (SIROCCO)*, pages 115–129, 2006.
- 39 J. E. Hopcroft and R. M. Karp. An  $n^{5/2}$  algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.*, 2(4):225–231, 1973.
- 40 A. Israeli and A. Itai. A fast and simple randomized parallel algorithm for maximal matching. *Inf. Process. Lett.*, 22(2):77–80, 1986.
- 41 G. Karakostas. A better approximation ratio for the vertex cover problem. *ACM Trans. Algorithms*, 5(4):41:1–41:8, 2009.
- 42 D. König. Gráfok és mátrixok. *Matematikai és Fizikai Lapok*, 38:116–119, 1931.
- 43 S. Khot and O. Regev. Vertex cover might be hard to approximate to within 2-epsilon. *J. Comput. Syst. Sci.*, 74(3):335–349, 2008.
- 44 S. Khuller, U. Vishkin, and N. E. Young. A primal-dual parallel approximation technique applied to weighted set and vertex covers. *J. Algorithms*, 17(2):280–289, 1994.
- 45 C. Koufogiannakis and N. E. Young. Distributed algorithms for covering, packing and maximum weighted matching. *Distributed Computing*, 24(1):45–63, 2011.

- 46 F. Kuhn, T. Moscibroda, and R. Wattenhofer. What cannot be computed locally! In *Proc. 23rd ACM Symp. on Principles of Distributed Computing (PODC)*, pages 300–309, 2004.
- 47 F. Kuhn, T. Moscibroda, and R. Wattenhofer. The price of being near-sighted. In *Proceedings of 17th Symp. on Discrete Algorithms (SODA)*, pages 980–989, 2006.
- 48 N. Linial and M. Saks. Low diameter graph decompositions. *Combinatorica*, 13(4):441–454, 1993.
- 49 Z. Lotker, B. Patt-Shamir, and S. Pettie. Improved distributed approximate matching. *J. ACM*, 62(5):38:1–38:17, 2015.
- 50 Z. Lotker, B. Patt-Shamir, and A. Rosén. Distributed approximate matching. *SIAM Journal on Computing*, 39(2):445–460, 2009.
- 51 G. L. Miller, R. Peng, and S. C. Xu. Parallel graph decompositions using random shifts. In *Proc. 25th ACM Symp. on Parallelism in Alg. and Arch. (SPAA)*, pages 196–203, 2013.
- 52 T. Nieberg. Local, distributed weighted matching on general and wireless topologies. In *Proc. Joint Workshop on Foundations of Mobile Computing (DIALM-POMC)*, pages 87–92, 2008.
- 53 D. Peleg. *Distributed Computing: A Locality-Sensitive Approach*. SIAM, 2000.
- 54 V. Rozhoň and M. Ghaffari. Polylogarithmic-time deterministic network decomposition and distributed derandomization. In *Proc. 52nd ACM Symp. on Theory of Computing (STOC)*, pages 350–363, 2020.
- 55 S. Schmid and J. Suomela. Exploiting locality in distributed SDN control. In *Proc. 2nd ACM SIGCOMM Works. on Hot Topics in Software Defined Netw. (HotSDN)*, pages 121–126, 2013.
- 56 M. Wattenhofer and R. Wattenhofer. Distributed weighted matching. In *Proc. 18th Conf. on Distributed Computing (DISC)*, pages 335–348, 2004.

## A Basic Tools

### A.1 Fractional Approximation Algorithm

In all our upper bounds, we need an efficient deterministic distributed approximation scheme for the fractional variants of the MWVC and the MWM problem. In [2], Ahmadi, Kuhn, and Oshman showed that for every instance of the MWM problem with weights in the range  $[1, W]$  and for every  $\varepsilon \in (0, 1]$ , it is possible to deterministically compute a  $(1 - \varepsilon)$ -approximate fractional solution in time  $O(\log(\Delta W)/\varepsilon^2)$  in the CONGEST model. In our algorithm to solve the MWVC problem, we need a variant of this algorithm, which works for the (unweighted) fractional  $w$ -matching problem. The following theorem shows that this can be done with the same asymptotic cost that we have for the fractional MWM problem.

► **Theorem 12.** *Let  $G = (V, E, w)$  be an undirected  $n$ -node graph with integer node weights  $w(v) \in \{1, \dots, W\}$ . Then, for every  $\varepsilon \in (0, 1]$ , there is a deterministic  $O(\log(\Delta W)/\varepsilon^2)$ -round CONGEST algorithm to compute a  $(1 - \varepsilon)$ -approximate solution to the fractional  $w$ -matching problem in  $G$  and a  $(1 + \varepsilon)$ -approximate solution to the minimum fractional weighted vertex cover problem.*

**Proof.** The theorem could be proven for arbitrary weights in the range  $[1, W]$  by adapting the algorithm of [2]. We here give a generic reduction, which works for integer weights and which allows to use the result of [2] (almost) in a blackbox manner.

We define an unweighted graph  $G' = (V', E')$  as follows. For every node  $v \in V$ ,  $V'$  contains  $w(v)$  nodes  $(v, 1), \dots, (v, w(v))$ . Further, for every edge  $\{u, v\} \in E$ , we add a complete bipartite graph between the corresponding nodes to  $G'$ , that is,  $E'$  contains all edges  $\{(u, i), (v, j)\}$  for  $i \in \{1, \dots, w(v)\}$  and  $j \in \{1, \dots, w(u)\}$ . We then apply the unweighted fractional matching algorithm of [2] to compute a  $(1 - \varepsilon)$ -approximate fractional matching of  $G'$ . The maximum degree of any node in  $G'$  is at most  $\Delta \cdot W$  and when running the algorithm in the CONGEST model on  $G'$ , the round complexity of the algorithm is therefore  $O(\log(\Delta W)/\varepsilon^2)$  as claimed (cf. Theorem 2 in [2]).

For an edge  $e \in E'$  of  $G'$ , assume that  $z_e$  is the fractional matching value of  $e$  in the computed fractional matching of  $G'$ . We can transform the fractional matching of  $G'$  into a fractional  $w$ -matching of  $G$  as follows. For each edge  $\{u, v\} \in E$  of  $G$ , we define  $y_{\{u, v\}} := \sum_{i=1}^{w(u)} \sum_{j=1}^{w(v)} z_{\{(u, i), (v, j)\}}$ . Note that we obtain a valid fractional  $w$ -matching because for every node  $u \in V$  of  $G$ , the sum of the fractional values of its edges is at most equal to number of copies of  $u$  in  $G'$ , which is equal to  $w(u)$ . In the other direction, given a fractional  $w$ -matching  $y_e$  of  $G$ , we can compute a fractional matching  $z_{e'}$  of  $G'$  of the same size in the following way. For each edge  $\{u, v\} \in E$  of  $G$ , we assign  $z_{\{(u, i), (v, j)\}} := y_{\{u, v\}} / (w(u) \cdot w(v))$ . The size of a maximum fractional  $w$ -matching on  $G$  is therefore equal to the size of a maximum fractional matching on  $G'$  and given a  $(1 - \varepsilon)$ -approximation of maximum fractional matching on  $G'$ , we therefore also obtain a  $(1 - \varepsilon)$ -approximation of maximum fractional  $w$ -matching on  $G$ .

It remains to show that we can efficiently run the fractional matching algorithm of [2] in the CONGEST model on  $G$ . Since every node of  $G$  has potentially a large number of copies in  $G'$ , it is not true that any CONGEST algorithm on  $G'$  can be run efficiently in the CONGEST model on  $G$ . However, the behavior of the algorithm of [2] is independent of the node IDs and since the algorithm is deterministic, all copies of a node  $u \in V$  are symmetric and therefore behave in exactly the same way. Each node of  $G$  can therefore simulate all its copies in  $G'$  at no additional cost.

We note that the same reduction has also been used in [33], where a maximal matching of  $G'$  is used to compute a 2-approximate weighted vertex cover of  $G$ . ◀

## A.2 Bipartitions to Cover all Paths

The following lemma is used as a tool for our deterministic maximum weighted matching algorithms. It shows that there exists a sequence of bipartitions that is “good” in every graph and for every collection of short augmenting paths and cycles.

► **Lemma 13.** *Let  $N > 0$  and  $k \leq N$  be two integers. There exists a collection of  $T = O(k \cdot 2^k \cdot \log N)$  functions  $f_1, \dots, f_T \in [N] \rightarrow \{0, 1\}$  such that for every vector  $(x_1, \dots, x_k) \in [N]^k$  with pairwise disjoint entries, there exists a function  $f_i$  in the collection such that  $f_i(x_j) = j \bmod 2$  for every  $1 \leq j \leq k$ .*

**Proof.** We will prove the lemma with a probabilistic argument. The probability that a randomly chosen  $f_i$  maps some  $x_j$  to 0 respectively 1 is  $1/2$ . We say an function  $f_i$  takes care of  $(x_1, \dots, x_k)$  if  $(f_i(x_1), \dots, f_i(x_k)) \in (0, 1, 0, 1, \dots)$ . The probability that  $f_i$  takes care for a specific vector is  $2^{-k}$ . Further, the probability that no function in a collection of  $T$  random function takes care of a fixed vector is  $(1 - 2^{-k})^T \leq e^{-T/2^k}$ . Since there are  $N^k$  different vectors, the probability that there exists a vector such that no function  $f_i$  takes care of it, is at most  $N^k \cdot e^{-T/2^k}$ . Choosing  $T > k \cdot 2^k \cdot \ln N$  pushes this probability below 1, which shows that there must exist a collection of  $T$  functions  $f_1, \dots, f_T$  s.t. for every possible vector at least one of those functions will take care. ◀

# Improved Distributed Fractional Coloring Algorithms

Alkida Balliu ✉

Gran Sasso Science Institute, L'Aquila, Italy

Fabian Kuhn ✉

University of Freiburg, Germany

Dennis Olivetti ✉

Gran Sasso Science Institute, L'Aquila, Italy

---

## Abstract

We prove new bounds on the distributed fractional coloring problem in the LOCAL model. A fractional  $c$ -coloring of a graph  $G = (V, E)$  is a fractional covering of the nodes of  $G$  with independent sets such that each independent set  $I$  of  $G$  is assigned a fractional value  $\lambda_I \in [0, 1]$ . The total value of all independent sets of  $G$  is at most  $c$ , and for each node  $v \in V$ , the total value of all independent sets containing  $v$  is at least 1. Equivalently, fractional  $c$ -colorings can also be understood as multicolorings as follows. For some natural numbers  $p$  and  $q$  such that  $p/q \leq c$ , each node  $v$  is assigned a set of at least  $q$  colors from  $\{1, \dots, p\}$  such that adjacent nodes are assigned disjoint sets of colors. The minimum  $c$  for which a fractional  $c$ -coloring of a graph  $G$  exists is called the fractional chromatic number  $\chi_f(G)$  of  $G$ .

Recently, [Bousquet, Esperet, and Pirot; SIROCCO '21] showed that for any constant  $\varepsilon > 0$ , a fractional  $(\Delta + \varepsilon)$ -coloring can be computed in  $\Delta^{O(\Delta)} + O(\Delta \cdot \log^* n)$  rounds. We show that such a coloring can be computed in only  $O(\log^2 \Delta)$  rounds, without any dependency on  $n$ .

We further show that in  $O(\frac{\log n}{\varepsilon})$  rounds, it is possible to compute a fractional  $(1 + \varepsilon)\chi_f(G)$ -coloring, even if the fractional chromatic number  $\chi_f(G)$  is not known. That is, the fractional coloring problem can be approximated arbitrarily well by an efficient algorithm in the LOCAL model. For the standard coloring problem, it is only known that an  $O(\frac{\log n}{\log \log n})$ -approximation can be computed in polylogarithmic time in the LOCAL model. We also show that our distributed fractional coloring approximation algorithm is best possible. We show that in trees, which have fractional chromatic number 2, computing a fractional  $(2 + \varepsilon)$ -coloring requires at least  $\Omega(\frac{\log n}{\varepsilon})$  rounds.

We finally study fractional colorings of regular grids. In [Bousquet, Esperet, and Pirot; SIROCCO '21], it is shown that in regular grids of bounded dimension, a fractional  $(2 + \varepsilon)$ -coloring can be computed in time  $O(\log^* n)$ . We show that such a coloring can even be computed in  $O(1)$  rounds in the LOCAL model.

**2012 ACM Subject Classification** Theory of computation → Distributed algorithms

**Keywords and phrases** distributed graph algorithms, distributed coloring, locality, fractional coloring

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2021.18

## 1 Introduction & Related Work

The distributed graph coloring problem is at the heart of the area of distributed graph algorithms and it is one of the prototypical problems to study distributed symmetry breaking. Already in [31], Linial showed that deterministically coloring a ring network with  $O(1)$  colors and thus more generally coloring  $n$ -node graphs of maximum degree  $\Delta$  with a number of colors that only depends on  $\Delta$  requires  $\Omega(\log^* n)$  rounds. In [39], Naor extended this lower bound to randomized algorithms. Subsequently, over the last three decades, the distributed coloring problem has been studied intensively and we now have a quite good understanding of the complexity of the problem. Mostly, researchers focused on the problem of computing a coloring with  $\Delta + 1$  colors, i.e., with the number of colors that can be



© Alkida Balliu, Fabian Kuhn, and Dennis Olivetti;  
licensed under Creative Commons License CC-BY 4.0

25th International Conference on Principles of Distributed Systems (OPODIS 2021).

Editors: Quentin Bramas, Vincent Gramoli, and Alessia Milani; Article No. 18; pp. 18:1–18:23



Leibniz International Proceedings in Informatics

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

obtained by a simple sequential greedy algorithm. In light of the  $\Omega(\log^* n)$  lower bounds of [31, 39], there is a long line of research to (deterministically) solve  $(\Delta + 1)$ -coloring in time  $f(\Delta) + O(\log^* n)$  for some function  $f$  (see, e.g., [5, 8, 9, 17, 21, 29, 36, 45]), where the current best bound of  $O(\sqrt{\Delta \log \Delta} + \log^* n)$  was proven in [8, 17, 36]. The complexity of distributed  $(\Delta + 1)$ -coloring has also been studied as a function of  $n$ . The randomized complexity has been known to be  $O(\log n)$  since the 1980s [1, 25, 31, 35] and it has recently been improved to  $O(\log^3 \log n)$  [10, 13, 20, 23] and even to  $O(\log^* n)$  for graphs of maximum degree  $\Delta \geq \log^{2+\varepsilon} n$  [22]. For a long time, the best deterministic  $(\Delta + 1)$ -coloring algorithms had a complexity of  $2^{O(\sqrt{\log n})}$  [3, 28, 40] and it was only recently shown in a breakthrough paper by Rozhoň and Ghaffari [42] that the distributed  $(\Delta + 1)$ -coloring problem (and many other distributed graph problems) can be solved in time  $\text{poly } \log n$  deterministically. Subsequently, the deterministic complexity of the distributed  $(\Delta + 1)$ -coloring problem has even been improved to  $O(\log^2 \Delta \cdot \log n)$  [20].

While much of the existing work on distributed vertex coloring is on the  $(\Delta + 1)$ -coloring problem, it is of course also relevant to understand the complexity of more restrictive or more relaxed variants of the problem, for example by considering vertex colorings with more or fewer colors. Already in [31], Linial showed that an  $O(\Delta^2)$ -coloring can be computed deterministically in time only  $O(\log^* n)$ . Over the years, there were several papers that considered distributed coloring algorithms to color graphs with at least  $\Delta + 1$  colors (e.g., [6, 9, 15, 26, 28, 37, 44]). One however needs to use  $\omega(\Delta)$  colors to obtain significantly faster distributed coloring algorithms. Colorings with less than  $\Delta + 1$  colors however require significantly more time. In [31], Linial shows that even on  $\Delta$ -regular trees, computing an  $O(\sqrt{\Delta})$ -coloring requires  $\Omega(\log_{\Delta} n)$  rounds deterministically.<sup>1</sup> This was improved in [12], where it is shown that even computing a  $\Delta$ -coloring of  $\Delta$ -regular trees requires  $\Omega(\log_{\Delta} n)$  rounds deterministically and  $\Omega(\log_{\Delta} \log n)$  rounds with randomization. There are algorithms that nearly match those bounds [19, 20, 41]. Further, by using network decompositions [3, 14, 33, 40, 42], it is possible to efficiently approximate the best possible vertex coloring [7].<sup>2</sup> In particular, in  $\text{poly } \log n$  rounds, it is possible to compute a coloring of a graph  $G$  with  $O\left(\frac{\log n}{\log \log n}\right) \cdot \chi(G)$  colors.

Another natural relaxation of the vertex coloring problem is the fractional coloring problem. A  $c$ -coloring of the nodes  $V$  of a graph  $G = (V, E)$  can be seen as a partition of  $V$  into  $c$  independent sets. A fractional  $c$ -coloring is an assignment of positive weights  $\lambda_I$  to the independent sets  $I$  of  $G$  such that for every node  $v \in V$ , the total weight of the independent sets that contain  $v$  is equal to (at least) 1 and such that the total weight of all independent sets is equal to  $c$ . The smallest  $c$  for which a graph  $G$  has a fractional  $c$ -coloring is called the fractional chromatic number  $\chi_f(G)$  of  $G$ . Alternatively, a fractional coloring can be defined as a multicoloring as follows. For two integer parameters  $p$  and  $q$  ( $p \geq q$ ), a  $(p : q)$ -coloring is an assignment of (at least)  $q$  colors to each node such that adjacent nodes are assigned disjoint sets of colors and such that the total number of distinct colors is equal to  $p$ . A  $(p : q)$ -coloring directly gives a fractional  $(p/q)$ -coloring and for any graph  $G$  with fractional chromatic number  $\chi_f(G)$ , there exists a pair of integers  $p$  and  $q$  for which a  $(p : q)$ -coloring of  $G$  with  $p/q = \chi_f(G)$  exists.

<sup>1</sup> Linial uses an explicit construction of regular high-girth graphs with large chromatic number for his lower bound, but he remarks that by using the right probabilistic construction, the lower bound on the number of colors can be improved to  $\Omega(\Delta/\log \Delta)$ .

<sup>2</sup> This approach exploits the standard distributed communication models, which in particular allow unbounded internal computations at all nodes.

From the distributed computing perspective, we believe that fractional colorings are interesting and relevant for two reasons. First, in some cases, where graph colorings are needed in practice, one can also use a fractional coloring. For example, if  $G$  describes the possible conflicts between the wireless transmissions of nodes in a radio network, a  $c$  coloring of the nodes of  $G$  can be used to obtain a TDMA schedule for the nodes of  $G$ . The length of such a schedule is equal to the number of colors  $c$  and every node can therefore be active in a  $\frac{1}{c}$ -fraction of all time slots. A  $(p : q)$ -coloring of the nodes of  $G$  can also directly be used to obtain a TDMA schedule of length  $p$  and in which every node is active in  $q$  of the time slots (i.e., in all the time slots corresponding to its colors). Each node is therefore active in a  $q/p$ -fraction of all time slots. By computing a fractional coloring instead of a standard coloring, we can therefore potentially increase the fraction of active slots for each node and thus also the usage of the communication channel. Further, understanding the complexity of distributed fractional coloring will generally improve our understanding of the complexity of distributed coloring: what parts of the difficulty in computing vertex colorings stem from the fact that we need to assign exactly one color to every node (and that we thus need to break symmetries) and what parts of the difficulties remain if we compute fractional colorings, where we can “average” over a possibly larger total number of colors.

We are aware of three previous publications that studied the distributed fractional coloring problem. In [27], it is shown that in any graph  $G$ , a fractional  $(\text{degree} + 1)$ -coloring with support  $N!$  can be computed in a single deterministic communication round (where  $N$  is the number of IDs). That is, there is a 1-round algorithm that computes a multicoloring with  $N!$  colors such that every node  $v$  gets assigned a set of at least  $N! / (\text{deg}_G(v) + 1)$  colors. That is, when considering fractional colorings, the standard  $(\Delta + 1)$ -coloring problem can be solved in a single time step. It is further shown that a fractional  $(1 + \varepsilon)(\text{degree} + 1)$ -coloring with support  $O(\Delta^2 \log N / \varepsilon^2)$  can also be computed in 1 round. In both bounds,  $N$  can be replaced by  $C$  if an initial proper  $C$ -coloring of  $G$  is given. Similar results were also shown in [24]. Very recently, Bousquet, Esperet, and Pirot [11] made some interesting further progress on the distributed fractional coloring problem.

In [11], it is shown that although a fractional  $(\Delta + 1)$ -coloring can be computed in a single communication round, in  $\Delta$ -regular graphs that do not contain  $K_{\Delta+1}$  as a subgraph, the fractional  $\Delta$ -coloring problem requires time  $\Omega(\log_{\Delta} n)$  deterministically and  $\Omega(\log_{\Delta} \log n)$  rounds with randomization. That is, for the fractional  $\Delta$ -coloring problem, the same lower bounds as for the standard  $\Delta$ -coloring problem hold.<sup>3</sup> It is also shown that in graphs that do not contain  $(\Delta + 1)$ -cliques, a  $(q\Delta + 1 : q)$ -coloring can be computed in time  $O(q^3 \Delta^{2q} + q \log^* n)$  deterministically. By setting  $q = 1/\varepsilon$ , this implies that for any  $\varepsilon > 0$ , a fractional  $(\Delta + \varepsilon)$ -coloring can be computed in time  $O(\Delta^{O(1/\varepsilon)} + \frac{1}{\varepsilon} \cdot \log^* n)$ . In addition, the paper shows that, for any constant  $\varepsilon > 0$  and constant integer  $d > 0$ , in regular  $d$ -dimensional grids, it is possible to compute a fractional  $(2 + \varepsilon)$ -coloring in  $O(\log^* n)$  rounds. Hence, while in bounded degree graphs, deterministically computing a fractional  $\Delta$ -coloring requires  $\Omega(\log n)$  rounds, one can get arbitrarily close in only  $O(\log^* n)$  rounds. Moreover, in graphs from a minor-closed family of graphs and with sufficiently large girth, it is possible to compute a fractional  $(2 + \varepsilon)$ -coloring in  $O(\frac{\log n}{\varepsilon})$  rounds, for any constant  $\varepsilon > 0$ . Those results in particular imply that in some graphs, fractional colorings that are arbitrarily close to the best such colorings can be computed.

<sup>3</sup> It has to be noted that for the standard  $\Delta$ -coloring problem, the lower bounds hold in  $\Delta$ -regular trees, but for the fractional  $\Delta$ -coloring problem, the known lower bounds (presented in [11]) only hold for graphs in which every node is contained in some  $K_{\Delta}$ .

In this paper, we improve on the results of [11] in several ways. We here give an overview over our results, for a detailed statement of the results, we refer to Section 3. First, we improve the time to compute a fractional  $(\Delta + \varepsilon)$ -coloring. We show that a coloring of the same quality as in [11] (i.e., a  $(q\Delta + 1 : q)$ -coloring for  $q = 1/\varepsilon$ ) can be computed in time  $O(\frac{1}{\varepsilon^2} \cdot \sqrt{\Delta} \cdot \text{poly log } \Delta + \frac{1}{\varepsilon} \cdot \log^* n)$  and a slightly worse  $(q\Delta : q - 1)$ -coloring for  $q = \Theta(\Delta/\varepsilon)$  can be computed deterministically in  $O(\log^2(\frac{1}{\varepsilon}) \cdot \sqrt{\Delta} \text{poly log } \Delta + (1 + \log_{\Delta} \frac{1}{\varepsilon}) \cdot \log^* n)$  communication rounds. Moreover, if we further increase the total number of colors, a fractional  $(\Delta + \varepsilon)$ -coloring can even be computed deterministically in time  $O(\log^2 \Delta + \log^2(\frac{1}{\varepsilon}) + \log^3(\frac{1}{\varepsilon})/\log \Delta)$ , i.e., we can improve the time dependency on  $\Delta$  to polylogarithmic and we can drop the dependency on the number of nodes  $n$  altogether. We further show that the dependency on  $n$  can also be removed in the algorithm for fractionally coloring grids. In  $d$ -dimensional grids, for constant  $\varepsilon > 0$  and constant  $d$ , a  $(2 + \varepsilon)$  can be computed in  $O(1)$  time. In addition, we study the problem of computing fractional colorings that are arbitrarily close to the best possible fractional colorings. For any  $\varepsilon > 0$ , we show that it is always possible to deterministically compute a fractional  $(1 + \varepsilon) \cdot \chi_f(G)$ -coloring of a graph  $G$  in time  $O(\frac{\log n}{\varepsilon})$  and we show that computing such a fractional coloring on trees requires  $\Omega(\frac{\log n}{\varepsilon})$  rounds even with randomization. Note that this is in contrast to the standard coloring problem for which we are not aware of a  $\text{poly log } n$ -time algorithm that computes an approximation to the minimum vertex coloring problem with an approximation ratio that is better than  $O(\frac{\log n}{\log \log n})$ .

The remainder of this paper is organized as follows. In Section 2 we describe the LOCAL model of distributed computing and we give some useful definitions. Section 3 contains the detailed statements of our contributions. In Section 4 we present some generic results that are then used in our algorithms. Section 5 contains deterministic algorithms for computing a fractional  $(\Delta + \varepsilon)$ -coloring. In Appendix A we show randomized and deterministic algorithms for computing arbitrarily good approximations of the chromatic number of a graph. Then, in Section 6 we present a lower bound of  $\Omega(\log n/\varepsilon)$  rounds for computing a fractional  $(2 + \varepsilon)$ -coloring. Finally, Appendix B contains our constant-time algorithm for fractional  $(2 + \varepsilon)$ -coloring on  $d$ -dimensional grids.

## 2 Model and Definitions

### 2.1 LOCAL model

The model of computation that we consider is the well-known LOCAL model of distributed computing. A distributed network is modeled as a graph where nodes are the computing entities, and edges represent communication links. Each node is equipped with a unique identifier (ID) from  $\{1, \dots, n^c\}$  where  $n$  is the total number of nodes in the graph and  $c \geq 1$  is a constant. Initially, each node knows its own ID and degree, the maximum degree  $\Delta$  of the graph, and the total number  $n$  of the nodes. The computation proceeds in synchronous rounds, where at each round each node sends messages to its neighbors, receives messages from its neighbors, and performs some local computation. In the LOCAL model the size of the messages and the local computation is not bounded. We say that an algorithm correctly solves a task in this model (e.g., a vertex coloring) in time  $T$  if each node provides a local output (e.g., a color) within  $T$  communication rounds, and the local outputs together yield a correct global solution (e.g., a proper coloring). In the randomized version of the LOCAL model, additionally, each node is equipped with a random bit string. In this paper we will consider both Monte Carlo and Las Vegas randomized algorithms. A  $T$ -rounds Monte Carlo algorithm must always terminate within  $T$  rounds, and the global output it produces must



be correct with high probability, that is, with probability at least  $1 - 1/n^c$ , for an arbitrary constant  $c \geq 1$ . A  $T$ -rounds Las Vegas algorithm must terminate within  $T$  rounds with high probability, and it must always produce a correct solution. Notice that, since the size of the messages is not bounded, we can see a  $T$ -round deterministic or a randomized Monte Carlo algorithm that runs at some node  $u$  as a mapping of the  $T$ -hop neighborhood of  $u$  into an output. This does not hold for Las Vegas algorithms, since there the running time is only bounded with high probability.

## 2.2 Definitions

We start by defining the notion of  $(p : q)$ -coloring. Informally, this coloring is an assignment of colors to the nodes of a graph, such that the colors come from a palette of  $p$  colors, and such that to each node are assigned at least  $q$  different colors. Neighboring nodes must have disjoint sets of colors.

► **Definition 1** ( $(p : q)$ -coloring). *Let  $p \geq q \geq 1$ . A  $(p : q)$ -coloring of a graph  $G = (V, E)$  is an assignment of a set  $X_v \subset [p]$  to each node  $v \in V$  such that for all  $v \in V$ ,  $|X_v| \geq q$ , and for all edges  $\{u, v\} \in E$ ,  $X_u \cap X_v = \emptyset$ .*

Sometimes we are not interested in the total number of colors, but just in the ratio between the total number of colors and the number of colors assigned to the nodes. This notion is captured by the definition of fractional coloring.

► **Definition 2** (Fractional  $c$ -coloring). *A fractional  $c$ -coloring is a  $(p : q)$ -coloring satisfying  $p/q \leq c$ .*

Given a  $(p : q)$ -coloring, we call  $p$  the *support* of the coloring. Naturally, apart from minimizing the ratio  $p/q$  and the time for computing a fractional coloring, we also want to minimize the support  $p$ .

The minimum value  $c$  for which there exists a fractional  $c$ -coloring is called fractional chromatic number.

► **Definition 3** (Fractional chromatic number). *The fractional chromatic number  $\chi_f(G)$  of a graph  $G$  is defined as*

$$\chi_f(G) := \inf \left\{ \frac{p}{q} : G \text{ has a } (p : q)\text{-coloring} \right\} = \min \left\{ \frac{p}{q} : G \text{ has a } (p : q)\text{-coloring} \right\}.$$

► **Definition 4** (Partial coloring). *A partial  $c$ -coloring is a coloring of the vertices of a graph such that each node is either colored from a color in  $\{1, \dots, c\}$ , or is uncolored. Similarly, a partial  $(p : q)$ -coloring is a coloring of the vertices of a graph such that each node, either has at least  $q$  colors in  $\{1, \dots, p\}$ , or is uncolored.*

We now provide some additional definitions that will be useful when describing our algorithms. Given a graph  $G$ , we denote with  $\deg_G(v)$  the degree of node  $v$  in  $G$ .

► **Definition 5** (List coloring). *In the  $c_v$ -list (vertex) coloring problem, each node  $v$  is equipped with a list of arbitrary  $c_v$  colors, and the goal is to assign to each node a color from its list, such that the resulting outcome is a proper coloring of the graph  $G$ . In particular, in the (degree +  $x$ )-list coloring problem, each node  $v$  has a list of size at least  $\deg_G(v) + x$ .*

► **Definition 6** (Degree-choosability). *A graph  $G = (V, E)$  is degree-choosable if it admits a  $c_v$ -list coloring for any list assignment satisfying  $|c_v| \geq \deg_G(v), \forall v \in V$ .*

► **Definition 7** (Network decomposition). A  $(c, d)$ -network decomposition of the graph  $G$  is a partition of the vertices of  $G$  into at most  $c$  disjoint color classes such that each connected subgraph induced by nodes of color  $i$  has strong diameter at most  $d$ .

► **Definition 8** (Distance). Let  $G = (V, E)$  be a graph. For a pair of nodes  $u, v \in V$ , we denote with  $\text{dist}(u, v)$  the hop-distance between  $u$  and  $v$  in  $G$ . We also denote the distance between a node  $v \in V$  and a set of nodes  $S \subseteq V$  as  $\text{dist}(v, S) = \min\{\text{dist}(v, u) \mid u \in S\}$ .

► **Definition 9** (Ruling set). An  $(\alpha, \beta)$ -ruling set is a set of nodes satisfying that the nodes in the set are at distance at least  $\alpha$  between each other, and nodes not in the set are at distance at most  $\beta$  from nodes in the set.

### 3 Our Results

Our first main contribution, presented in Section 5, is in the improvement of the main algorithm of [11]. In particular, we start by showing that a fractional  $(\Delta + \varepsilon)$ -coloring, with small support, can be obtained in a time that depends only polynomially in  $\Delta$  and  $\varepsilon$ . In [11], this dependency is exponential. In the following, we assume  $\Delta \geq 3$ <sup>4</sup>.

► **Theorem 10.** A  $(q\Delta : q-1)$ -coloring, for an arbitrary integer  $q > 0$ , can be deterministically computed in time  $O(\alpha^2 \log \Delta \cdot T + \alpha \log^* n)$  in the LOCAL model, where  $T$  is the time required to solve the  $(\text{degree} + 1)$ -list coloring problem given an  $O(\Delta^2)$ -coloring in input, and where  $\alpha = O(1 + \log_\Delta q)$ .

If we set  $q = \Theta(\Delta/\varepsilon)$ , we obtain the following corollary.

► **Corollary 11.** For any  $\varepsilon > 0$ , the fractional  $(\Delta + \varepsilon)$ -coloring problem, with support  $O(\Delta^2/\varepsilon)$ , can be solved deterministically in time

$$O\left(\left(\log \Delta + \frac{\log^2(1/\varepsilon)}{\log \Delta}\right) \cdot T + \left(1 + \frac{\log(1/\varepsilon)}{\log \Delta}\right) \cdot \log^* n\right),$$

where  $T$  is the time required to solve the  $(\text{degree} + 1)$ -list coloring problem given an  $O(\Delta^2)$ -coloring in input.

Since the  $(\text{degree} + 1)$ -list coloring problem, given an  $O(\Delta^2)$ -coloring, can be solved in  $O(\sqrt{\Delta} \text{poly log } \Delta)$  rounds deterministically [8, 17, 36], we obtain the following corollary.

► **Corollary 12.** For any constant  $\varepsilon > 0$ , the fractional  $(\Delta + \varepsilon)$ -coloring problem, with support  $O(\Delta^2)$ , can be solved in  $O(\sqrt{\Delta} \text{poly log } \Delta + \log^* n)$  deterministic rounds.

Then, we show that we can obtain a different tradeoff between the support and the running time.

► **Theorem 13.** A  $(q\Delta + 1 : q)$ -coloring, for an arbitrary integer  $q > 0$ , can be deterministically computed in time  $O(q^2 \log \Delta \cdot T + q \log^* n)$  in the LOCAL model, where  $T$  is the time required to solve the  $(\text{degree} + 1)$ -list coloring problem given an  $O(\Delta^2)$ -coloring in input.

If we set  $q = \Theta(1/\varepsilon)$ , since  $T = O(\sqrt{\Delta} \text{poly log } \Delta)$ , we obtain the following corollary, that shows that at the cost of a slightly worse running time, we obtain a better support.

---

<sup>4</sup> We note that trivial adaptations of our algorithms also work for  $\Delta = 2$ .

► **Corollary 14.** *For any  $\varepsilon > 0$ , the fractional  $(\Delta + \varepsilon)$ -coloring problem, with support  $O(\Delta/\varepsilon)$ , can be solved deterministically in time  $O\left(\frac{1}{\varepsilon^2} \cdot \sqrt{\Delta} \cdot \text{poly log } \Delta + \frac{1}{\varepsilon} \cdot \log^* n\right)$ .*

We then prove that, at the cost of drastically increasing the number of colors, it is possible to improve the dependency on  $\Delta$ , and to entirely remove the dependency on  $n$ .

► **Theorem 15.** *For any  $\varepsilon > 0$ , the fractional  $(\Delta + \varepsilon)$ -coloring problem can be solved deterministically in time*

$$O\left(\left(\log \Delta + \frac{\log^2(1/\varepsilon)}{\log \Delta}\right) \cdot \log(\Delta/\varepsilon)\right) = O\left(\log^2 \Delta + \log^2 \frac{1}{\varepsilon} + \frac{\log^3(1/\varepsilon)}{\log \Delta}\right).$$

► **Corollary 16.** *For any  $\varepsilon > 1/\Delta^c$ , where  $c > 0$  is an arbitrary constant, the fractional  $(\Delta + \varepsilon)$ -coloring problem can be solved in  $O(\log^2 \Delta)$  deterministic rounds.*

Our second contribution, presented in Appendix A, is in showing that, by allowing a logarithmic dependency on  $n$  in the running time, we can obtain fractional colorings that are arbitrarily close to the optimum. We provide both randomized and deterministic algorithms, that differ in the required support and in the running time. Let  $p/q = \chi_f(G)$  be the fractional chromatic number of  $G$ . The algorithms that we provide do not require to know  $p$  and  $q$ , but if these values are provided, or even if just the value of  $\chi_f(G)$  is provided, then our algorithms obtain a fractional coloring with smaller support. In particular, if  $p$  and  $q$  are known to the nodes, let  $p' = p$  and  $q' = q$ . Otherwise, let  $p' = \chi \log n/\varepsilon^2$  and  $q' = (1 - \varepsilon)p'/\chi_f(G)$ , where  $\chi = \chi_f(G)$  if  $\chi_f(G)$  is known to the nodes, and  $\Delta + 1$  otherwise. We first show the following.

► **Theorem 17.** *Let  $G = (V, E)$  be a graph that admits a  $(p : q)$  coloring, and let  $t = O(\log n/\varepsilon)$ , for an arbitrary  $\varepsilon > 0$ . There is a randomized LOCAL algorithm that, with high probability, computes a  $(tp' : (1 - \varepsilon)tq')$ -coloring, that is, a fractional  $(1 + O(\varepsilon))\frac{p}{q}$ -coloring, in  $O(\log n/\varepsilon)$  rounds.*

We then show two different deterministic algorithms, that provide different tradeoffs between the support and the running time.

► **Theorem 18.** *Let  $G = (V, E)$  be a graph that admits a  $(p : q)$ -coloring, and let  $t = O(\text{poly } n/\varepsilon)$ , for an arbitrary  $\varepsilon > 0$ . There is a deterministic LOCAL algorithm that computes a  $(tp' : (1 - \varepsilon)tq')$ -coloring, that is, a fractional  $(1 + O(\varepsilon))\frac{p}{q}$ -coloring, in  $O(\log n/\varepsilon)$  rounds.*

► **Theorem 19.** *Let  $G = (V, E)$  be a graph that admits a  $(p : q)$  coloring, and let  $t = O(\log n/\varepsilon)$ , for an arbitrary  $\varepsilon > 0$ . There is a deterministic LOCAL algorithm that computes a  $(tp' : (1 - \varepsilon)tq')$ -coloring, that is, a fractional  $(1 + O(\varepsilon))\frac{p}{q}$ -coloring, in  $O(\log n(\log^2 n + \text{ND})/\varepsilon)$  rounds, where  $\text{ND} \leq \text{poly log } n$  is the time required to compute an  $(O(\log n), O(\log n))$ -network decomposition.*

In Section 6 we prove that the  $O(\log n/\varepsilon)$  time dependency for computing an almost optimal fractional coloring is necessary, even on trees, and even for randomized algorithms.

► **Theorem 20.** *Computing a fractional  $(2 + \varepsilon)$ -coloring on trees in the LOCAL model requires  $\Omega(\log n/\varepsilon)$ , even for randomized algorithms.*

Finally, in Appendix B, we consider grids. In [11], it has been shown that, for any constant  $\varepsilon$  and  $d$ , in  $d$ -dimensional grids, it is possible to compute a fractional  $(2 + \varepsilon)$ -coloring in time  $O(\log^* n)$ . We show that the same problem can be solved in constant time.

► **Theorem 21.** *Let  $G$  be a  $d$ -dimensional grid. For any  $\varepsilon > 0$ , there is a deterministic LOCAL algorithm that computes a fractional  $(2 + \varepsilon)$ -coloring on  $G$ , that runs in  $2^{O(d^2 + d \log \frac{1}{\varepsilon})}$  rounds.*

## 4 Generic results

In this section, we prove some generic statements that will be useful in the following sections.

### 4.1 From partial colorings to fractional colorings

We start by showing that, given an algorithm that computes a partial fractional coloring satisfying that each node has some fixed probability of being colored, then it is possible, at the cost of increasing the total number of colors, to compute a proper fractional coloring.

► **Lemma 22.** *Assume there exists a randomized algorithm  $A(n, \varepsilon)$  that runs in  $T(n, \varepsilon)$  rounds and, with probability at least  $1 - f$ , such that  $f = f(n, \varepsilon) \leq \varepsilon$ , computes a partial  $(p : q)$ -coloring satisfying that each node is uncolored with probability at most  $\varepsilon$ . Then, there exists a randomized algorithm that runs in  $T(n, \varepsilon/4)$ -rounds and, for an arbitrary  $f'$ , with probability at least  $1 - f'$ , computes a  $(p' : q')$ -coloring, where  $p' = pt$ ,  $q' = (1 - \varepsilon)qt$ , and  $t = O(\frac{1}{\varepsilon} \log \frac{n}{f'})$ .*

**Proof.** We run  $A(n, \varepsilon/4)$  for  $t = \frac{6}{\varepsilon} \log \frac{n}{f'}$  times in parallel. Clearly, the running time is still  $T(n, \varepsilon/4)$ . For each execution, we use an independent palette of colors of size  $p$ , and hence we obtain a palette of  $pt$  total colors. We need to prove that, with probability at least  $1 - f'$ , we assign at least  $(1 - \varepsilon)qt$  colors to each node.

Consider an arbitrary node  $u$ . Let  $X_i = 1$  if node  $u$  is uncolored during execution  $i$ , and  $X_i = 0$  otherwise. By assumption, a node is uncolored with probability at most  $\varepsilon/4 + f(n, \varepsilon/4) \leq \varepsilon/2$ , hence  $P(X_i = 1) \leq \varepsilon/2$ . Let  $X = \sum_{i=1}^t X_i$ . By linearity of expectation, we get that  $\mathbb{E}[X] \leq \varepsilon t/2$ . By a Chernoff bound, we get that:

$$P(X \geq \varepsilon t) \leq e^{-\frac{\varepsilon t}{6}} \leq \frac{f'}{n}.$$

By a union bound we get that each node is uncolored in at most  $\varepsilon t$  colorings with probability at least  $1 - f'$ . Hence, with probability at least  $1 - f'$ , each node receives  $q$  colors for at least  $(1 - \varepsilon)t$  times. ◀

### 4.2 From private randomness to shared randomness

We now prove a useful lemma, that allows us to reduce the number of random bits used by a randomized algorithm. We will later use this lemma to derandomize fractional coloring algorithms without increasing the support too much. Note that, in the statement of the lemma, the number of bits used by the original algorithm does not play a role in the resulting algorithm. In fact, as a starting point, we essentially only need to know that the amount of randomness, as a function of  $n$ , can be bounded.

► **Lemma 23.** *Let  $A$  be a randomized algorithm that runs in  $T$  rounds and solves some problem  $P$  with probability of success at least  $1 - f$ , by using  $b = b(n)$  bits of private randomness, where nodes have no private inputs except for their random bit strings and their identifiers. Then, there exists a randomized algorithm  $A'$  that uses only  $O(\log \frac{n}{f})$  bits of shared randomness and solves  $P$  in  $T$  rounds with probability of success at least  $1 - 2f$ .*

**Proof.** The proof follows a standard argument along the lines of Newman's theorem in communication complexity (see, e.g., [30]). Let  $B = Nb$ , where  $N = n^c$ , for some constant  $c \geq 1$ , is the size of the ID space. Note that any algorithm that uses  $b$  bits of private

randomness can be trivially converted into an algorithm that uses  $B$  bits of shared randomness. Also, note that by fixing the string of shared randomness used by the nodes, we obtain a deterministic algorithm.

Let  $\ell = 2^B$ , and let  $R = \{R_1, \dots, R_\ell\}$  be the set of possible shared random bit string assignments. We will prove below, using the probabilistic method, that there is a set  $S \subseteq R$  of size  $k = O(\frac{n^2}{f})$  satisfying that, for any graph  $G$  of  $n$  nodes, algorithm  $A$  fails for at most a  $2f$  fraction of the strings in  $S$ . This means that we can construct an algorithm  $A'$  that, given a bit string of shared randomness of length  $O(\log k) = O(\log \frac{n^2}{f})$ , chooses an element from  $S$  uniformly at random, and then executes the  $T$ -round algorithm  $A$  by using that bit string. This algorithm  $A'$  runs in  $T$  rounds and solves  $P$  with a failure probability of at most  $2f$ .

We now prove that, by choosing  $S$  at random, there is non-zero probability that it satisfies the requirements. We construct  $S = \{s_1, \dots, s_k\}$  by sampling with replacement  $k$  strings from  $R$ . Consider an arbitrary graph  $G$  of  $n$  nodes. Let  $X_i = 1$  if the execution of  $A$  on  $G$  by using the bit string  $s_i$  would fail. Otherwise, let  $X_i = 0$ . Since  $s_i$  has been chosen uniformly at random, and since the failure probability of  $A$  is at most  $f$ , then  $P(X_i = 1) \leq f$ . By linearity of expectation, it holds that  $\mathbb{E}[X] \leq fk$ . Let  $X = \sum_{1 \leq i \leq k} X_i$ . Note that the variables are clearly independent. Hence, by a Chernoff bound, we get that

$$P(X \geq 2fk) \leq e^{-\frac{fk}{3}}.$$

If  $X \geq 2fk$  we say that  $S$  is *bad* for  $G$ . Note that there are at most  $2^{n^2}$  graphs of  $n$  nodes with no private inputs, and that there are at most  $\binom{N}{n}$  possible ID assignments on each graph. Hence, by a union bound, the probability that  $S$  is bad for *some* graph of  $n$  nodes is at most

$$\binom{N}{n} \cdot 2^{n^2} \cdot e^{-\frac{fk}{3}} \leq e^{cn \ln n + n^2 - \frac{fk}{3}}.$$

Choosing  $k = 6n^2/f$  ensures that the above expression is strictly less than 1, for  $n$  sufficiently large. Note that  $k = O(\frac{n^2}{f})$ . ◀

### 4.3 From randomized fractional colorings to deterministic fractional colorings

We now show that, given a randomized algorithm for fractional coloring, it is possible to obtain a deterministic algorithm with the same running time, at the cost of increasing the support.

► **Lemma 24.** *Assume there exists a randomized  $T$ -round algorithm  $A$  that computes a  $(p : q)$ -coloring with probability at least  $1 - f$ . Then, there exists a deterministic  $T$ -round algorithm that computes a  $(p' : q')$ -coloring, where  $p' = pt$ ,  $q' = (1 - 2f)qt$ , and  $t = 2^{O(\log \frac{n}{f})}$ .*

**Proof.** We first apply Lemma 23 to reduce the amount of randomness required by algorithm  $A$  to  $B = O(\log \frac{n}{f})$  bits of shared randomness, obtaining a new algorithm  $A'$  that runs in  $T$  rounds and computes a  $(p : q)$  coloring with probability at least  $1 - 2f$ . We cannot directly run  $A'$ : since nodes are not provided with shared randomness, this is not possible. Instead, we run  $A'$  in parallel for all possible  $t = 2^B$  values of the shared random bit string assignment. In each run, we use an independent palette of  $p$  colors, and hence we use  $pt$  colors in total. Since algorithm  $A'$  succeeds with probability at least  $1 - 2f$ , then in at least a fraction  $1 - 2f$  of the executions all nodes receive at least  $q$  colors. Hence, we obtain that each node receives at least  $(1 - 2f)qt$  colors in total. ◀

## 5 Fast algorithm

In this section, we present an algorithm that is able to compute a fractional  $(\Delta + \varepsilon)$ -coloring, with a running time that depends only polynomially in  $\Delta$ , and that requires small support. Later, we will show how to modify the algorithm to obtain a logarithmic dependency in  $\Delta$ , at the cost of having a much larger support. More formally, we start by proving the following theorem.

► **Theorem 10.** *A  $(q\Delta : q-1)$ -coloring, for an arbitrary integer  $q > 0$ , can be deterministically computed in time  $O(\alpha^2 \log \Delta \cdot T + \alpha \log^* n)$  in the LOCAL model, where  $T$  is the time required to solve the  $(\text{degree} + 1)$ -list coloring problem given an  $O(\Delta^2)$ -coloring in input, and where  $\alpha = O(1 + \log_\Delta q)$ .*

### High level idea

Our algorithm, on a high level, works as follows. We first compute a clustering of the graph by computing an  $(\alpha, \beta)$ -ruling set, for suitable parameters  $\alpha$  and  $\beta$ , and by connecting each node to the cluster centered at the nearest ruling set node. We then proceed by coloring the nodes in a special way. In particular, we compute  $q$  different colorings, such that each coloring is a  $\Delta$ -coloring of the graph, where some nodes are allowed to be uncolored, and such that each node is uncolored in at most one of the  $q$  colorings.

In order to prove that such a coloring can be computed efficiently, we will exploit an important property of the computed clustering: for each cluster, it holds that it either contains many nodes (at least  $q$ ), or it contains a degree-choosable component. We will show that this implies that we can color the nodes such that, in each cluster that contains a degree-choosable component, the  $\Delta$ -coloring problem is solved properly, while in the other clusters we can choose a specific node to leave uncolored. If the cluster is large enough, we can have a different uncolored node for each of the  $q$  colorings, obtaining that each node is uncolored for at most one coloring.

The computed coloring allows us to assign at least  $q - 1$  colors to each node of the graph, where the colors come from a palette of size  $q\Delta$ , and hence we obtain a  $(q\Delta : q - 1)$ -coloring. By choosing the right size of the clusters, we can also prove that the running time is polynomial in  $\Delta$ , and more precisely that it is only slightly worse than the time required to solve the  $(\text{degree} + 1)$ -list coloring problem, which we use as a subroutine.

### 5.1 The clustering

We start by proving that it is possible to compute a clustering of a graph in such a way that it satisfies some desirable properties. In particular, we prove the following lemma.

► **Lemma 25.** *Let  $\alpha = c(1 + \log_\Delta q)$ , for some constant  $c$  and an arbitrary integer  $q > 0$ . It is possible to compute a clustering of a graph  $G$  such that the following holds.*

- *Each cluster has a strong diameter of at most  $2\alpha^2 \log \Delta$ .*
- *Each cluster contains:*
  - *at least  $q$  nodes, or*
  - *a node of degree at most  $\Delta - 1$ , or*
  - *a degree-choosable component, such that all neighbors of the nodes in the degree-choosable component are also contained in the cluster.*

*This clustering can be computed in  $O(\alpha^2 \log \Delta + \alpha \log^* n)$  deterministic rounds.*

In order to prove this lemma, we will use the following lemmas present in the literature.

► **Lemma 26** (Lemma 8 of [19]). *Let  $G = (V, E)$  be a graph and  $v \in V$  be a node such that inside the  $r$ -radius neighborhood of  $v$  there are no degree-choosable components and every node has degree  $\Delta$ . Then, for each even  $r$ , there are at least  $(\Delta - 1)^{r/2}$  nodes at distance  $r$  from  $v$ .*

► **Lemma 27** (Ruling sets, [4, 43]). *A  $(2, \beta)$ -ruling set can be computed in time  $O(\beta\Delta^{2/\beta} + \log^* n)$  deterministic rounds.*

We are now ready to prove Lemma 25.

**Proof.** We start by computing an  $(\alpha, (\alpha - 1)\alpha \log \Delta)$ -ruling set, by computing a  $(2, \alpha \log \Delta)$ -ruling set on  $G^{\alpha-1}$ , the  $(\alpha - 1)$ th power of  $G$ . By applying Lemma 27 with  $\beta = \alpha \log \Delta$ , this ruling set can be computed in  $T = O(\alpha^2 \log \Delta + \alpha \log^* n)$  deterministic rounds. Then, in additional  $O(\alpha^2 \log \Delta)$  rounds, each node finds the nearest ruling set node (breaking ties arbitrarily), and joins its cluster.

Clearly, the running time requirement is satisfied. We need to prove that this clustering satisfies the required properties. The first property is clearly satisfied, since  $\alpha^2 \log \Delta$  is an upper bound on the distance between an arbitrary node and its nearest ruling set node.

We now show that the second property is also satisfied. Let  $v$  be the ruling set node of an arbitrary cluster  $C$ , that is, its center. By the definition of  $(\alpha, \beta)$ -ruling set, all nodes at distance at most  $\lfloor \alpha/2 - 1 \rfloor$  from  $v$  are all contained in  $C$ . Consider the set  $S$  of nodes at distance at most  $k$  from  $v$ , where  $k \in \{\lfloor \alpha/2 - 2 \rfloor, \lfloor \alpha/2 - 3 \rfloor\}$  is even. If there is a node of degree at most  $\Delta - 1$  in  $S$ , then the property is clearly satisfied. Otherwise, all nodes in  $S$  have degree  $\Delta$ , and by Lemma 26 we get that either there is a degree-choosable component in  $S$ , or there are at least  $(\Delta - 1)^{k/2}$  nodes in  $S$ . In the first case, note that the neighbors of nodes in  $S$  are all contained in  $C$ , and hence the property is satisfied. In the second case, we obtain that the cluster contains at least  $(\Delta - 1)^{\lfloor c(1 + \log \Delta q)/2 - 3 \rfloor/2}$  nodes, that, for a large enough constant value of  $c$ , is at least  $q$ , and hence the property is satisfied in this case as well. ◀

## 5.2 The algorithm

We now describe an algorithm that computes a fractional  $(\Delta + \varepsilon)$ -coloring. In the following, we will make use of the notion of partial  $\Delta$ -coloring (see Definition 4).

On a high level, the main idea of the algorithm is to compute  $q$  (possibly) different partial  $\Delta$ -colorings (where each of the colorings come from a different palette), such that each node is uncolored in at most 1 of the colorings. In this way, we can assign  $q - 1$  colors to each node, from a palette of  $q\Delta$  total colors. We now show how the properties stated in Lemma 25 can be used for this purpose.

► **Lemma 28.** *Let  $G$  be a graph that is clustered according to Lemma 25, where each cluster that contains at least  $q$  nodes also contains a special marked node. Let  $T$  be the time required to solve the  $(\text{degree} + 1)$ -list coloring problem, and let  $R$  be the bound on the diameter of the clusters. Then, in  $O(T \cdot R)$  deterministic rounds it is possible to solve the partial  $\Delta$ -coloring problem such that only marked nodes remain uncolored.*

**Proof.** We prove this lemma by slightly modifying the core of the deterministic  $\Delta$ -coloring algorithm presented in Ghaffari et al. [19]. Each node  $v$  starts by spending  $R$  rounds to gather its entire cluster  $C_v$ . In this way, it can see if there is a marked node, or a node with degree at most  $\Delta - 1$ , or a degree-choosable component (at least one of these cases must apply). If there is a marked node  $z$  in  $C_v$ , let  $S_{C_v} = \{z\}$ , otherwise, if there is a node  $z'$

## 18:12 Improved Distributed Fractional Coloring Algorithms

with degree at most  $\Delta - 1$ , then let  $S_{C_v} = \{z'\}$ , otherwise, let  $S_{C_v}$  be the set of nodes of an arbitrary degree-choosable component guaranteed to exist by Lemma 25. Note that, without additional communication,  $v$  can compute its distance from the nearest node  $s$  in  $S_{C_v}$ .

Let  $G_i$  be the subgraph induced by nodes at distance  $i$  from their nearest node in the set, that is,  $G_i = \{u \mid \text{dist}(u, S_{C_u}) = i\}$ . Let  $G_{>i}$  (resp.  $G_{\geq i}$ ) be the graph induced by all nodes contained in  $G_j$ , for all  $j > i$  (resp.  $j \geq i$ ). Note that, for all  $i > 0$ , the nodes of  $G_i$ , in  $G_{\geq i}$ , have degree at most  $\Delta - 1$ , since the maximum degree in  $G$  is  $\Delta$ , and all nodes in  $G_i$  have at least one neighbor in  $G_{i-1}$ . Also, note that  $G_{>R}$  is empty.

We proceed as follows. Assume that  $G_{>i}$  is properly  $\Delta$ -colored (we start with  $i = R$ , where the statement trivially holds, since  $G_{>R}$  is empty), and let  $c(u)$  be the color of a node  $u$  in  $G_{>i}$ . We show that we can  $\Delta$ -color  $G_{\geq i}$ . For each node  $v$  in  $G_i$ , we define its list of available colors as  $L_v = \{1, \dots, \Delta\} \setminus \{c(u) \mid u \in N(v) \cap G_{>i}\}$ . Since the degree of nodes of  $G_i$  in  $G_{\geq i}$  is at most  $\Delta - 1$ , then  $L_v$  defines a (degree + 1)-list coloring instance of  $G_i$ , that can be solved in  $T$  rounds. By iterating this procedure for  $i = R, \dots, 1$ , we obtain that all nodes of  $G$ , except the ones in  $S = \bigcup \{S_{C_v} \mid v \in V\}$ , are properly  $\Delta$ -colored.

Finally, we handle the nodes in  $S$ . If  $S_{C_v}$  contains a marked node, we just leave it uncolored. Otherwise, if  $S_{C_v}$  contains a node with degree at most  $\Delta - 1$ , we color it with an arbitrary available color. Otherwise, if  $S_{C_v}$  contains a degree-choosable component, then, for each node  $u \in S_{C_v}$ , we define  $L_u$  as above. This time,  $L_u$  defines a degree-list coloring instance. Note that, in general, the degree-list coloring problem may be unsolvable, but this is never the case in a degree-choosable component, by definition. Since, for each pair of clusters  $C_1, C_2$ ,  $S_{C_1}$  and  $S_{C_2}$  are non-adjacent, then it is possible to solve all the degree-list coloring instances in parallel, by brute force, in  $R$  rounds. ◀

We are now ready to describe the algorithm. First of all, nodes start by computing an  $O(\Delta^2)$ -coloring. Then, nodes compute the clustering described in Lemma 25. Let  $R = O(\alpha^2 \log \Delta)$  be the maximum diameter of the clusters. Then, nodes spend  $R$  rounds to check the type of their cluster, that is, if there is a degree-choosable component satisfying the required property, or if there are at least  $q$  nodes, or if there is a node with degree at most  $\Delta - 1$ . In all clusters  $C$  containing at least  $q$  nodes, we choose  $q$  arbitrary distinct nodes  $\{v_{C,1}, \dots, v_{C,q}\}$ . Then, we apply Lemma 28 for  $q$  times in parallel. During the application  $i$ , the nodes that are considered marked are  $\{v_{C,i}\}$ . We obtain  $C_1, \dots, C_q$  partial  $\Delta$ -colorings of  $G$ , such that each node is uncolored in at most one coloring. Hence, we use a palette of  $q\Delta$  colors, such that each node has at least  $q - 1$  colors, that is, we obtain a  $(q\Delta : q - 1)$ -coloring.

### Time complexity

The previously described algorithm computes a  $(q\Delta : q - 1)$ -coloring. Hence, in order to prove Theorem 10, we need to give a bound on its running time. Computing the  $O(\Delta^2)$ -coloring can be done in  $O(\log^* n)$  rounds. Computing the clustering requires  $O(\alpha^2 \log \Delta + \alpha \log^* n)$  rounds. Gathering the cluster requires  $O(\alpha^2 \log \Delta)$  rounds. The application of Lemma 28 requires  $O(T \cdot \alpha^2 \log \Delta)$  rounds, where  $T$  is the time for solving a (degree + 1)-list coloring instance given an  $O(\Delta^2)$ -coloring. Recall that  $\alpha = c(1 + \log_{\Delta} q)$ . Hence, we obtain an overall time complexity of  $O(\alpha^2 \log \Delta \cdot T + \alpha \log^* n)$ , where  $\alpha = O(1 + \log_{\Delta} q)$ .

### 5.3 Faster algorithm

We now show that, at the cost of drastically increasing the number of colors, we can improve the dependency on  $\Delta$ , and entirely remove the dependency on  $n$ . In particular, we will prove the following.



► **Theorem 15.** *For any  $\varepsilon > 0$ , the fractional  $(\Delta + \varepsilon)$ -coloring problem can be solved deterministically in time*

$$O\left(\left(\log \Delta + \frac{\log^2(1/\varepsilon)}{\log \Delta}\right) \cdot \log(\Delta/\varepsilon)\right) = O\left(\log^2 \Delta + \log^2 \frac{1}{\varepsilon} + \frac{\log^3(1/\varepsilon)}{\log \Delta}\right).$$

We start by explaining how to remove the  $O(\alpha \log^* n)$  dependency from the algorithm of Theorem 10, obtaining the following intermediate result.

► **Lemma 29.** *For any  $\varepsilon > 0$ , the fractional  $(\Delta + \varepsilon)$ -coloring problem can be solved deterministically in time  $O\left(\left(\log \Delta + \frac{\log^2(1/\varepsilon)}{\log \Delta}\right) \cdot T\right)$ , where  $T$  is the time required to solve the  $(\text{degree} + 1)$ -list coloring problem given an  $O(\Delta^2)$ -coloring in input.*

**Proof.** On a high level, the  $\log^* n$  dependency appears in the time complexity of the algorithm for two reasons:

- The algorithm spends  $O(\alpha \log^* n)$  rounds for computing the clustering, and this is because, in order to compute an  $(\alpha, \beta)$ -ruling set, we first need to find an  $O(\Delta^{2\alpha})$ -coloring of  $G^\alpha$ .
- The algorithm spends  $O(\log^* n)$  rounds to find an  $O(\Delta^2)$  coloring of  $G$ . Note that, given an  $O(\Delta^{2\alpha})$ -coloring of  $G^\alpha$ , we can reduce this runtime to  $O(\log^* \Delta^{2\alpha})$ .

Hence, if, in  $G^\alpha$ , nodes are already provided with an  $O(\Delta^{2\alpha})$ -coloring, then we can get rid of the  $\log^* n$  dependency, obtaining a faster algorithm, that requires only  $O(\alpha^2 \log \Delta \cdot T + \log^* \Delta^{2\alpha}) = O(\alpha^2 \log \Delta \cdot T)$  rounds. Unfortunately, finding such a coloring requires  $\Omega(\log^* n)$  rounds [32]. In order to overcome this issue, we do the following. We first spend a constant number of rounds to find a color randomly, such that each node has a large enough probability of being colored. Then, we run Theorem 10 on the subgraph induced by nodes that are colored correctly. In this way, we obtain an algorithm that computes a partial fractional coloring, satisfying that each node is uncolored with some fixed probability. We can then apply Lemma 22 and Lemma 24 to obtain a deterministic algorithm that finds a proper fractional coloring.

In order to find the required coloring, we proceed as follows. Each node picks a color uniformly at random from a palette of  $c$  colors, for some value  $c$  to be fixed later. Then, each node  $v$  checks its  $\alpha$ -radius neighborhood, and if there is a node  $u$  with the same color, then  $v$  becomes uncolored. For any node  $u$  contained in the  $\alpha$ -radius neighborhood of  $v$ , we have that  $P(u \text{ and } v \text{ pick different colors}) \geq 1 - 1/c$ . Let  $\bar{\Delta} = \Delta^\alpha$  be an upper bound on the degree of  $G^\alpha$ . Then,  $P(u \text{ is colored}) \geq (1 - 1/c)^{\bar{\Delta}}$ , that by the Bernoulli inequality is at least  $1 - \bar{\Delta}/c$ . Hence, by applying the algorithm of Theorem 10 on the subgraph induced by colored nodes, we obtain an algorithm that runs in  $O(\alpha^2 \log \Delta \cdot T)$  rounds and computes a partial  $(q\bar{\Delta} : q-1)$ -coloring where each node is uncolored with probability at most  $\bar{\Delta}/c$ . We can now apply Lemma 22 (note that, since our algorithm never fails, then  $f = 0$ ), obtaining a randomized algorithm that terminates in  $O(\alpha^2 \log \Delta \cdot T)$  rounds and computes a  $(q\bar{\Delta}t : (1 - \bar{\Delta}/c)(q-1)t)$ -coloring, for some  $t$  that depends on the target failure probability  $f'$  and  $n$ . Then, we can apply Lemma 24 to obtain a deterministic algorithm that runs in  $O(\alpha^2 \log \Delta \cdot T)$  rounds and computes a  $(q\bar{\Delta}t' : (1 - 2f')(1 - \bar{\Delta}/c)(q-1)t')$ -coloring, for some  $t'$  that depends on  $f'$  and  $n$ . By fixing  $c = q\bar{\Delta}$  and  $f' = \frac{1}{2q}$ , we obtain a deterministic algorithm for computing a fractional  $\frac{q\bar{\Delta}}{(1-1/q)^2(q-1)}$ -coloring, that, for  $q = O(\Delta/\varepsilon)$ , is a fractional  $(\Delta + O(\varepsilon))$ -coloring. The running time, for such a value of  $q$ , is  $O((\log \Delta + \log^2(1/\varepsilon)/\log \Delta) \cdot T)$ , as required. ◀

**Proof of Theorem 15**

**Proof.** We now explain how to remove the dependency on  $T$ , and obtain Theorem 15. Let  $q = \Theta(\Delta/\varepsilon)$ . In order to obtain a faster algorithm, we start by showing that we can replace the dependency on  $T$  in Lemma 28, with  $O(\log q)$ , at the cost of leaving some nodes uncolored. Hence, we obtain a “sloppy” version of Lemma 28. Recall that  $T$  is the time required to run a procedure that solves a  $(\text{degree} + 1)$ -list coloring instance. Instead of running such a procedure, we run a procedure that partially solves an instance in  $O(\log q) = O(\log \frac{\Delta}{\varepsilon})$  rounds, such that a node remains uncolored with probability at most  $1/q$ . This can be achieved by letting each node try to pick an available color uniformly at random for  $O(\log q)$  times [25].

We now show that it is possible to obtain an algorithm that computes a partial  $(\Delta : 1)$ -coloring satisfying that each node is uncolored with probability at most  $2/q$ . First, we compute the clustering as in the original algorithm, that is, by applying Lemma 25. Then, on clusters of size at least  $q$ , we mark a node of the cluster chosen uniformly at random. By applying the sloppy version of Lemma 28, we obtain what we need.

Finally, by applying Lemma 22 and Lemma 24, we obtain a deterministic algorithm that solves fractional  $(\Delta + O(\varepsilon))$ -coloring in time  $O((\log \Delta + \frac{\log^2(1/\varepsilon)}{\log \Delta}) \cdot \log(\Delta/\varepsilon))$ , proving the theorem.  $\blacktriangleleft$

**5.4 Better support**

We now provide a different algorithm, that has slightly worse running time compared to the one of Theorem 10, but that is able to give a fractional  $(\Delta + \varepsilon)$ -coloring with smaller support. In particular, we will prove the following theorem.

► **Theorem 13.** *A  $(q\Delta + 1 : q)$ -coloring, for an arbitrary integer  $q > 0$ , can be deterministically computed in time  $O(q^2 \log \Delta \cdot T + q \log^* n)$  in the LOCAL model, where  $T$  is the time required to solve the  $(\text{degree} + 1)$ -list coloring problem given an  $O(\Delta^2)$ -coloring in input.*

We will exploit the following lemma, first presented in [2], and used also in [11].

► **Lemma 30** (Proposition 8 of [2]). *Let  $q \geq 1$  be an integer and let  $P = (v_1, \dots, v_{2q+1})$  be a path. Assume that for  $i \in \{1, 2q + 1\}$  the vertex  $v_i$  has a list  $L_{v_i}$  of at least  $q + 1$  colors, and for any  $2 \leq i \leq 2q$ ,  $v_i$  has a list  $L_{v_i}$  of at least  $2q + 1$  colors. Then, each vertex  $v_i$  of  $P$  can be assigned a subset  $S_i \subseteq L_{v_i}$  of  $q$  colors, so that adjacent vertices are assigned disjoint sets.*

Similarly as in the algorithm of Theorem 10, we start by computing an  $(\alpha, (\alpha - 1)\alpha \log \Delta)$ -ruling set, by computing a  $(2, \alpha \log \Delta)$ -ruling set on  $G^{\alpha-1}$ , the  $(\alpha - 1)$ -th power of  $G$ . This time, we choose  $\alpha = 4q + 4$ . Then, we also compute an  $O(\Delta^2)$  coloring. Then, we form clusters by letting each node  $u$  join the cluster  $C_v$  centered at the nearest ruling set node  $v$ . By the definition of  $(\alpha, \beta)$ -ruling set, all nodes at distance at most  $\lfloor \alpha/2 - 1 \rfloor$  from  $v$  are contained in  $C_v$ . Hence, in  $C_v$ , there exists at least one induced path of  $\lfloor \alpha/2 - 1 \rfloor$  nodes satisfying that all neighbors of nodes in  $P$  are fully contained in  $C_v$  (that is, the path obtained by taking  $v$  and the nodes contained in some shortest path from  $v$  to some node at distance  $\lfloor \alpha/2 - 1 \rfloor$  from  $v$ ). Note that  $\lfloor \alpha/2 - 1 \rfloor = 2q + 1$ . Let  $S_v$  be the set of nodes of the path.

Similarly as in the proof of Lemma 28, we can spend  $O(\alpha^2 \log \Delta \cdot T)$  rounds to find a partial  $\Delta$ -coloring that leaves uncolored only nodes in  $\bigcup S_v$ . This partial coloring can be trivially converted into a partial  $(q\Delta : q)$ -coloring where each colored node has exactly  $q$  colors. Then, for each node  $u$  contained in some  $S_v$ , we can define its list of available colors

as  $\{1, \dots, q\Delta + 1\} \setminus \{C(z) \mid z \in N(u)\}$ , where  $C(z)$  is the set of colors assigned to node  $z$ . Note that, from the list of available colors of  $u$ , we removed at most  $q$  colors for each neighbor of  $u$ . Hence, the endpoints of the path satisfy  $|L_u| \geq q + 1$  since they have at most  $\Delta - 1$  colored neighbors, and the inner nodes satisfy  $|L_u| \geq 2q + 1$  since they have at most  $\Delta - 2$  colored neighbors. Hence, by applying Lemma 30 we get that nodes can complete the coloring by brute force, since paths are not connected to each other.

We spend  $O(q^2 \log \Delta + q \log^* n)$  rounds to compute a ruling set,  $O(\log^* n)$  rounds to compute the  $O(\Delta^2)$  coloring. The rest of the algorithm requires  $O(q^2 \log \Delta \cdot T)$  rounds. Each node receives  $q$  colors, from a palette of total  $q\Delta + 1$  colors. Hence, the theorem follows.

## 6 Lower bound

In this section, we show a lower bound of  $\Omega(\log n/\varepsilon)$  rounds for computing a fractional  $(2 + \varepsilon)$ -coloring, which holds already on trees, and even for randomized algorithms. We first show that there exist graphs with girth  $\Omega(\log n/\varepsilon)$  and fractional chromatic number strictly larger than  $2 + \varepsilon$ . Then we argue that, if we take an algorithm that, on trees, achieves a fractional  $(2 + \varepsilon)$ -coloring in  $o(\log n/\varepsilon)$  rounds, and we execute it on such graphs, it must fail, since the obtained fractional coloring would be too good. We get our contradiction on the existence of such an algorithm by arguing that, in time  $o(\log n/\varepsilon)$ , a node cannot distinguish whether it is on a tree or on these graphs.

We start by describing a graph family of interest. Let  $\mathcal{G}^*$  be a graph family that contains all graphs with  $n$  nodes,  $m = O(n)$  edges, girth  $\Omega(\log n)$ , and where the largest independent set has size at most  $n/c$ , for some large constant  $c$ , and for infinite values of  $n$ . Such a family of graphs is known to exist [34]. Starting from a graph  $G = (V, E) \in \mathcal{G}^*$ , we construct a graph  $H$  by replacing all edges of  $G$  with a path of length  $2k + 1$ , for some  $k = \Theta(1/\varepsilon)$ . In other words, let  $e = \{u, v\}$  be an edge in  $G$ . We replace  $e$  by a path  $P^e = (u, w_1^e, w_2^e, \dots, w_{2k}^e, v)$ . We refer to nodes in  $P^e$  as *path nodes*, and we refer to the  $w_1^e, w_2^e, \dots, w_{2k}^e$  nodes as *inner path nodes*. Let  $\mathcal{G}$  be the family that contains all such graphs. By construction, a graph  $H \in \mathcal{G}$  has  $N = n + 2km$  nodes and girth  $\Omega(\log n/\varepsilon)$ . We now show the following lemma about the fractional chromatic number of these graphs.

► **Lemma 31.** *Let  $H$  be a graph in  $\mathcal{G}$ . The fractional chromatic number of  $H$  is strictly larger than  $2/(1 - c'\varepsilon)$ , for some constant  $c' > 0$ .*

**Proof.** Let  $S$  be any independent set of  $H$ . We modify  $S$  and compute a new independent set  $S'$  of  $H$  such that  $|S'| \geq |S|$  and  $S'$  contains exactly  $mk$  inner path nodes. Note that this implies that for each path  $P^e = (u, w_1^e, w_2^e, \dots, w_{2k}^e, v)$ , at most one of the two endpoints can be in  $S'$ . Let  $P^e = (u, w_1^e, w_2^e, \dots, w_{2k}^e, v)$  be a path in  $H$  where at most  $k - 1$  inner path nodes are in  $S$ . There are two cases: either at most one of the two endpoints is in  $S$ , or both  $u$  and  $v$  are in  $S$ .

In the former case, we modify  $S$  in the following way. W.l.o.g., let  $u$  be a node not in  $S$ . We start from  $u$  and compute an optimal independent set inside  $P^e$  sequentially, by starting with  $w_1^e$  in the independent set and then putting every other node in the set. This procedure puts in the independent set  $k$  inner path nodes of  $P^e$ , and note that the obtained set is still independent.

In the latter case, we remove node  $u$  from the set, and then we proceed as in the former case. Note that the obtained set is still independent, and it is not smaller. In fact, before the modification, there were at most  $k - 1$  nodes of  $P^e \setminus \{u, v\}$  in  $S$ , and hence at most  $k$  nodes of  $P^e \setminus \{v\}$  in  $S$ , and after the modification we have  $k$  nodes of  $P^e \setminus \{u, v\}$  in the set.

## 18:16 Improved Distributed Fractional Coloring Algorithms

We call  $S'$  the independent set resulting from the above operations. Notice that  $|S'| \geq |S|$ . Recall that each graph in  $\mathcal{G}$  is obtained from a graph  $G \in \mathcal{G}^*$  where the largest independent set has size at most  $n/c$ . Note also that, by construction, if we project  $S'$  onto  $G$ , then we obtain an independent set of  $G$ . Hence,

$$\begin{aligned} |S'| &\leq \frac{n}{c} + mk = \frac{N}{2} - \frac{(c-2)n}{2c} = \frac{N}{2} \left( 1 - \frac{2(c-2)n}{2cN} \right) = \frac{N}{2} \left( 1 - \frac{(c-2)n}{c(n+2km)} \right) \\ &= \frac{N}{2} \left( 1 - \frac{(c-2)n}{c(n+2\frac{c_1}{\varepsilon}c_2n)} \right) \text{ where } k = \frac{c_1}{\varepsilon} \text{ and } m = c_2n \\ &= \frac{N}{2} \left( 1 - \frac{c-2}{c(1+2\frac{c_1}{\varepsilon}c_2)} \right) < \frac{N}{2} (1 - c_3\varepsilon) \text{ for some } c_3 > 0 \text{ that depends on } c_1 \text{ and } c. \end{aligned}$$

Hence, any color can be assigned to strictly less than a fraction  $(1 - c_3\varepsilon)/2$  of the nodes, implying that the fractional chromatic number of  $H$  is strictly larger than  $2/(1 - c_3\varepsilon)$ . ◀

From the above lemma we get the following corollary.

► **Corollary 32.** *There exists an infinite family of graphs of girth  $\Omega(\log n/\varepsilon)$  and fractional chromatic number strictly larger than  $2 + \varepsilon$ .*

We are now ready to prove our main theorem.

► **Theorem 33.** *Computing a  $(2 + \varepsilon)$ -fractional coloring on trees in the LOCAL model requires  $\Omega(\log n/\varepsilon)$ , even for randomized algorithms.*

**Proof.** The proof follows a standard indistinguishability argument, already used to prove, e.g., that coloring trees with  $o(\Delta/\log \Delta)$  colors requires  $\Omega(\log_{\Delta} n)$  rounds, even for randomized algorithms [32]. For simplicity we prove our claim for the deterministic case, but it can be extended to the randomized case with standard techniques.

Suppose there exists an algorithm  $A_T$  that, on trees, computes a fractional  $(2 + \varepsilon)$ -coloring in  $o(\log n/\varepsilon)$  rounds. Let  $G$  be a graph satisfying Corollary 32. In  $o(\log n/\varepsilon)$  rounds, each node in  $G$  does not see any cycle, and hence we could run algorithm  $A_T$  on  $G$  and it would not notice that it is being run on a graph that is not a tree. However,  $A_T$  must fail on  $G$ , since, by Corollary 32, the fractional chromatic number of  $G$  is strictly larger than  $2 + \varepsilon$ . Hence, suppose we run  $A_T$  on  $G$  and it fails on the neighboring nodes  $u$  and  $v$  who, as an output of  $A_T$ , got some common color (note that the algorithm may also fail on just one node by assigning to it too few colors, but the lower bound argument follows in the same way).

Recall that a  $t$ -round algorithm in the LOCAL model can be seen as a mapping from  $t$ -radius neighborhoods into outputs. Let  $B_u$  and  $B_v$  be the views of radius  $t$  of nodes  $u$  and  $v$ , respectively, that  $A_T$  then maps into the outputs of  $u$  and  $v$ . Let  $B = B_u \cup B_v$ . The subgraph in  $B$  does not contain cycles and it has radius  $o(\log n/\varepsilon)$ . Starting from  $B$ , we construct a tree that contains  $B$ , and where we add additional nodes in order to obtain an  $n$ -node tree  $T$  (note that the additional nodes can be added without altering the  $t$ -radius views of  $u$  and  $v$ , since, in  $B$ , there exists at least one node at distance at least  $t$  from both of them). Nodes  $u$  and  $v$  in  $T$  have the same exact view as in  $G$ , hence they output the same improper fractional coloring, meaning that  $A_T$  fails on  $T$ , which is a contradiction. Hence,  $A_T$  cannot exist, proving the theorem. ◀

## References

- 1 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.
- 2 Yves Aubry, Jean-Christophe Godin, and Olivier Togni. Every triangle-free induced subgraph of the triangular lattice is  $(5m, 2m)$ -choosable. *Discret. Appl. Math.*, 166:51–58, 2014.
- 3 B. Awerbuch, A. V. Goldberg, M. Luby, and S. A. Plotkin. Network decomposition and locality in distributed computation. In *Proc. 30th IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 364–369, 1989.
- 4 Alkida Balliu, Sebastian Brandt, and Dennis Olivetti. Distributed lower bounds for ruling sets. In *61st IEEE Annual Symposium on Foundations of Computer Science, (FOCS)*, pages 365–376, 2020.
- 5 L. Barenboim. Deterministic  $(\Delta + 1)$ -coloring in sublinear (in  $\Delta$ ) time in static, dynamic, and faulty networks. *Journal of the ACM*, 63(5):1–22, 2016.
- 6 L. Barenboim and M. Elkin. Deterministic distributed vertex coloring in polylogarithmic time. In *Proc. 29th Symp. on Principles of Distributed Computing (PODC)*, pages 410–419, 2010.
- 7 L. Barenboim, M. Elkin, and C. Gavoille. A fast network-decomposition algorithm and its applications to constant-time distributed computation. In *Proc. 22nd Coll. on Structural Information and Communication Complexity (SIROCCO)*, pages 209–223, 2015.
- 8 L. Barenboim, M. Elkin, and U. Goldenberg. Locally-iterative distributed  $(\Delta + 1)$ -coloring below Szegedy-Vishwanathan barrier, and applications to self-stabilization and to restricted-bandwidth models. In *Proc. 37th ACM Symp. on Principles of Distributed Computing (PODC)*, pages 437–446, 2018.
- 9 L. Barenboim, M. Elkin, and F. Kuhn. Distributed  $(\Delta + 1)$ -coloring in linear (in  $\Delta$ ) time. *SIAM Journal on Computing*, 43(1):72–95, 2015.
- 10 L. Barenboim, M. Elkin, S. Pettie, and J. Schneider. The locality of distributed symmetry breaking. *Journal of the ACM*, 63:20:1–20:45, 2016.
- 11 Nicolas Bousquet, Louis Esperet, and Fraigniaudnçois Piro. Distributed algorithms for fractional coloring. In *Proc. 28th Coll. on Structural Information and Communication Complexity (SIROCCO)*, pages 15–30, 2021.
- 12 S. Brandt, O. Fischer, J. Hirvonen, B. Keller, T. Lempäinen, J. Rybicki, J. Suomela, and J. Uitto. A lower bound for the distributed Lovász local lemma. In *Proc. 48th ACM Symp. on Theory of Computing (STOC)*, pages 479–488, 2016.
- 13 Y.-J. Chang, W. Li, and S. Pettie. An optimal distributed  $(\Delta + 1)$ -coloring algorithm? In *Proc. 50th ACM Symp. on Theory of Computing (STOC)*, 2018.
- 14 M. Elkin and O. Neiman. Distributed strong diameter network decomposition. In *Proc. 35th ACM Symp. on Principles of Distributed Computing (PODC)*, pages 211–216, 2016.
- 15 Michael Elkin, Seth Pettie, and Hsin-Hao Su.  $(2\Delta - 1)$ -edge-coloring is much easier than maximal matching in the distributed setting. In *Proc. 26th ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 355–370, 2015.
- 16 Salwa Faour and Fabian Kuhn. Approximating Bipartite Minimum Vertex Cover in the CONGEST Model. In *24th International Conference on Principles of Distributed Systems (OPODIS)*, pages 29:1–29:16, 2021.
- 17 P. Fraigniaud, M. Heinrich, and A. Kosowski. Local conflict coloring. In *Proc. 57th IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 625–634, 2016.
- 18 M. Ghaffari, D. Harris, and F. Kuhn. On derandomizing local distributed algorithms. In *Proc. 59th Annual Symposium on Foundations of Computer Science, (FOCS)*, pages 662–673, 2018.
- 19 Mohsen Ghaffari, Juho Hirvonen, Fabian Kuhn, and Yannic Maus. Improved distributed  $\Delta$ -coloring. *Distributed Comput.*, 34(4):239–258, 2021.
- 20 Mohsen Ghaffari and Fabian Kuhn. Deterministic distributed vertex coloring: Simpler, faster, and without network decomposition. In *Proc. 62nd IEEE Symp. on Foundations of Computer Science (FOCS)*, 2021.

- 21 A. V. Goldberg, S. A. Plotkin, and G. E. Shannon. Parallel symmetry-breaking in sparse graphs. *SIAM Journal on Discrete Mathematics*, 1(4):434–446, 1988.
- 22 Magnús M. Halldórsson, Alexandre Nolin, and Tigran Tonoyan. Ultrafast distributed coloring of high degree graphs. *CoRR*, abs/2105.04700, 2021. [arXiv:2105.04700](https://arxiv.org/abs/2105.04700).
- 23 D. G. Harris, J. Schneider, and H.-H. Su. Distributed  $(\Delta + 1)$ -coloring in sublogarithmic rounds. In *Proc. 48th ACM Symp. on Theory of Computing (STOC)*, 2016.
- 24 Henning Hasemann, Juho Hirvonen, Joel Rybicki, and Jukka Suomela. Deterministic local algorithms, unique identifiers, and fractional graph colouring. *Theor. Comput. Sci.*, 610:204–217, 2016.
- 25 Öjvind Johansson. Simple distributed  $\Delta+1$ -coloring of graphs. *Inf. Process. Lett.*, 70(5):229–232, 1999.
- 26 K. Kothapalli, M. Onus, C. Scheideler, and C. Schindelhauer. Distributed coloring in  $O(\sqrt{\log n})$  bit rounds. In *Proc. 20th IEEE Int. Parallel and Distributed Processing Symp. (IPDPS)*, 2006.
- 27 F. Kuhn. Local multicoloring algorithms: Computing a nearly-optimal TDMA schedule in constant time. In *Proc. Symp. on Theoretical Aspects of Computer Science (STACS)*, pages 613–624, 2009.
- 28 F. Kuhn. Faster deterministic distributed coloring through recursive list coloring. In *Proc. 32st ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 1244–1259, 2020.
- 29 F. Kuhn and R. Wattenhofer. On the complexity of distributed graph coloring. In *Proc. 25th ACM Symp. on Principles of Distributed Computing (PODC)*, pages 7–15, 2006.
- 30 Eyal Kushilevitz and Noam Nisan. *Communication Complexity*. Cambridge University Press, 1996.
- 31 N. Linial. Distributive graph algorithms – global solutions from local data. In *Proc. 28th IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 331–335, 1987.
- 32 N. Linial. Locality in distributed graph algorithms. *SIAM Journal on Computing*, 21(1):193–201, 1992.
- 33 N. Linial and M. Saks. Low diameter graph decompositions. *Combinatorica*, 13(4):441–454, 1993.
- 34 Alexander Lubotzky, Ralph Phillips, and Peter Sarnak. Ramanujan graphs. *Comb.*, 8(3):261–277, 1988.
- 35 M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM Journal on Computing*, 15:1036–1053, 1986.
- 36 Y. Maus and T. Tonoyan. Local conflict coloring revisited: Linial for lists. In *Proc. 34th Int. Symp. on Distributed Computing (DISC)*, pages 16:1–16:18, 2020.
- 37 Yannic Maus. Distributed graph coloring made easy. In *Proc. 33rd ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*, pages 362–372, 2021.
- 38 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, 2013.
- 39 Moni Naor. A lower bound on probabilistic algorithms for distributive ring coloring. *SIAM J. on Discrete Math.*, 4(3):409–412, 1991.
- 40 A. Panconesi and A. Srinivasan. Improved distributed algorithms for coloring and network decomposition problems. In *Proc. 24th ACM Symp. on Theory of Computing (STOC)*, pages 581–592, 1992.
- 41 A. Panconesi and A. Srinivasan. The local nature of  $\Delta$ -coloring and its algorithmic applications. *Combinatorica*, 15(2):255–280, 1995.
- 42 V. Rozhoň and M. Ghaffari. Polylogarithmic-time deterministic network decomposition and distributed derandomization. In *Proc. 52nd ACM Symp. on Theory of Computing (STOC)*, pages 350–363, 2020.
- 43 Johannes Schneider, Michael Elkin, and Roger Wattenhofer. Symmetry breaking depending on the chromatic number or the neighborhood growth. *Theor. Comput. Sci.*, 509:40–50, 2013.

- 44 Johannes Schneider and Roger Wattenhofer. A new technique for distributed symmetry breaking. In *Proc. 29th ACM Symp. on Principles of Distributed Computing (PODC)*, pages 257–266, 2010.
- 45 M. Szegedy and S. Vishwanathan. Locality based graph coloring. In *Proc. 25th ACM Symp. on Theory of Computing (STOC)*, pages 201–207, 1993.

## A Approximating the fractional chromatic number

In this section, we show that it is possible to find arbitrarily good approximations of the fractional chromatic number. In particular, given a graph  $G$  with fractional chromatic number  $\chi_f(G) = p/q$ , we provide randomized and deterministic algorithms that are able to find a fractional  $(1 + \varepsilon)\chi_f(G)$ -coloring, for any  $\varepsilon > 0$ . Our algorithms use a different amount of total colors, depending on whether the nodes know  $p$  and  $q$  or not, or if they know  $\chi_f(G)$ .

On a high level, we show that it is possible to cluster a graph such that each node is unclustered with probability at most  $\varepsilon$ , and such that any pair of clusters is at least 2 hops apart (that is, for any two nodes that are in different clusters it holds that they do not share an edge). In this way, inside each cluster, we can optimally solve the fractional  $\chi_f(G)$ -coloring, or find a good-enough approximation if  $p$  and  $q$  are not known. Then, by applying Lemma 22 and Lemma 24, we obtain randomized and deterministic algorithms for computing a fractional coloring that approximates the fractional chromatic number.

### A.1 Computing a clustering

In order to obtain a clustering of the graph, we slightly modify the clustering algorithm of Miller, Peng, and Xu [38] (MPX), to make it compute clusters that are 2 hops apart, such that each node is unclustered with probability at most  $\varepsilon$ , and each cluster has weak diameter  $O(\log n/\varepsilon)$ .

► **Lemma 34.** *Given a graph  $G = (V, E)$ , there is a randomized algorithm that computes a 2-hops-apart clustering of  $G$  such that each node is unclustered with probability at most  $\varepsilon$ , and each cluster has weak diameter  $O(\log n/\varepsilon)$  with high probability. This algorithm terminates in  $O(\log n/\varepsilon)$  rounds with high probability.*

**Proof.** Since we are going to use a modified version of the MPX procedure, we start by describing the standard MPX procedure.

Each node  $u$  chooses independently a *shift*  $\delta_u$  from an exponential distribution with parameter  $\gamma = \varepsilon/2$ . Let the *shifting distance* from  $u$  to  $v$  be denoted as  $\text{dist}_{-\delta}(u, v) = \text{dist}(u, v) - \delta_u$ . Each node  $v$  is then assigned to the cluster  $C_u$  centered at the node  $u$  that, among all nodes in  $G$ , minimizes the value of the shifted distance  $\text{dist}_{-\delta}(u, v)$ , breaking ties arbitrarily. Miller, Peng, and Xu [38] proved that, with high probability, each cluster has radius  $O(\log n/\varepsilon)$ . While the original procedure is designed to work in the PRAM model, it is folklore that it can be easily converted into a distributed algorithm that terminates in  $O(\log n/\varepsilon)$  rounds with high probability. This procedure obtains a partitioning of the vertices into clusters satisfying the following properties.

- Each cluster is connected, that is, for any two nodes  $u$  and  $v$  it holds that, if both are in the same cluster  $C$ , then the nodes in the shortest path between  $u$  and  $v$  are also in  $C$ .
- Each cluster has strong diameter  $O(\log n/\varepsilon)$  with high probability.
- Each edge has probability at most  $\varepsilon$  to be an intercluster edge.

We run this procedure, and then we modify the obtained clustering in the following way.<sup>5</sup> Let  $\{u, v\} \in E$  be an edge such that its endpoints  $u$  and  $v$  are on different clusters. For each such edge, we choose the endpoint with the smallest identifier and we remove it from the cluster. We say that these nodes are *unclustered*. After this operation it holds that, for any two nodes that are in different clusters, they do not share an edge, meaning that the clusters are at least 2 hops apart, as desired. Notice that, by removing nodes from the clusters we may lose the guarantee on the strong diameter of the clusters. However, it still holds that the *weak* diameter of each cluster is  $O(\log n/\varepsilon)$  w.h.p., and this is enough for our purposes.

What is left to show is that each node is unclustered with probability at most  $\varepsilon$ . Consider an unclustered node  $u$ . Let  $C_{u'}$  be the cluster centered at node  $u'$  where  $u$  belonged to before being removed. Since  $u$  is unclustered, it means that there is a node  $w$  neighbor of  $u$  such that, before removing nodes from the clusters, nodes  $u$  and  $w$  were assigned to different clusters (otherwise  $u$  would not be unclustered). Let  $C_{w'}$  be the cluster centered at  $w'$  that was initially assigned to node  $w$ . By construction of these clusters, we have that  $\text{dist}_{\delta}(u', u) \leq \text{dist}_{\delta}(w', w) + 1$  and, at the same time,  $\text{dist}_{\delta}(w', w) \leq \text{dist}_{\delta}(u', u) + 1$ . Hence,  $|\text{dist}_{\delta}(u', u) - \text{dist}_{\delta}(w', w)| \leq 1$ , implying that  $|\text{dist}_{\delta}(u', u) - \text{dist}_{\delta}(w', u)| \leq 2$ . In other words, the absolute value of the difference between the smallest and the second smallest shifting distance of an unclustered node is at most 2. In [38] it has been proven that, for each node, the probability that this event happens is at most  $2\gamma = \varepsilon$ . We thus obtain that each node is unclustered with probability at most  $\varepsilon$ . ◀

In the remaining of this section we use the variables  $p'$  and  $q'$ , that are defined as follows.

- If  $p$  and  $q$  are known to the nodes, then  $p' = p$  and  $q' = q$ .
- Otherwise, let  $p' = \chi c \log n / \varepsilon^2$  and  $q' = (1 - \varepsilon)p' / \chi_f(G)$ , where  $\chi = \chi_f(G)$  if  $\chi_f(G)$  is known to the nodes, and  $\Delta + 1$  otherwise.

## A.2 Solving a partial fractional coloring

We now prove that, by using the clustering algorithm of Lemma 34, it is possible to find a partial  $(p' : q')$ -coloring.

► **Lemma 35.** *There exists a randomized  $O(\log n/\varepsilon)$ -round algorithm  $A$  that computes a partial  $(p' : q')$ -coloring satisfying that, with probability at least  $1 - 1/n$ , each node is uncolored with probability at most  $\varepsilon$ .*

**Proof.** Note that the algorithm described in Lemma 34 is Las Vegas, but we can turn it into a Monte Carlo algorithm by truncating its execution after  $O(\log n/\varepsilon)$  steps, and leave unclustered every node that did not terminate. Since the original algorithm terminates in  $O(\log n/\varepsilon)$  rounds with high probability (that is, at least  $1 - 1/n$ ), then this new algorithm always terminates in  $O(\log n/\varepsilon)$  rounds, which is also an upper bound on the diameter of the clusters, and leaves each node unclustered with probability at most  $\varepsilon + 1/n$  by a union bound. Hence, by slightly scaling  $\varepsilon$ , we obtain the same guarantees as the original algorithm.

Hence, we start by running the (Monte Carlo variant of the) clustering algorithm. Then, since each cluster has weak diameter at most  $R = O(\log n/\varepsilon)$ , we can spend  $R$  rounds for computing in parallel, in each cluster, by brute force, a  $(p' : q')$ -coloring (we will later argue why such a coloring always exists). Note that this is possible even if  $q'$  is not known to the nodes, as they can just find the best possible solution. Since unclustered nodes do not get a color, and since clusters are 2 hops apart, then there are no neighboring nodes that get the same color.

<sup>5</sup> A similar modification has been used, for example, in [16].



We need to argue why a  $(p' : q')$ -coloring always exists. Let  $G$  be a graph that is  $(p : q)$ -colored. We show that there is a randomized process that, with non-zero probability, produces a  $(p' : q')$ -coloring. By the probabilistic method, this implies that  $G$  admits a  $(p' : q')$ -coloring, and hence that also each cluster admits a  $(p' : q')$ -coloring.

We sample with replacement  $p'$  colors from  $p$  colors. Consider an arbitrary node  $u$ . Let  $X_i = 1$  if, during the  $i$ -th sampling, we sample a color that node  $u$  has among its colors. Otherwise, let  $X_i = 0$ . Since node  $u$  has at least  $q$  out of the  $p$  possible colors, then  $P(X_i = 1) \geq q/p$ . Let  $X = \sum_{i=1}^{p'} X_i$ . By linearity of expectation, it holds that  $\mathbb{E}[X] \geq \frac{p'q}{p} = \frac{p'}{\chi_f(G)} = q'/(1 - \varepsilon)$ . By a Chernoff bound, we get that

$$P(X \leq q') \leq e^{-\frac{\varepsilon^2}{2} \cdot \frac{p'}{\chi_f(G)}} = e^{-\frac{\varepsilon^2 c \chi \log n}{\varepsilon^2 2 \chi_f(G)}} \leq n^{-c/2}.$$

By a union bound, we have that each node has less than  $q'$  colors with probability at most  $n^{1-c/2}$ . By choosing  $c \geq 4$  we get that each node has at least  $q'$  colors with probability at least  $1 - 1/n$ . ◀

### A.3 Putting things together

We now prove the existence of randomized and deterministic algorithms for approximating the fractional chromatic number, with running time  $O(\log n/\varepsilon)$ . Note that, for  $\varepsilon < 1/n$ , we can trivially solve the problem by gathering the entire graph on a node and brute forcing a solution. Hence, in the following, assume  $\varepsilon \geq 1/n$ .

Lemma 35 guarantees the existence of a randomized algorithm that runs in  $O(\log n/\varepsilon)$  rounds, and with probability at least  $1 - f$ , where  $f = 1/n$ , computes a partial  $(p' : q')$ -coloring satisfying that each node is uncolored with probability at most  $\varepsilon$ . By applying Lemma 22, we obtain that there exists a randomized algorithm, that also runs in  $O(\log n/\varepsilon)$  rounds and, with probability at least  $1 - f'$ , where  $f' = 1/n^c$  for an arbitrary constant  $c \geq 1$ , computes a partial  $(p'' : q'')$ -coloring, where  $p'' = p't$ ,  $q'' = (1 - \varepsilon)q't$ , and  $t = O(\log n/\varepsilon)$ . Hence, we obtain the following.

► **Theorem 17.** *Let  $G = (V, E)$  be a graph that admits a  $(p : q)$  coloring, and let  $t = O(\log n/\varepsilon)$ , for an arbitrary  $\varepsilon > 0$ . There is a randomized LOCAL algorithm that, with high probability, computes a  $(tp' : (1 - \varepsilon)tq')$ -coloring, that is, a fractional  $(1 + O(\varepsilon))_q^p$ -coloring, in  $O(\log n/\varepsilon)$  rounds.*

Then, starting from this algorithm, we can apply Lemma 24, where  $f = 1/n^c$  for an arbitrary constant  $c \geq 1$ , and obtain that there exists a deterministic algorithm that also runs in  $O(\log n/\varepsilon)$  rounds, and computes a partial  $(p''' : q''')$ -coloring, where  $p''' = p''t'$ ,  $q''' = (1 - 2f)q''t'$ , and  $t' = \text{poly } n$ . Since  $\varepsilon \geq 1/n \geq f$ , then this means that, compared to the randomized algorithm, we lose at most an  $O(\varepsilon)$  fraction of colors. Hence, we obtain the following.

► **Theorem 18.** *Let  $G = (V, E)$  be a graph that admits a  $(p : q)$ -coloring, and let  $t = O(\text{poly } n/\varepsilon)$ , for an arbitrary  $\varepsilon > 0$ . There is a deterministic LOCAL algorithm that computes a  $(tp' : (1 - \varepsilon)tq')$ -coloring, that is, a fractional  $(1 + O(\varepsilon))_q^p$ -coloring, in  $O(\log n/\varepsilon)$  rounds.*

### A.4 Less colors

We now show an alternative way for derandomizing the algorithm of Theorem 17, obtaining the same guarantees on the number of colors, but with a slightly worse running time. For this purpose, we use the following result of Ghaffari, Harris, and Kuhn [18].

► **Theorem 36** (Theorem 1.1 of [18]). *Any  $r$ -round randomized LOCAL algorithm for a locally checkable problem can be transformed into a deterministic LOCAL algorithm with complexity  $O(r(\log^2 n + \text{ND}))$ , where  $\text{ND}$  is the time required to compute an  $(O(\log n), O(\log n))$ -network decomposition.*

We note that Theorem 1.1 of [18] applies to randomized algorithms that satisfy the following requirements.

1. They always terminate within  $r$  rounds.
  2. Each node sets a flag to 1 if its output is incorrect, and to 0 otherwise. With high probability, the flag should be 0.
  3. Each node should be able to check, in  $O(1)$  rounds, whether its flag has the correct value.
- In order to apply Theorem 36 and derandomize the algorithm of Theorem 17, we need to show that we can tweak the algorithm to satisfy the above requirements. The algorithm presented in Theorem 17 clearly satisfies the first requirement. However, it is not compatible with the second one, since the guarantee on the quality of the coloring does not always hold. More precisely, the number of colors obtained by a node only holds with high probability, and for this reason, if  $q'$  is not known, a node may not notice that its output is incorrect, and it fails to set its flag correctly. Let us see how to handle this issue.

If  $q'$  is known to the nodes, then they can just set their flag to 1 if they have strictly less than  $(1 - \varepsilon)q't$  colors, and 0 otherwise. If  $q'$  is not known, we perform a preprocessing step to compute a value of  $q'$  (that may be different for different nodes), that will then be used by the nodes to decide whether to set their flag or not. The preprocessing step works as follows. Let  $T$  be the running time of the algorithm that we want to derandomize. Each node  $v$  spends  $2T$  rounds to gather its  $2T$ -radius neighborhood. Then, it checks, in that neighborhood, what is the maximum value  $q'$  for which there exists a  $(p't : (1 - \varepsilon)q't)$ -coloring. Observe that the obtained value  $q'$  is a lower bound on the number of colors that  $v$ , while executing the algorithm of Theorem 17, is able to obtain when it belongs to a cluster. This follows from the fact that, any cluster where  $v$  belongs to must be fully contained in its  $2T$ -radius neighborhood. Note that the preprocessing step also guarantees that the third requirement of Theorem 1.1 of [18] is satisfied, since each node can just check whether the flag is set correctly depending on the number of colors that it has and the value of  $q'$ .

Hence, by combining Theorem 36 with the obtained variant of the algorithm of Theorem 17, we obtain the following theorem.

► **Theorem 19.** *Let  $G = (V, E)$  be a graph that admits a  $(p : q)$  coloring, and let  $t = O(\log n / \varepsilon)$ , for an arbitrary  $\varepsilon > 0$ . There is a deterministic LOCAL algorithm that computes a  $(tp' : (1 - \varepsilon)tq')$ -coloring, that is, a fractional  $(1 + O(\varepsilon)) \frac{p}{q}$ -coloring, in  $O(\log n (\log^2 n + \text{ND}) / \varepsilon)$  rounds, where  $\text{ND} \leq \text{poly log } n$  is the time required to compute an  $(O(\log n), O(\log n))$ -network decomposition.*

## B Grids

In [11], it has been shown that, for any constant  $\varepsilon$  and  $d$ , in  $d$ -dimensional grids, it is possible to compute a fractional  $(2 + \varepsilon)$ -coloring in time  $O(\log^* n)$ . We show that the same problem can be solved in constant time.

The algorithm of [11] computes a  $(2q + 4 \cdot 6^d : q)$ -coloring that runs in  $O(d\ell(2\ell)^d + d\ell \log^* n)$  rounds, where  $\ell = q + 2 \cdot 6^d$ . The running time is dominated by the time required to compute a maximal independent set on  $G^\ell$ , where the distance is taken w.r.t. the infinity norm (that is, for two nodes  $u$  and  $v$  with coordinates  $(u_1, \dots, u_d)$  and  $(v_1, \dots, v_d)$  their

distance is  $\max_{1 \leq i \leq d} \{|u_i - v_i|\}$ . In fact, a maximal independent set can be computed in time  $O(\Delta + \log^* n)$ , and on  $G^\ell$  we have that  $\Delta = (2\ell + 1)^d$ , and hence there is an overhead of  $O(d\ell)$  in the runtime. After computing the independent set, the rest of the algorithm requires just  $O(d\ell(2\ell)^d)$  rounds. By setting  $q = 2^{O(d + \log \frac{1}{\varepsilon})}$ , this algorithm gives a fractional  $(2 + \varepsilon)$ -coloring in time  $T_{\text{coloring},n} + T_{\text{rest}}$ , where  $T_{\text{coloring},c} = 2^{O(d + \log \frac{1}{\varepsilon})} \log^* c$  and  $T_{\text{rest}} = 2^{O(d^2 + d \log \frac{1}{\varepsilon})}$ . Similarly as in the proof of Lemma 29, we can replace the  $\log^* n$  dependency with  $\log^* c$ , if nodes are provided with a distance- $d\ell$   $c$ -coloring.

We compute a partial distance- $d\ell$  coloring by letting nodes pick a color uniformly at random. Nodes that obtain an invalid coloring, uncolor themselves. We would like to execute the algorithm of [11] on the subgraph induced by colored nodes, but we cannot, since the subgraph is not a grid anymore. In order to solve this issue, we consider only nodes satisfying that, within their running time, they cannot notice that the graph is not a grid. We call these nodes *happy*. In other words, a node is happy if and only if, within the running time of the algorithm, it does not see any uncolored node. On these nodes, by a standard indistinguishability argument, the algorithm must work correctly. Note that this running time depends on  $c$ , but we will pick a value of  $c$  satisfying that the total running time is anyways strictly less than  $kT_{\text{rest}}$ , for some large enough constant  $k$ . The probability that a node is colored is at least  $1 - \Delta/c$ . Since a node sees at most  $d^{kT_{\text{rest}}}$  nodes within its running time, the probability that a node is happy is at least  $1 - \frac{\Delta}{c} d^{kT_{\text{rest}}}$ , meaning that a node is unhappy with probability at most  $\frac{\Delta}{c} d^{kT_{\text{rest}}}$ . We want this probability to be at most  $\varepsilon$ , and for that, we can pick  $c = \Delta d^{kT_{\text{rest}}} / \varepsilon$ . Note that, for such a value of  $c$ ,  $T_{\text{coloring},c} + T_{\text{rest}} \leq kT_{\text{rest}}$ , as required.

Hence, there is an algorithm that in  $2^{O(d^2 + d \log \frac{1}{\varepsilon})}$  rounds computes a partial fractional  $(2 + \varepsilon)$ -coloring satisfying that each node is uncolored with probability at most  $\varepsilon$ . By applying Lemma 22 and Lemma 24, we obtain the following.

► **Theorem 21.** *Let  $G$  be a  $d$ -dimensional grid. For any  $\varepsilon > 0$ , there is a deterministic LOCAL algorithm that computes a fractional  $(2 + \varepsilon)$ -coloring on  $G$ , that runs in  $2^{O(d^2 + d \log \frac{1}{\varepsilon})}$  rounds.*



# Distributed Recoloring of Interval and Chordal Graphs

Nicolas Bousquet  

Univ. Lyon, Université Lyon 1, LIRIS UMR CNRS 5205, F-69621, Lyon, France

Laurent Feuilloley  

Univ. Lyon, Université Lyon 1, LIRIS UMR CNRS 5205, F-69621, Lyon, France

Marc Heinrich  

University of Leeds, UK

Mikaël Rabie  

Université de Paris, CNRS, IRIF, F-75013, Paris, France

---

## Abstract

---

One of the fundamental and most-studied algorithmic problems in distributed computing on networks is graph coloring, both in bounded-degree and in general graphs. Recently, the study of this problem has been extended in two directions. First, the problem of recoloring, that is computing an efficient transformation between two given colorings (instead of computing a new coloring), has been considered, both to model radio network updates, and as a useful subroutine for coloring. Second, as it appears that general graphs and bounded-degree graphs do not model real networks very well (with, respectively, pathological worst-case topologies and too strong assumptions), coloring has been studied in more specific graph classes. In this paper, we study the intersection of these two directions: distributed recoloring in two relevant graph classes, interval and chordal graphs.

More formally, the question of recoloring a graph is as follows: we are given a network, an input coloring  $\alpha$  and a target coloring  $\beta$ , and we want to find a schedule of colorings to reach  $\beta$  starting from  $\alpha$ . In a distributed setting, the schedule needs to be found within the LOCAL model, where nodes communicate with their direct neighbors synchronously. The question we want to answer is: how many rounds of communication are needed to produce a schedule, and what is the length of this schedule?

In the case of interval and chordal graphs, we prove that, if we have less than  $2\omega$  colors,  $\omega$  being the size of the largest clique, extra colors will be needed in the intermediate colorings. For interval graphs, we produce a schedule after  $O(\text{poly}(\Delta) \log^* n)$  rounds of communication, and for chordal graphs, we need  $O(\omega^2 \Delta^2 \log n)$  rounds to get one.

Our techniques also improve classic coloring algorithms. Namely, we get  $\omega + 1$ -colorings of interval graphs in  $O(\omega \log^* n)$  rounds and of chordal graphs in  $O(\omega \log n)$  rounds, which improves on previous known algorithms that use  $\omega + 2$  colors for the same running times.

**2012 ACM Subject Classification** Theory of computation  $\rightarrow$  Distributed algorithms

**Keywords and phrases** Distributed coloring, distributed recoloring, interval graphs, chordal graphs, intersection graphs

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2021.19

**Related Version** *Full Version:* <https://arxiv.org/abs/2109.06021> [11]

**Funding** This work was supported by ANR project GrR (ANR-18-CE40-0032).

**Acknowledgements** We thank the reviewers for their useful comments, and Marthe Bonamy for starting this project with us. The second author thanks Fabian Kuhn and Václav Rozhoň for their kind and expert answers to his questions on network decomposition.



© Nicolas Bousquet, Laurent Feuilloley, Marc Heinrich, and Mikaël Rabie; licensed under Creative Commons License CC-BY 4.0

25th International Conference on Principles of Distributed Systems (OPODIS 2021).

Editors: Quentin Bramas, Vincent Gramoli, and Alessia Milani; Article No. 19; pp. 19:1–19:17

Leibniz International Proceedings in Informatics



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

## 1 Introduction

Finding a proper coloring of the network is one of the most studied problems in the LOCAL model of distributed computing. It is a typical symmetry-breaking problem, and it can be used as a building block to solve many other problems [4]. It also has direct applications, and a popular one since [31] is the assignment of frequencies (or time slots) in wireless networks. Consider a network of stations using radio communication. A simple model consists in considering that either two stations are close enough to communicate, but then they should use different emission frequencies to avoid collisions, or they are far apart, cannot communicate, and can safely use the same frequency. An assignment of frequencies that avoids conflict is equivalent to a proper coloring. (Note that this is the simplest model, one could also consider that communication and collision happen at different distances.)

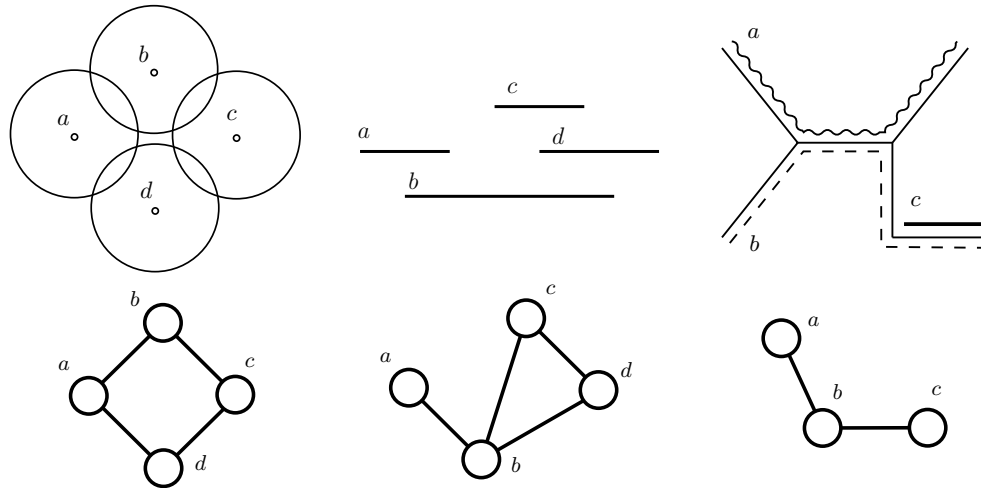
Simply computing a coloring is not enough in some contexts. Indeed, suppose that we have an algorithm for the classic coloring task: starting from a coloring with a high number of colors (for example, unique identifiers), we get a coloring with fewer colors. Now, how do we update the frequencies of the network? If we do it simultaneously without stopping all communications, there might be conflicts during the transition period, between the nodes still using the old frequencies and the ones using the new frequencies. Thus, we need to stop all communications during the transition period, which is inconvenient if the network is performing other tasks in parallel. Therefore, instead of just a new coloring, we would like to have a series of colorings, such that at any step the color change is safe, that is old and new colors do not conflict. In this paper, we aim at finding such schedules in a distributed way. More precisely, we tackle the following more general recoloring question: how to go from an input coloring to a target coloring, such that at each step, the coloring is proper and the vertices whose colors are changing form an independent set of the network. To do so, it will sometimes be necessary to assume that we are allowed to use a few additional colors, that are not present in the input and target colorings.

For this problem, three questions naturally arise: Do we need extra colors to be able to produce a schedule, and if yes, how many? Can we find a new schedule locally? And how long the recoloring schedule needs to be? We study the problem from the point of view of the LOCAL model, thus we want our algorithms to use a sublinear number of communication rounds. For this setting, it is known that in general graphs we sometimes need to use many additional colors (*e.g.*  $k - 1$  extra colors to go from a  $k$ -coloring to another). This is bad news, as using extra colors can be considered as costly (*e.g.* because we need to reserve frequencies to make the transitions). Fortunately, radio networks have topologies that are more constrained than general graphs, and we want to exploit these properties to design schedules that use less extra colors.

In a first approximation, radio networks can be modeled as intersection graphs of (unit) disks in the plane (see Figure 1). In a unit disk graphs, we associate to each node a location on the plane, and two nodes are in conflict (*i.e.*, linked by an edge) if their (unit) disks intersect. Unit disks can always be colored with  $3\omega$  colors [30], where  $\omega$  is the size of the maximum clique of the graph. In contrast, there exists triangle-free graphs with arbitrarily large chromatic number. However, deciding if there exists a  $k$ -coloring remains NP-complete in the centralized setting, even in unit disks graphs. In general, we do not have a good understanding of the coloring of unit-disk graphs, and we know very little about recoloring, even in the centralized setting.

One can then wonder what happens in other simpler geometric graph classes. For instance, if we assume that all the centers are on the same line in the plane, the obtained graphs are called *interval graphs*. An interval graph is an intersection graph of intervals of the real line

(see Figure 1). In other words, an interval of the real line is associated to every node of the graph and there is an edge between two nodes if their corresponding intervals intersect. Interval graphs can be colored with  $\omega$  colors in polynomial time in the centralized setting.



■ **Figure 1** Illustrations of the different intersection graph classes mentioned. On each column, the geometric intersection model is on top, and the corresponding graph on the bottom. The left-most column is for unit-disk graph, the middle one for interval graphs, and the right-most for chordal graphs. For the chordal intersection model, the base tree uses plain edges,  $a$  is the wavy subtree,  $b$  is the dashed one, and  $c$  the bold one. (This last intersection model is simplistic, to allow a readable drawing. In general the subtrees are not paths, and for this specific graph, a simpler geometric representation exists.)

A natural generalization of interval graphs are chordal graphs. Chordal graphs are one of the most-well studied classes in graph theory, with deep structures, fast algorithms, and many important applications [21, 6, 33]. They are intersection graphs of subtrees of a tree. In other words, given a tree  $T$ , every vertex of a chordal graph  $G$  is associated to a subtree of  $T$  and two nodes of  $G$  are adjacent if their corresponding subtrees intersect (see Figure 1).

Our main results are recoloring algorithms for interval and chordal graphs. Our techniques also yield coloring algorithms that improve on the state of the art.

## 1.1 Our results

Our main results are recoloring algorithms for interval and chordal graphs. Let us describe a bit more formally what a distributed recoloring algorithm is. We use the definition introduced in [9]. A *valid recoloring schedule* from a coloring  $\alpha$  to a coloring  $\beta$  consists in a sequence of colorings that starts with  $\alpha$  and ends in  $\beta$  such that, at every step, the coloring is proper and the vertices whose colors are modified at a given step (called the *recoloring vertices*) form an independent set of the graph. A distributed recoloring algorithm is an algorithm such that every node computes its own schedule. In this paper, we compute the schedule in the LOCAL model. Each node can check the validity of the schedule by comparing its own with its neighbors: they check that at each step, the coloring is locally proper, and that if it changes its own color during a step, none of its neighbors does the same.

In this paper, we will study how many *rounds* are used in the LOCAL model to produce a schedule of some *length*. We first focus on interval graphs and then extend our result to chordal graphs. We first prove the following:

► **Theorem 1.** *Let  $G$  be an interval graph and  $\alpha, \beta$  be two proper  $k$ -colorings of  $G$ . It is possible to find a schedule to transform  $\alpha$  into  $\beta$  in the LOCAL model in  $O(\text{poly}(\Delta) \log^* n)$  rounds using at most:*

- $c$  additional colors, with  $c = \omega - k + 4$ , if  $k \leq \omega + 2$ , with a schedule of length  $\text{poly}(\Delta)$ ,
- 1 additional color if  $k \geq \omega + 3$ , with a schedule of length  $\text{poly}(\Delta)$ ,
- no additional color if  $k \geq 2\omega$  with a schedule of exponential-in- $\Delta$  length.
- no additional color if  $k \geq 4\omega$  with a schedule of length  $O(\omega\Delta)$ .

We also prove in Lemma 14 that the complexity of second item of Theorem 1 is optimal, in the sense that with less than  $2\omega$  colors, the number of rounds (as well as a schedule) must be linear in the size of the graph. Basically, when the number of colors is smaller than  $2\omega$ , if we do not have additional colors, we might need to recolor vertices from the border of a graph in order to be able to recolor vertices in the middle. On the contrary, when the number is at least  $2\omega$ , we can save a color using only local modifications, and then proceed as if we had one additional unused color. Unfortunately, in order to free this additional color, we use a schedule of size exponential in  $\Delta$ .<sup>1</sup> On the positive side, this color saving step (and actually the whole recoloring) can be performed with a short schedule when  $k \geq 4\omega$ .

For the first item, as argued above, at least one additional color is needed. The colors that we use in addition to that one come from a result of [10] that we use (almost) as a black-box (see the next section). If the number of colors could be decreased in the context of [10], then it could also be decreased in our work. We left as an open problem the question of deciding if the numbers of additional colors can be reduced to 1.

One can naturally wonder what are the dependencies in  $\Delta$  and  $\omega$  on our complexity and schedule lengths, as we did not give them explicitly. We are using as a black-box the recoloring result on chordal graphs of [10], which gives a centralized recoloring algorithm in  $O(\omega^4 \cdot \Delta \cdot n)$  steps, where only one vertex can be recolored at each step. The schedule we produce contains as subsequence a constant number of such black-box schedules, corresponding to subgraphs containing about  $O(\omega^4 \cdot \Delta^2)$  vertices. In order to keep the proofs as simple as possible, we did not try to optimize this polynomial function. In particular, the recoloring procedure of [10] can probably be adapted in the LOCAL model with a parallel schedule shorter than  $O(\omega^4 \cdot \Delta \cdot n)$ .

Since for interval graphs, the gap between  $\omega$  and  $\Delta$  can be arbitrarily large, it would be interesting to determine if the recoloring schedule (as well as the number of rounds) can be reduced to a function of  $\omega$  only.

Our second main result consists in extending this recoloring result to chordal graphs. Namely, we prove:

► **Theorem 2.** *Let  $G$  be a chordal graph and  $\alpha, \beta$  be two proper  $k$ -colorings of  $G$ . It is possible to find a schedule of length  $n^{O(\log \Delta)}$  to transform  $\alpha$  into  $\beta$  in  $O(\omega^2 \Delta^2 \log n)$  rounds in the LOCAL model using at most:*

- $c$  additional colors, with  $c = \omega - k + 4$ , if  $k \leq \omega + 2$ ,
- 1 additional color if  $k \geq \omega + 3$ .

Note that while the recoloring schedule of Theorem 1 is polynomial, the one obtained in Theorem 2 is superpolynomial in  $\Delta$  (but the number of rounds remains polynomial). We left as an open problem the existence of polynomial schedule. In terms of number of rounds,

---

<sup>1</sup> It is likely that the  $O(n^{d+1})$ -recoloring algorithm in the centralized setting for  $d$ -degenerate graphs can be adapted in order to provide a polynomial schedule instead, but we did not do it to keep the proof as short as possible.



we also pay a logarithmic factor in comparison to the  $O(\log^* n)$  rounds used for interval graphs. It would be interesting to know if there exists a  $(\omega + x)$ -coloring algorithm for chordal graphs for a constant  $x$ , that only needs  $O(\log^* n)$  rounds. We will explain in the subsection that follows, that these differences in schedule length and running time are inherent to our approach, and going beyond those would require new techniques.

While designing tools for recoloring, we also produce improved algorithms for the coloring problem. In a nutshell, it was known how to get  $(\omega + 2)$ -colorings for interval and chordal graphs, and we improve this to  $(\omega + 1)$  colors. More precisely, it was proved in [23] that an  $(\omega + 2)$ -coloring of interval graphs can be obtained in the LOCAL model in  $O(\omega \log^* n)$  rounds.<sup>2</sup> Our first result consists in improving the result of Halldorsson and Konrad [23] by proving:

► **Theorem 3.** *Interval graphs can be colored with  $(\omega + 1)$ -colors in  $O(\omega \log^* n)$  rounds in the LOCAL model.*

Note that  $(\omega + 1)$  is the best we can hope for in sublinear time in paths (which are interval graphs with  $\omega = 2$ ), since paths cannot be colored with 2 colors in less than  $\Omega(n)$  rounds [26, 4]. This can actually be generalized for any fixed  $\omega$ , by considering the  $(\omega - 1)$ -th power of a path. The  $k$ -th power of a path is a graph with vertex set  $v_1, \dots, v_n$  where two vertices  $v_i, v_j$  are adjacent if and only if  $|i - j| \leq k$ . Hence, each maximal clique corresponds to  $\omega$  consecutive nodes. An  $\omega$ -coloring of such a graph would require the algorithm to put a same color every  $\omega$  nodes in the path, which again requires  $\Omega(n)$  communication rounds.

For chordal graphs, Konrad and Zamaraev [24] proved that an  $(\omega + 2)$ -coloring can be found in  $O(\omega \log n)$  steps. Again, we can reduce the number of colors to  $\omega + 1$ .

► **Theorem 4.** *Chordal graphs can be colored with  $(\omega + 1)$ -colors in  $O(\omega \log n)$  rounds in the LOCAL model.*

Again, the dependency in  $n$  is optimal for  $\omega = 2$  because (unoriented) trees cannot be 3-colored in  $o(\log n)$  rounds [26, 4]. We leave as an open question if  $\Omega(\log n)$  communication rounds are required to produce an  $(\omega + 1)$ -coloring for  $\omega > 2$ .

## 1.2 Our techniques

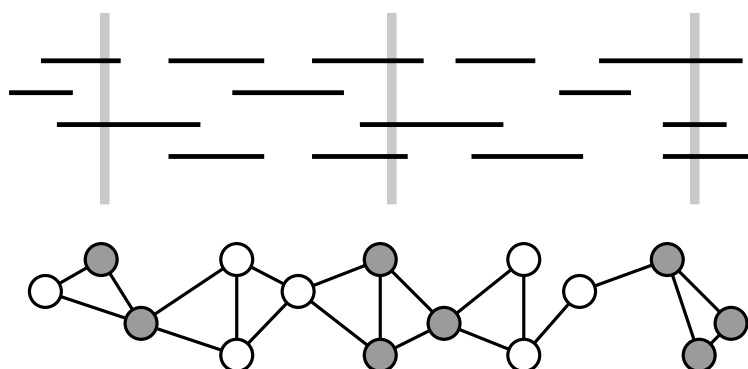
Our algorithms are using graph decompositions. Basically, we first split the graph into components of controlled diameter, and then work on the different components, taking care of not creating conflicts at the borders.

To get more into the details, let us start with coloring. For interval graphs, we use a variation of a decomposition from [23]. It consists in proving that we can find some subsets of vertices that cut the interval graphs into parts whose diameter is large enough, but not too large, in  $O(\log^* n)$  rounds (with a multiplicative factor depending on the diameter of each part). See Figure 2. Our contribution is in the second part of the algorithm. We show that we can start from an arbitrary coloring of the cuts, and extend this partial coloring into a tight  $(\omega + 1)$ -coloring of  $G$ , whereas previous algorithms needed some extra slack in the number of colors.

For chordal graph, [24] provides a recursive decomposition into interval graphs, that we adapt to our setting. The idea is to use a rake-and-compress strategy (in the spirit of [28]) to decompose the graph in  $O(\log n)$  layers, each layer being a union of interval graphs. A

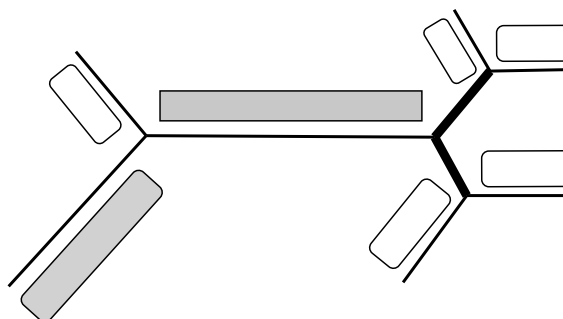
<sup>2</sup> Actually, in [23] the result is stated as a  $(1 + \epsilon)$ -approximation of the optimal  $\omega$ -coloring, with the condition that  $\epsilon \geq 2/\omega$ , which is equivalent to an  $(\omega + 2)$ -coloring.

19:6 Distributed Recoloring of Interval and Chordal Graphs



■ **Figure 2** Illustration of the interval decomposition of [23]. On top the interval representation, and on the bottom, the graph. The decomposition consists in choosing cliques (the gray nodes) whose removal decomposes the graphs into subgraphs of controlled diameter (the white nodes). Note that the graph can then be seen as a path alternating between gray cliques and white subgraphs. What we do is slightly different, as our cuts are not cliques, but the intuition is the same.

typical phase of rake-and-compress on a tree consists in removing the leaves (the rake part), and removing or contracting the long paths (the compress part). After  $O(\log n)$  such phases, the tree is empty. In our algorithm, the tree structure is the underlying tree of the chordal graph, like in Figure 1.<sup>3</sup> We consider here as leaves the pending interval subgraphs. These are subgraphs made by taking the intervals that are on the branches of the tree leading to a leaf (see Fig. 3). On the other hand, long paths correspond to long separating interval subgraphs, whose removal disconnects the graph, and that have large enough diameter.



■ **Figure 3** Illustration of one step of the decomposition of chordal graphs into interval graphs (of controlled diameter). The tree of plain edges is the underlying tree. The rectangles with rounded corners correspond to the pending interval graphs. The rectangle with spiky corners correspond to a separating interval graph. The gray rectangles are the ones that are too long and will be further decomposed. The bold edges correspond to part of the graphs that are not pending, nor long and separating. These part are the ones that are kept for the next phases, all the other vertices are removed.

Let  $V_i$  be the set of nodes of the interval graph removed at phase  $i$ . The coloring algorithm then consists in coloring recursively  $V_k, \dots, V_1$ , in this order. That is, we start by coloring the vertices that have been removed last. Assume that we have a coloring of  $V_{i+1}, \dots, V_k$ .

<sup>3</sup> Actually, there are two such trees, the one that correspond to the geometric representation, that we use in this introduction, and the clique tree, which has a more graph-theoretic definition, and that we use in the proofs. The two are essentially equivalent.

The key point is that when we consider a new interval subgraph, by construction, only its borders can be already colored, and therefore we are back to the situation we had for interval graphs. We then get the same number of colors. The logarithmic-in- $n$  complexity comes from the  $O(\log n)$  layers of the rake-and-compress, that we must process sequentially.

Now, let us describe our recoloring techniques. For interval graphs, Theorem 1 is obtained using two tools coming from the centralized setting. First, we use Kempe chains that have been used in several graph coloring proofs in the last decades. Given a graph  $G$  and a coloring  $c$  of  $G$ , an  $(a, b)$ -component is a connected component of  $G$  restricted to the vertices colored  $a$  and  $b$ . A *Kempe component* is an  $(a, b)$ -component for some pair of colors  $a, b$ . One can remark that, given a proper coloring and an  $(a, b)$ -component, permuting the colors  $a$  and  $b$  still leaves a proper coloring. In recoloring, this permutation can be performed by using one extra color as buffer. In a distributed setting, using directly those transformations can be perilous, as components can be as large as the diameter of the graph. We adapt Kempe components for distributed algorithms by using an additional color to cut these Kempe components and then permute locally the colors on these components<sup>4</sup>. This will allow us to color independently some parts of the graph with the target coloring.

Our second main tool is a recoloring technique introduced in [10] for recoloring interval graphs in the centralized setting. We show how to adapt their technique to the distributed setting. The result of [10] essentially ensures that if a large enough (that is, of size  $\text{poly}(\Delta)$ ) number of consecutive vertices  $X$  are colored in a desirable way, then we can recolor all the vertices around  $X$  with the target coloring by simply recoloring vertices locally. The idea of the proof consists in sliding little by little the set of vertices  $X$  colored with the desirable coloring from left to right in such a way that when a vertex leaves the set, it has its target color.

For chordal graphs (Theorem 2), we use Theorem 1 as a black-box, as well as an adaptation of the distributed partition of chordal graphs into interval pieces from [24]. We then prove that if we have a recoloring schedule of  $G[\cup_{j \geq i+1} V_j]$  then we can adapt it into a recoloring schedule for  $G[\cup_{j \geq i} V_j]$  using Theorem 1 and classical recoloring tools.

Notice that the schedule for chordal graphs is large in comparison with the ones for interval graphs (order of  $n^{O(\Delta)}$  versus  $\text{poly}(\Delta \cdot \log^* n)$ ). This comes from the fact that in the extension of the recoloring schedule of  $G[\cup_{j \geq i+1} V_j]$  to  $G[\cup_{j \geq i} V_j]$ , we need to leave a polynomial amount of recoloring steps to recolor vertices of  $V_i$  between consecutive recolorings of vertices of  $G[\cup_{j \geq i+1} V_j]$ .

The schedule having a  $\log n$  factor instead of a  $\log^* n$  factor originates once again from the decomposition of [24] that uses a logarithmic number of layers.

### 1.3 Organization of the paper

We start with the related work in Section 2, and some preliminaries in Section 3. Then in Section 4, we explain how to decompose interval graphs and how to modify colorings in such graphs. As a corollary, we get our coloring result for interval graphs. Section 5 is devoted to the recoloring of interval graphs, based on the decomposition. Finally, Section 6 tackles chordal graphs, describing our decomposition, coloring and recoloring results. Many proofs, as well as full subsections of Section 5, are deferred to the full version [11].

---

<sup>4</sup> Kempe chains have already been used in the distributed setting, see e.g. [29].

## 2 Related work

### Distributed coloring

Distributed coloring in bounded-degree and general graphs has been extensively studied. We refer to the monograph [4] for a general overview of the domain. In the LOCAL model, the classic coloring problem is  $(\Delta + 1)$ -coloring, where  $\Delta$  is the maximum degree of the graph. In this paper, we are interested in coloring and recoloring with a nearly optimal number of colors (which can be much lower than  $\Delta$ ). There are ways to cope with such small color palette while preserving some locality: studying (multiplicative) approximation of the optimal coloring [2, 5] and/or restricting to special graph classes, for example planar graphs [20, 1, 15] or bounded arboricity graphs [3, 19].

The line of work that is the closest to ours is about coloring interval and chordal graphs. It started with [22] who proved an 8-approximation of the optimal coloring, in time  $O(\log^* n)$  in interval graphs. This result was then improved to  $(1 + \epsilon)$ -approximation in time  $O(\frac{1}{\epsilon} \log^* n)$  in [23]. The  $\epsilon$  of this theorem can be as small as  $2/\omega$ , which means that [23] obtains an  $(\omega + 2)$ -coloring in time  $O(\omega \log^* n)$ . Coloring in chordal graphs was considered in [24]. The authors prove a  $(1 + \epsilon)$ -approximation in time  $O(\frac{1}{\epsilon} \log n)$ . Again, the bounds on  $\epsilon$  allow to derive an  $(\omega + 2)$ -coloring, and here the running time is  $O(\omega \log n)$  rounds.

### Distributed Reconfiguration

The introduction of recoloring, *i.e.* reconfiguration of colorings, in the distributed setting is due to [9]. They, for instance, provide algorithms to recolor trees and subcubic graphs with one extra color, and an impossibility result for 3-colored toroidal grids with only one extra color. In particular, they find a constant size schedule with  $O(\log n)$  communication rounds on trees if an extra color is allowed. Trees are chordal graphs with  $\omega = 2$ , hence our results relate directly to that, and we use a similar *Rake-and-Compress* approach. Then [13] considered the distributed reconfiguration of maximal independent sets (MIS). Before these papers, some results of [29] can be seen as recoloring, as they describe a way to find a  $\Delta$ -coloring from a given  $(\Delta + 1)$ -coloring by using “augmenting paths” that actually are Kempe changes.

### Centralized graph recoloring

In the centralized setting, graph recoloring received a considerable attention in the last decades. In this overview, we will focus on single-vertex reconfiguration, that is, the model where exactly one vertex is recolored at each step. In that model, it is known that every  $k$ -coloring can be transformed into any other as long as  $k$  is at least the degeneracy  $d$  of  $G$  plus two [17, 14]. However, the number of recoloring steps is *a priori* exponential in  $n$ . Cereceda conjectured in [14] that, when  $k \geq d + 2$ , there exists a quadratic transformation between any pair of  $k$ -colorings of any  $d$ -degenerate graph. Bousquet and Heinrich [12] proved that there always exist  $O(n^{d+1})$  transformation between any pair of  $k$ -colorings when  $k \geq d + 2$ .

Since chordal graphs are perfect graphs, it is well known that the degeneracy is related to the cliques number by:  $d = \omega - 1$ . For interval and chordal graphs, Bonamy et al. [8] proved that there exists a quadratic transformation between any pair of vertices when  $k \geq \omega + 1$  and they provide a quadratic lower bound when  $k = \omega + 1$ . This existence of a quadratic transformation has been extended to bounded treewidth graphs [7]. Recently Bartier and Bousquet proved in [10] that, when  $k \geq \omega + 3$ , there always exists a linear transformation

between any two colorings of a chordal graph. Moreover, in contrast to most of the centralized recoloring algorithms, this algorithm is “local” and will be used as one of the blocks of our proof.

### Graph decompositions

The first phase of our algorithms consists in decomposing the graph into components of small diameter with some additional properties. These decompositions are close to what is known as network decompositions, which are partitions of the nodes of the graph into few classes, such that every class has all its connected components of small diameter (see [18] for an overview of this topic, and in particular the recent advances). For geometric graphs, such as interval, unit-disks graphs, it is known that a network decomposition with a constant number of classes and constant diameter can be obtained in  $O(\log^* n)$  rounds [25, 32], which implies that all the classic problems such as  $(\Delta + 1)$ -coloring or maximal matching can be solved in  $O(\log^* n)$  rounds. For trees, [9] establishes a network decomposition with constant size components in time  $O(\log n)$ , which is enough for recoloring. For our purpose, which is to (re)color interval and chordal graphs with few colors, standard network decompositions are not powerful enough, and we need to build more constrained decompositions. Our decompositions are inspired by the ones of [23] and [24].

## 3 Basic properties of interval and chordal graphs

*Interval graphs* are intersection graphs of intervals of the real line. *Proper interval graphs* are the intersection graphs of a set of intervals of size one. Equivalently, they correspond to interval graphs where no interval is included in the other. It is easy to check that, starting from an interval graph, and keeping only the intervals that are maximal for inclusion, we get a proper interval graph.

An interval graph can also be characterized as an intersection graph of subpaths of a path, and we can extend this definition to define chordal graphs. A graph is *chordal* if it is an intersection graph of subtrees of a tree. Equivalently, it is the class of graphs for which all the cycles of length at least 4 has a chord.

Before we list some properties of interval, proper intervals and chordal graphs, let us define the notion of degeneracy.

► **Definition 5.** *A graph  $G$  is  $d$ -degenerate if there exists an ordering  $v_1, \dots, v_n$  of  $V$  such that for every  $i \leq n - 1$ , the number of neighbors of  $v_i$  in  $\{v_{i+1}, \dots, v_n\}$  is at most  $d$ .*

Let us remind a few facts about interval graphs.

► **Observation 6.** *The following holds.*

1. *Any chordal graph can be colored with  $\omega$  colors and is  $(\omega - 1)$ -degenerate [27].*
2. *The max degree  $\Delta$  can be as large as  $n$ , even if  $\omega$  is small (consider  $n - 1$  disjoint small intervals contained in a big one).*
3. *In an interval graph, the set of intervals that are minimal for inclusion corresponds to a proper interval. The same holds for “maximal for inclusion” [23].*
4. *In any proper interval graph,  $\Delta \leq 2\omega - 2$  (since every neighboring interval should cross one of the extremities of the vertex interval.)*

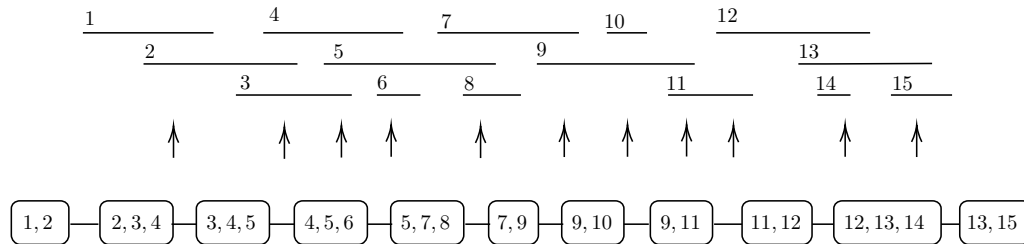
**Clique trees, clique paths and borders**

An essential tool to study chordal graphs is the notion of clique tree.

► **Definition 7** (See e.g. [6]). *Let  $G$  be a chordal graph. A clique tree of  $G$  is a tree  $T$  together with a function that associates to every vertex a connected subtree of  $T$ . Two vertices  $x$  and  $y$  are connected in  $G$  if and only if the subtrees of  $x$  and  $y$  intersect. For each node  $u \in T$ , the subset of nodes in  $G$  whose subtree contains  $u$  is called the bag of  $u$ . Note that each bag of a node of  $T$  forms a clique in  $G$ , so each bag contains at most  $\omega$  vertices (and there always exists a clique tree on at most  $n$  nodes).*

*For interval graphs, the tree  $T$  is a path, and it is called the clique path of  $G$  (see Figure 4 for an illustration).*

We introduce two more notions. Let  $G$  be an interval graph. A set of vertices  $X$  is said to be *consecutive* if it consists in the union of the vertices contained in consecutive cliques of the clique path of  $G$ . The *border* of  $X$  is the subset of  $X$  connected to at least one vertex which is not in  $X$ . In other words, it is the set of vertices of the first and last clique of the clique path containing  $X$  that have a neighbor not in  $X$ . One can easily notice that all the vertices of the border of  $X$  can be partitioned into two cliques (the ones that belong to the first and the last clique of the set of bags).



■ **Figure 4** The first picture represents a set of intervals. The arrows correspond to points of the axis where there is a maximal clique. The second picture describes the clique path of the graph, whose nodes are the maximal cliques.

We will need the following remarks about interval graphs:

- **Remark 8.** Let  $G$  be an interval graph given with an interval representation of  $G$ . Let  $x$  be a vertex of  $G$ . Then:
  - $N(x)$  contain all the vertices  $v$  whose intervals intersect the interval of  $x$ .
  - The removal of  $N(x) \cup \{x\}$  separates the vertices with interval at the left of  $x$  with the vertices with interval at the right of  $x$ .<sup>5</sup> A direct consequence is that, for each node  $x$  that is not contained in a bag of a node of one extremity of the clique path, its box separates the graph.
  - For every  $r \geq 2$ , consider the set  $Y = \cup_{i \leq r} N^i(x)$ . The subset of vertices of  $Y$  that are incident to any vertex of  $V \setminus Y$  is composed of at most two cliques  $A, B$ . Moreover, there is a clique path of  $G[Y]$  where  $A$  is the first clique and  $B$  is the last clique.

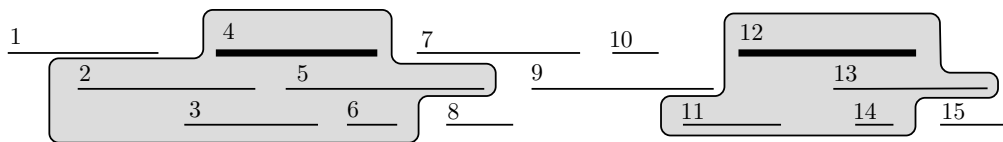
<sup>5</sup> Note that there might be more components since, for instance, the “left graph” might not be connected.

#### 4 Decomposing and coloring interval graphs

In the sequential setting, it is easy to compute an interval representation of the graph, and such a representation is often useful in the design of algorithms. For example, one can compute a maximum independent set in a greedy manner, scanning the interval by the increasing right-most endpoint. In our setting, the nodes do not have access to such a representation, and it is not possible to compute one locally. We will build a weaker local version of the interval representation to facilitate the design of coloring and recoloring algorithms. This idea originates from [23, 22].

We now introduce the notion of *boxes*. Remember that an  $(a, b)$ -ruling-set is a set of nodes  $S$ , such that for any  $u, v \in S$   $u$  and  $v$  are at distance at least  $a$ , and any node is at distance at most  $b$  from a node of  $S$ .

► **Definition 9.** Given a  $(4, 5)$ -ruling-set  $S$ , for every node  $v$  of  $S$ , we define its box as its closed neighborhood in the graph.



■ **Figure 5** An example of a box decomposition. The boxes are the gray areas. The other areas are interboxes. The nodes of the ruling sets are the bold intervals. The intervals 6, 8, 10 and 14, are included into other interval thus they are not considered for the computation of the ruling set.

► **Proposition 10.** For a  $(4, 5)$ -ruling-set  $S$  in an interval graph  $G$ , the following holds:

1. The boxes of the nodes in  $S$  are disjoint, and two nodes from two different boxes cannot be adjacent.
2. The removal of all the boxes leaves a set of connected components, that we call interboxes. We have a path alternating boxes and interboxes (by defining adjacency between boxes and interboxes by the existence of an edge linking the two). We call the virtual path the path on vertex set  $S$  whose adjacency is given by the sequence of boxes defined above.
3. The interboxes have diameter at most 11.

The proof, as well as all the other missing proof of this section, can be found in the full version [11].

Now from a distributed point of view, we say that the nodes compute a box decomposition if they compute a  $(4, 5)$ -ruling-set  $S$ , and for every node, either it is in a box, and then it knows the two adjacent boxes, and the structure of the graph between these boxes, or it is not in a box, and knows between which two boxes it is, and the structure of the graph between these boxes.

Let  $S$  be a  $(4, 5)$ -ruling set of  $G$ . If we replace every box  $a$  by a single vertex and every interbox by an edge, the resulting graph is a path called the *ruling path*. Two vertices of  $S$  are said to be consecutive if they are adjacent in the ruling path and at distance at most  $r$  if they are at distance at most  $r$  in the ruling path.

► **Lemma 11.** A box decomposition can be found in  $O(\log^* n)$  rounds in the LOCAL model.

## 19:12 Distributed Recoloring of Interval and Chordal Graphs

Let  $M$  be a box decomposition. As in the proof of Lemma 11, there is a natural virtual path between these boxes. Let  $B, B'$  be two boxes of a box decomposition. We say that the boxes  $B, B'$  are at distance  $r$ , if they are at distance  $r$  in the virtual path of the box decomposition. The subgraph between  $B$  and  $B'$  is the subgraph of  $G$  composed of the boxes  $B, B'$ , all the boxes  $B$  and  $B'$  (in the virtual path) and all the interboxes between  $B$  and  $B'$ . Note that if the boxes  $B$  and  $B'$  are at distance  $r$ , then the subgraph between  $B$  and  $B'$  can be computed in  $O(r)$  rounds in the LOCAL model.

We will now provide a technical lemma that will be helpful in several places of the paper.

► **Lemma 12.** *Let  $A, B$  be two boxes of two nodes at distance 3 and  $H$  be the subgraph between  $A$  and  $B$ . Let  $\alpha$  be a proper  $k$ -coloring of  $H$ , and let  $x$  and  $y$  be two arbitrary colors. Let  $\beta$  be the  $k$ -coloring of  $H$  made by permuting  $x$  and  $y$  in  $\alpha$ . There exists a  $(k+1)$ -coloring of  $H$ , that corresponds to  $\alpha$  on  $A$  and  $\beta$  on  $B$ .*

► **Lemma 13.** *Consider the subgraph  $H$  induced by the nodes of two boxes  $A$  and  $B$ , separated by at least  $3k$  other boxes, and all the nodes in between. Consider also two arbitrary proper  $k$ -coloring  $\alpha$  and  $\beta$  of  $H$ . Then there exists a  $(k+1)$ -coloring of  $H$ , that corresponds to  $\alpha$  on  $A$  and  $\beta$  on  $B$ .*

Given this tool, we easily get the following theorem.

► **Theorem 3.** *Interval graphs can be colored with  $(\omega+1)$ -colors in  $O(\omega \log^* n)$  rounds in the LOCAL model.*

**Proof.** One first computes the box decomposition in time  $O(\log^* n)$ , then, given the paths of boxes, one can iterate an MIS algorithm for paths (e.g. Cole-Vishkin algorithm [16]) to compute a  $(3\omega, 6\omega)$ -ruling set  $S$  of boxes. This uses  $O(\omega \log^* n)$  rounds. Then the nodes of the boxes of  $S$  computes an  $\omega$ -coloring of their own box. Finally, we use the lemma above to fill the gaps, which also takes  $O(\omega)$  rounds. ◀

## 5 Recoloring interval graphs

The goal of this section is to prove the following theorem.

► **Theorem 1.** *Let  $G$  be an interval graph and  $\alpha, \beta$  be two proper  $k$ -colorings of  $G$ . It is possible to find a schedule to transform  $\alpha$  into  $\beta$  in the LOCAL model in  $O(\text{poly}(\Delta) \log^* n)$  rounds using at most:*

- $c$  additional colors, with  $c = \omega - k + 4$ , if  $k \leq \omega + 2$ , with a schedule of length  $\text{poly}(\Delta)$ ,
- 1 additional color if  $k \geq \omega + 3$ , with a schedule of length  $\text{poly}(\Delta)$ ,
- no additional color if  $k \geq 2\omega$  with a schedule of exponential-in- $\Delta$  length.
- no additional color if  $k \geq 4\omega$  with a schedule of length  $O(\omega\Delta)$ .

The result is tight in terms of number of colors for the two last items, because of the following lemma:

► **Lemma 14.** *In interval graphs of clique number  $\omega$ , if no additional color is allowed, then finding a recoloring from a  $c$ -coloring to a  $c'$ -coloring with  $c, c' < 2\omega$  requires  $\Omega(n/\omega)$  rounds in the LOCAL model and a schedule of length  $\Omega(n/\omega)$ .*

**Proof.** We will consider the  $\omega$ -th power of a path. We can build an interval representation of such a graph, by representing every vertex at position  $i$  by the interval  $[i + 1/4, i + k + 3/4]$ . If we consider a power of a path of clique number  $\omega$ , and color its  $i$ -th vertex with color  $i \bmod 2\omega - 1$ , all the vertices but the first and the last  $\omega$  ones are *frozen* (i.e. we cannot change



their color without changing the color of some vertex in its neighborhood before). Thus, a recoloring can only happen little by little starting from the extremities of the interval graph. Thus, a recoloring schedule has length  $\Omega(n)$  and the nodes in the middle of the power path cannot stop before  $\Omega(n)$  rounds, as their slot in the schedule will depend on the length of the power path. Moreover, a node in the middle needs to know its distance to an extremity, which takes  $\Omega(n)$  rounds in the LOCAL model. ◀

## 5.1 Outline of the proof

We now give an outline of the proof of Theorem 1. The full version [11] contains detailed explanations and complete proofs for each step of this outline. Let  $\alpha, \beta$  be two  $k$ -colorings.

**Step 1.** Compute a  $(4, 5)$ -ruling set  $S$  in  $O(\log^* n)$  rounds.

**Step 2.** In graph recoloring, instead of transforming  $\alpha$  into  $\beta$ , a classical method consists in proving that both  $\alpha$  and  $\beta$  can be recolored into a so-called canonical coloring  $\gamma$  with desirable properties (see e.g. [8, 10]). If transformations  $\mathcal{S}_1, \mathcal{S}_2$  from respectively  $\alpha$  and  $\beta$  to  $\gamma$  exist, then  $\alpha$  can be transformed into  $\beta$  via the transformation  $\mathcal{S}_1 \mathcal{S}_2^{-1}$ .

Theorem 3 ensures that an  $(\omega + 1)$ -coloring of  $G$  can be found in  $O(\omega \log^* n)$  rounds. Let us denote by  $\gamma$  such a coloring. The goal of the proof will consist in proving that we can find a transformation from  $\alpha$  to  $\gamma$ .

**Step 3.** We show that if  $k \geq 2\omega$ , then we can reduce the number of colors without any additional color. We first compute an independent set  $S'$  of  $S$  at constant distance. The main idea of this step consists in proving that all the vertices in the interboxes between two consecutive vertices of  $S'$  but the vertices of the boxes of  $S'$  can be recolored with a color smaller than  $2\omega$  without recoloring the boxes of  $S'$  (which guarantees that we can perform this recoloring simultaneously everywhere in the graph). By repeating this operation twice, we then prove that all the vertices are recolored with a color smaller than  $2\omega$ .

When the number is at least  $4\omega$ , we prove a stronger result, since we directly provide a transformation from  $\alpha$  to  $\beta$  in  $O(\omega\Delta)$  rounds.

**Step 4.** At this point, after applying Step 3 if  $k \geq 2\omega$ , we can assume that we have one additional color. Indeed, in the two first cases of Theorem 1, we are allowed to use at least one extra color, and when  $k \geq 2\omega$ , we have freed at least one color at Step 3.

The goal of Step 4 consists in coloring some boxes with their colors in  $\gamma$ . To do that, we will perform some Kempe changes, cut at some large enough distance, using one additional color. By repeating this process several times, we will prove that a box plus all its neighbors at distance  $\Omega(\text{poly } \Delta)$  can be colored with the target coloring.

Using this technique, we can prove that some portions of diameter  $\Omega(\text{poly}(\Delta))$  of the interval graph which are at distance  $f(\Delta)$  from each other are colored with the target coloring  $\gamma$ .

**Step 5.** By Step 4, we can assume that some portions of diameter  $\Omega(\text{poly}(\Delta))$  of the interval graph which are at distance  $f(\Delta)$  from each other are colored with the target coloring  $\gamma$ . We can now use as a black-box a result of Bartier and Bousquet [10] that ensures that we can recolor all the vertices between a consecutive set of boxes with the target coloring without recoloring the border of these sets as long as a sufficiently large number of consecutive vertices

of the initial coloring are colored “nicely”. Since many consecutive vertices of the initial coloring are colored with the target coloring and such a coloring is “nice”, we can use the result of [10] as a blackbox.

After Step 5, all the vertices have received their final color, which concludes the proof.

## 6 Decomposing, coloring and recoloring chordal graphs

We now investigate how to extend the results from the previous sections to chordal graphs. The algorithms for coloring and recoloring proceed in roughly two steps. The first step computes a partition of the chordal graph, obtained by iteratively removing interval subgraphs with controlled diameter. During the computation of this decomposition, we will assume that the nodes of the graph have access to a local view of a clique tree of the graph  $G$ . These local views can be computed in a distributed fashion with a constant number of rounds using the method from [24] (see Section 6.1 for more details).

In the second step, the coloring algorithm consists in adding back step by step the vertices which have been deleted in the previous phase. Informally speaking, for each connected component, either the vertices that have been removed are attached to a single clique, and then we know that all the vertices attached to that clique form a  $(\omega - 1)$ -degenerate graph, and we can then give them a color between 1 and  $\omega$  greedily. Either those vertices are attached to two cliques at large enough distance, and we can then use the machinery introduced in Section 4 to color them with  $\omega + 1$  colors in total. Since at each step, each set has bounded diameter, one leader can decide of the coloring for all the vertices of that set. For the recoloring algorithm, the idea is almost the same. The algorithm constructs iteratively a recoloring schedule by adding back step by step the vertices which were removed in the previous phase either using degeneracy or the tools of Section 5.

### 6.1 Interval decomposition of chordal graphs

► **Definition 15.** *Let  $G$  be a chordal graph, and  $H$  a subgraph of  $G$ . We say that  $H$  is a pending interval graph if  $H$  is an interval graph, and the border of  $H$  is a subset of the first clique in a clique path of  $H$ . We say that  $H$  is a separator interval graph, if it is an interval graph, and the vertices of the border all belong to either the first or the last clique of a clique path of  $H$ .*

An interval decomposition of width  $D$  and depth  $\ell$  of a chordal graph  $G$  is a partition  $V_1, \dots, V_\ell$  of the vertices of  $G$  such that,  $G[V_i]$  is a disjoint union of interval graphs with diameter at most  $3D$ , and every connected component of  $G[V_i]$  is either:

- a pending interval in  $G[V_1, \dots, V_i]$ ;
- or a separator interval graph with diameter at least  $D$ , in  $G[V_1, \dots, V_i]$ .

The result below is adapted from the proofs of [24].

► **Lemma 16.** *For every  $D \geq 0$ , there exists a distributed algorithm which computes an interval decomposition of width  $D$  and depth  $O(\log n)$  in  $O(D \log n)$  rounds in the LOCAL model.*

The proof can be found in the full version [11].

### 6.2 $(\omega + 1)$ -coloring chordal graphs

In this section, we show how an interval decomposition can be used to compute an  $(\omega + 1)$ -coloring of chordal graphs in the LOCAL model. Namely, we prove the following theorem.

► **Theorem 4.** *Chordal graphs can be colored with  $(\omega + 1)$ -colors in  $O(\omega \log n)$  rounds in the LOCAL model.*

The proof of Theorem 4 follows from the decomposition of Lemma 16 and the lemma below.

► **Lemma 17.** *There is an algorithm that, given a graph  $G$  and an interval decomposition of  $G$  of depth  $\ell$  and width at least  $15(\omega + 1)$ , computes a  $(\omega + 1)$  coloring of  $G$  in  $O(\omega \ell)$  rounds in the LOCAL model.*

The proof can be found in the full version [11].

### 6.3 Recoloring chordal graphs

In this section, we describe how to use an interval decomposition in order to compute a recoloring schedule. We prove the following theorem:

► **Theorem 2.** *Let  $G$  be a chordal graph and  $\alpha, \beta$  be two proper  $k$ -colorings of  $G$ . It is possible to find a schedule of length  $n^{O(\log \Delta)}$  to transform  $\alpha$  into  $\beta$  in  $O(\omega^2 \Delta^2 \log n)$  rounds in the LOCAL model using at most:*

- $c$  additional colors, with  $c = \omega - k + 4$ , if  $k \leq \omega + 2$ ,
- 1 additional color if  $k \geq \omega + 3$ .

Theorem 2 uses our decomposition (Lemma 16) and the result below.

► **Lemma 18.** *Let  $G$  be an interval graph and  $\alpha, \beta$  be two proper  $\omega + 3$ -colorings of  $G$ . We suppose we are given an interval decomposition of width  $D = \Omega(\omega^2 \Delta^2)$  and depth  $\ell$  of  $G$ . It is possible to find a schedule of length at most  $O(\text{poly}(\Delta)^\ell)$  to transform  $\alpha$  into  $\beta$  in  $O(D\ell)$  rounds in the LOCAL model.*

**Proof.** Let  $D$  chosen polynomial in  $\omega$  and  $\Delta$  (its value will be specified later). Note that an interval graph of diameter  $D = \text{poly}(\omega, \Delta)$  has a number of nodes that is  $O(\text{poly} \Delta)$ , because this number is upper bounded by  $D\Delta$  and  $\omega \leq \Delta$ . Also let  $k'$  be the total of number of colors used (including the additional colors). Let us denote by  $V_1, \dots, V_\ell$  the interval decomposition of  $G$ , and  $G_i = G[V_1 \cup \dots \cup V_i]$ . Let  $\alpha$  and  $\beta$  the two colorings given as input, and let  $\alpha_i$  to  $\beta_i$  be the restrictions of  $\alpha$  and  $\beta$  to  $G_i$ . The algorithm proceeds by building successively a recoloring schedule  $\lambda_i$  for  $G_i$  from  $\alpha_i$  to  $\beta_i$ , and progressively extending this recoloring schedule for the rest of the graph. In order to simplify the proof, we will require that the schedule produced by the algorithm has an additional property, namely that all the vertices recolored at a step  $j$  of the schedule, are recolored with the color  $j \bmod k'$ . Note that we can easily produce a schedule satisfying this property, up to multiplying its length by a factor of  $k'$ . This property will be useful later in order to extend progressively the schedule.

Since  $G_1$  is a disjoint union of interval graphs of diameter  $O(D)$ , we can compute a recoloring schedule in  $G_1$  from  $\alpha_1$  to  $\beta_1$  of length linear in the number of nodes by [10], which is in  $O(\text{poly} \Delta)$  in  $O(D)$  rounds.

Assume that we have computed a recoloring schedule in  $G_{i-1}$ . We now describe how to extend this schedule to  $G_i$ . Before each step of  $\lambda_{i-1}$ , we insert  $t$  steps during which only the vertices of  $V_i$  are recolored, where  $t = \text{poly}(\Delta)$  is the length of the schedule we produced for interval graphs (multiplied by  $k'$  to ensure the fact that the recolored vertices in each step go to the same color). Finally, after the last step of  $\lambda_{i-1}$ , we insert again  $t$  steps where only the vertices of  $V_i$  are recolored such that they can reach their target coloring.

Let us consider the step  $j$  of  $\lambda_{i-1}$ , and let  $\sigma$  be the coloring of  $G_i$  just before this step. Let  $c = j \bmod k'$  be the color with which vertices recolored at this step are recolored with. We know that  $\sigma|_{V_i}$  uses only  $\omega + 3$  colors. Our goal is to recolor the vertices of  $V_i$  so that no vertex is colored  $c$ , and the recoloring step of  $\lambda_{i-1}$  can be safely applied.

Since  $G_{i-1}$  is obtained from  $G_i$  by removing pending interval graphs, and separating interval graphs of diameter at least  $D$ , there is a coloring  $\sigma'$  of  $G_i$  which agrees with  $\sigma$  on  $G_{i-1}$  and which uses at most  $\omega + 1$  colors, all different from  $c$  for the vertices in  $V_i$ . Taking  $D = 240\omega^2\Delta^2$ , the same conditions as for interval graphs are satisfied, and there is a recoloring schedule from  $\sigma$  to  $\sigma'$  of length  $t$ . Similarly, after the last step of  $\lambda_{i-1}$ , if  $\sigma$  is the current coloring, then only  $\omega + 1$  colors are used in  $V_i$ , and by Theorem 1, there is a recoloring schedule from  $\sigma$  to  $\beta_i$  in  $G_i$  of length  $t$ .

This new schedule can be computed for each component of  $G[V_i]$  in a centralized way, and up to multiplying the length of the schedule by  $k'$ , we can assume that at step  $j$ , vertices are recolored with the color  $j \bmod k'$ . In a distributed setting, this requires  $O(D)$  steps, since each component has diameter  $O(D)$ . Since we need  $O(D)$  steps to extend the recoloring schedule from  $G_{i-1}$  to  $G_i$ , the algorithm uses at most  $O(D\ell)$  rounds in total. Moreover, the schedule has length at most  $t^\ell = (\text{poly } \Delta)^\ell$ . ◀

Now, for Theorem 2, from Lemma 16, we get  $\ell$  in  $O(\log n)$ , and we set  $D$  in  $O(\omega^2\Delta^2)$ , hence we have an algorithm in  $O(\omega^2\Delta^2 \log n)$  rounds that produces a schedule of length  $(\text{poly } \Delta)^{\log n}$  that is  $n^{O(\log \Delta)}$ .

---


## References

- 1 Pierre Aboulker, Marthe Bonamy, Nicolas Bousquet, and Louis Esperet. Distributed coloring in sparse graphs with fewer colors. *Electron. J. Comb.*, 26(4):P4.20, 2019.
- 2 Leonid Barenboim. On the locality of some np-complete problems. In Artur Czumaj, Kurt Mehlhorn, Andrew M. Pitts, and Roger Wattenhofer, editors, *ICALP 2012*, volume 7392, pages 403–415, 2012. doi:10.1007/978-3-642-31585-5\_37.
- 3 Leonid Barenboim and Michael Elkin. Sublogarithmic distributed MIS algorithm for sparse graphs using nash-williams decomposition. *Distributed Comput.*, 22(5-6):363–379, 2010. doi:10.1007/s00446-009-0088-2.
- 4 Leonid Barenboim and Michael Elkin. *Distributed Graph Coloring: Fundamentals and Recent Developments*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2013. doi:10.2200/S00520ED1V01Y201307DCT011.
- 5 Leonid Barenboim, Michael Elkin, and Cyril Gavoille. A fast network-decomposition algorithm and its applications to constant-time distributed computation. *Theor. Comput. Sci.*, 751:2–23, 2018. doi:10.1016/j.tcs.2016.07.005.
- 6 Jean RS Blair and Barry Peyton. An introduction to chordal graphs and clique trees. In *Graph theory and sparse matrix computation*, pages 1–29. Springer, 1993.
- 7 Marthe Bonamy and Nicolas Bousquet. Recoloring graphs via tree decompositions. *Eur. J. Comb.*, 69:200–213, 2018. doi:10.1016/j.ejc.2017.10.010.
- 8 Marthe Bonamy, Matthew Johnson, Ioannis Lignos, Viresh Patel, and Daniël Paulusma. Reconfiguration graphs for vertex colourings of chordal and chordal bipartite graphs. *J. Comb. Optim.*, 27(1):132–143, 2014. doi:10.1007/s10878-012-9490-y.
- 9 Marthe Bonamy, Paul Ouvrard, Mikaël Rabie, Jukka Suomela, and Jara Uitto. Distributed recoloring. In *DISC 2018*, volume 121, pages 12–1, 2018.
- 10 Nicolas Bousquet and Valentin Bartier. Linear transformations between colorings in chordal graphs. In *ESA 2019*, volume 144 of *LIPICs*, pages 24:1–24:15, 2019. doi:10.4230/LIPICs.ESA.2019.24.
- 11 Nicolas Bousquet, Laurent Feuilloley, Marc Heinrich, and Mikaël Rabie. Distributed recoloring of interval and chordal graphs. *CoRR*, abs/2109.06021, 2021. arXiv:2109.06021.

- 12 Nicolas Bousquet and Marc Heinrich. A polynomial version of cereceda’s conjecture. *CoRR*, abs/1903.05619, 2019.
- 13 Keren Censor-Hillel and Mikaël Rabie. Distributed reconfiguration of maximal independent sets. *Journal of Computer and System Sciences*, 112:85–96, 2020.
- 14 L. Cereceda. *Mixing Graph Colourings*. PhD thesis, London School of Economics and Political Science, 2007.
- 15 Shiri Chechik and Doron Mukhtar. Optimal distributed coloring algorithms for planar graphs in the LOCAL model. In *SODA 2019*, pages 787–804. SIAM, 2019. doi:10.1137/1.9781611975482.49.
- 16 Richard Cole and Uzi Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Inf. Control.*, 70(1):32–53, 1986. doi:10.1016/S0019-9958(86)80023-7.
- 17 M. Dyer, A. D. Flaxman, A. M Frieze, and E. Vigoda. Randomly coloring sparse random graphs with fewer colors than the maximum degree. *Random Structures & Algorithms*, 29(4):450–465, 2006.
- 18 Mohsen Ghaffari. Network decomposition and distributed derandomization (invited paper). In *SIROCCO 2020*, volume 12156, pages 3–18, 2020. doi:10.1007/978-3-030-54921-3\_1.
- 19 Mohsen Ghaffari and Christiana Lymouri. Simple and near-optimal distributed coloring for sparse graphs. In *DISC 2017*, volume 91 of *LIPICs*, pages 20:1–20:14, 2017. doi:10.4230/LIPICs.DISC.2017.20.
- 20 Andrew V. Goldberg, Serge A. Plotkin, and Gregory E. Shannon. Parallel symmetry-breaking in sparse graphs. *SIAM J. Discret. Math.*, 1(4):434–446, 1988. doi:10.1137/0401044.
- 21 Martin Charles Golumbic. *Algorithmic graph theory and perfect graphs*. Elsevier, 1980.
- 22 Magnús M. Halldórsson and Christian Konrad. Distributed algorithms for coloring interval graphs. In *DISC 2014*, volume 8784, pages 454–468, 2014. doi:10.1007/978-3-662-45174-8\_31.
- 23 Magnús M. Halldórsson and Christian Konrad. Improved distributed algorithms for coloring interval graphs with application to multicoloring trees. *Theor. Comput. Sci.*, 811:29–41, 2020. doi:10.1016/j.tcs.2018.11.028.
- 24 Christian Konrad and Viktor Zamaraev. Distributed minimum vertex coloring and maximum independent set in chordal graphs. In *MFCS 2019*, volume 138 of *LIPICs*, pages 21:1–21:15, 2019. doi:10.4230/LIPICs.MFCS.2019.21.
- 25 Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. On the locality of bounded growth. In *PODC 2005*, pages 60–68. ACM, 2005. doi:10.1145/1073814.1073826.
- 26 Nathan Linial. Locality in distributed graph algorithms. *SIAM J. Comput.*, 21(1):193–201, 1992. doi:10.1137/0221015.
- 27 Frédéric Maffray. On the coloration of perfect graphs. In *Recent Advances in Algorithms and Combinatorics*, pages 65–84. Springer, 2003.
- 28 Gary L. Miller and John H. Reif. Parallel tree contraction part 1: Fundamentals. *Adv. Comput. Res.*, 5:47–72, 1989.
- 29 Alessandro Panconesi and Aravind Srinivasan. The local nature of  $\delta$ -coloring and its algorithmic applications. *Combinatorica*, 15(2):255–280, 1995.
- 30 Marinus Johannes Petrus Peeters. On coloring  $j$ -unit sphere graphs. Technical report, Tilburg University, School of Economics and Management, 1991.
- 31 Ram Ramanathan. A unified framework and algorithm for channel assignment in wireless networks. *Wirel. Networks*, 5(2):81–94, 1999. doi:10.1023/A:1019126406181.
- 32 Johannes Schneider and Roger Wattenhofer. An optimal maximal independent set algorithm for bounded-independence graphs. *Distributed Comput.*, 22(5-6):349–361, 2010. doi:10.1007/s00446-010-0097-1.
- 33 Lieven Vandenberghe and Martin S. Andersen. Chordal graphs and semidefinite optimization. *Found. Trends Optim.*, 1(4):241–433, 2015. doi:10.1561/2400000006.




# Non-Blocking Dynamic Unbounded Graphs with Worst-Case Amortized Bounds

Bapi Chatterjee ✉ 

Indraprastha Institute of Information Technology Delhi, India

Sathya Peri ✉ 

Indian Institute of Technology Hyderabad, India

Muktikanta Sa ✉ 

Télécom SudParis – Institut Polytechnique de Paris, France

Komma Manogna ✉

Indian Institute of Technology Hyderabad, India

---

## Abstract

---

Today’s graph-based analytics tasks in domains such as blockchains, social networks, biological networks, and several others demand real-time data updates at high speed. The real-time updates are efficiently ingested if the data structure naturally supports dynamic addition and removal of both edges and vertices. These dynamic updates are best facilitated by concurrency in the underlying data structure. Unfortunately, the existing dynamic graph frameworks broadly refurbish the static processing models using approaches such as versioning and incremental computation. Consequently, they carry their original design traits such as high memory footprint and batch processing that do not honor the real-time changes. At the same time, multi-core processors—a natural setting for concurrent data structures—are now commonplace, and the analytics tasks are moving closer to data sources over lightweight devices. Thus, exploring a fresh design approach for real-time graph analytics is significant.

This paper reports a novel concurrent graph data structure that provides breadth-first search, single-source shortest-path, and betweenness centrality with concurrent dynamic updates of both edges and vertices. We evaluate the proposed data structure theoretically – by an amortized analysis – and experimentally via a C++ implementation. The experimental results show that (a) our algorithm outperforms the current state-of-the-art by a throughput speed-up of up to three orders of magnitude in several cases, and (b) it offers up to **80x** lighter memory-footprint compared to existing methods. The experiments include several counterparts: Stinger, Ligra and GraphOne. We prove that the presented concurrent algorithms are non-blocking and linearizable.

**2012 ACM Subject Classification** Theory of computation → Concurrent algorithms

**Keywords and phrases** concurrent data structure, linearizability, non-blocking, directed graph, breadth-first search, single-source shortest-path, betweenness centrality

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2021.20

**Related Version** *Full Version*: <https://arxiv.org/abs/2003.01697>

**Supplementary Material** *Software (Source Code)*: <https://github.com/sngraha/PANIGRAHAM>

**Funding** This work was partially funded by National Supercomputing Mission, Govt. of India under the project “Concurrent and Distributed Programming primitives and algorithms for Temporal Graphs”(DST/NSM/R&D\_Exascale/2021/16).

## 1 Introduction

A graph represents the pairwise relationships between objects or entities that underlie the complex frameworks such as blockchains, social networks, semantic-web, biological networks and many others. The contemporary applications of graph algorithms in real-time analytics,



© Bapi Chatterjee, Sathya Peri, Muktikanta Sa, and Komma Manogna;  
licensed under Creative Commons License CC-BY 4.0

25th International Conference on Principles of Distributed Systems (OPODIS 2021).

Editors: Quentin Bramas, Vincent Gramoli, and Alessia Milani; Article No. 20; pp. 20:1–20:25

Leibniz International Proceedings in Informatics



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

such as product recommendation or influential user tracking [31] over social network graphs, demand *dynamic* addition and removal of vertices and/or edges over time. Existing approaches for graph analytics can be broadly classified in *batch analytics*, e.g. GraphTinker [29], where a graph operation is performed over a static temporal snapshot of the data structure, and *stream analytics* e.g. Kineograph [14], where a temporal window of incoming data is studied. In general, these approaches inherently assume that the dynamic updates are monotonic: the structure of the graph largely remaining unaffected. A deviation from the assumed data ingestion pattern severely affects their design optimizations. Notwithstanding, in such techniques concurrency is predominantly limited in a “true” real-time sense. Furthermore, in anticipation of growing number of edges, they allocate a large chunk of memory. Recent trends show that there is an emerging niche for the analytics tasks closer to the data sources, such as mobile and edge devices [8]. On such platforms, though multi-core is getting progressively ubiquitous [40], unlike data-center-based settings, memory is limited and therefore the graph applications with unlimited dynamic updates must aim to have a lightweight memory footprint. In substance, the pursuit of an efficient lightweight real-time *concurrent graph analytics* framework with a fresh design approach is imperative.

### Concurrent Data Structures

With the rise of multi-core computers, *concurrent data structures* have become popular, for they are able to harness the power of multiple cores effectively. Several concurrent data structures have been developed in recent years such as: stacks [23], queues [4, 24, 32, 35], linked-lists [13, 21, 22, 48, 49, 50], hash tables [37, 38], binary search trees [6, 10, 13, 19, 39, 43], etc. On concurrent graphs, Kallimanis et al. [30] presented dynamic traversals and Chatterjee et al. [12] presented reachability queries. However, graph analytics queries, for example, single-source-shortest-path (SSSP) queries, which appertain to link-prediction in social networks or betweenness centrality, which finds applications in stock markets [44], are much more complex than reachability. The aforementioned queries inherently scan through (almost) the entire graph. In a dynamic setting, a concurrent update of a vertex or an edge can potentially render the output of such queries inconsistent.

To elucidate, consider computing the shortest path between two vertices. It requires exploring all possible paths between them, followed by returning the set of edges that make the shortest path. It is easy to see that an addition of an edge to another path can make it shorter than the one returned, and similarly, a removal of an edge (from it) could make it no longer the shortest. Imagine the addition and removal to be concurrent with the query, which can certainly benefit the application. Clearly, the return of the query can be inconsistent with the latest state of the graph.

In a concurrent dynamic graph, we require the updates and queries be *consistent*. To motivate, consider the computation of the risk-adjusted performance of a stock-portfolio via betweenness centrality [44]. In a dynamic setting, where the results of such analytics tasks influence the high-stake financial decisions, it is significant that a user is supplied with a consistent query result.

A commonly accepted correctness-criterion for concurrent data structures is *linearizability* [27], which intuitively infers that the output of a concurrent execution of a set of operations should appear as executed in a certain sequential order. Separating a graph query from concurrent updates by way of locking the shared vertices and edges can achieve linearizability. However, locking the portion of the graph that requires access by a query, which often could very well be its entirety, would obstruct a large number of concurrent fast updates. Even an effortful interleaving of the query- and update-locks at a finer granularity



does not protect against pitfalls such as deadlock, convoying, etc [25]. A more attractive option is to implement *non-blocking (lock-free)* progress, which ensures that some non-faulty (non-crashing) threads complete their operations in a finite number of steps [26]. Surprisingly, non-blocking linearizable design of queries that synchronize with concurrent updates in a dynamic graph are difficult.

**Proposed work.** In this paper, we describe the design and implementation of a graph data structure, which provides (a) three useful operations – breadth-first search (BFS), single-source shortest-path (SSSP), and betweenness centrality (BC), (b) dynamic updates of edges and vertices concurrent with the operations, (c) non-blocking progress with linearizability, and (d) a light memory footprint. We call it PANIGRAHAM <sup>a</sup>: **P**actical **N**on-blocking **G**raph **A**lgorithms.

## Algorithm Overview

In a nutshell, we implement a concurrent non-blocking dynamic directed graph data structure as an *adjacency-list* formed by a composition of lock-free sets: a lock-free hash-table and multiple lock-free binary search trees (BSTs). The set of outgoing edges  $E_v$  from a vertex  $v \in V$  is implemented by a BST, whereas,  $v$  itself is a node of the hash-table (as shown in Figure 1). Addition/removal of a vertex amounts to the same operation of a node in the lock-free hash-table, whereas, addition/removal of an edge translates likewise to a lock-free BST. Although lock-free progress is composable [16], thereby ensuring lock-free updates in the graph, however, optimizing these operations is nontrivial as shown by us in this paper. The operations – BFS, SSSP, BC – are implemented by specialized partial snapshots of the composite data structure. In a dynamic concurrent non-blocking setting, we apply multi-scan/validate [1] to ensure the linearizability of a partial snapshot. We prove that these operations are *non-blocking*. The empirical results show the effectiveness of our algorithms.

## Related work

Libraries of parallel implementation of graph operations are abundant in literature. A relevant survey can be found in [5]. To mention a few well-known ones: PowerGraph [20], Galois [33], Ligra [45], Ligra+ [46], MGraph [51], Congra [42], Congra+ [41]. However, they primarily focus on static queries and natively do not allow updates to the data structure, let alone concurrency.

Broadly, these libraries use the *compressed sparse row (CSR)* format, a read-only representation, to implement a graph. In principle, the basic designs of an adjacency list and the CSR are almost identical [47], however, in practice the CSR exhibits better cache efficiency due to locality [7]. In a dynamic setting, a serious drawback of the CSR format is the need for reprocessing the entire structure for vertex updates and the array that stores edge information for edge updates.

To our knowledge, Stinger [18] was the first large-scale practical implementation that supported dynamic updates in a graph. They implement a graph as an *edge-list*: edges incident on a vertex are stored in a linked-list of edge-blocks. The vertices constitute a logical vertex array, thereby the edge-blocks are referenced. The edge-blocks contain the metadata such as timestamps and mark of valid edges. In practice, they allocate a big chunk

<sup>a</sup> Panigraham is the Sanskrit translation of Marriage, which undoubtedly is a prominent event in our lives resulting in networks represented by graphs.

## 20:4 Non-Blocking Dynamic Unbounded Graphs

of memory (by default, half of the available system memory) to minimize cost of allocation for addition of edges after initialization. The removal of both edges and vertices is provided via metadata-based marks. However, vertex addition requires copying the entire structure. Furthermore, by design update operations are not allowed to be concurrent with queries. In contrast, PANIGRAHAM is concurrent, non-blocking, uses a hash-table for vertex-list and BSTs to contain edge-nodes. Moreover, the memory consumption is determined by the actual data contained in the data structure. Stinger was extended and optimized in some recent works such as GraphOne [34], GraphTinker [29]. GraphOne hybridizes an edge-list and an adjacency-list to support batch processing. Their methodology maintains versions of these lists to provide intermittent batch updates to an analytics engine. Clearly, for real-time lightweight settings, this method suffers from similar drawbacks as Stinger. On the other hand, GraphTinker builds upon Stinger and replaces linear probing with better hashed searches on the edge-lists. Their approach also shows some load balancing as the data structure grows. Nevertheless, none of these methods are efficient for dynamic vertex additions, and updates and queries are inherently sequentialized. By contrast, PANIGRAHAM provides fully concurrent queries and updates as a fundamental design component and ensures correctness (linearizability). Aspen [17] is another recent framework that extends Stinger to support graph updates with graph queries. However, the interface provided by them is very different - acquire, set and release. It is not immediately clear how to use their framework for concurrent graph updates and compare it with our framework.

### Contributions and paper summary

- First, we describe the non-blocking directed graph data structure as a composition of lock-free hash table and binary search trees. (Section 2)
- After that, we introduce our novel framework as an interface operation with its correctness and progress guarantee (Section 3) followed by the descriptions of concurrent implementation of BFS, SSSP, and BC.
- We present an experimental evaluation of our algorithm comparing it against the existing parallel graph libraries Ligra [45] and Stinger [18] with respect to the throughput and memory footprint (Section 4). Our experiments demonstrate the power of non-blocking concurrency for dynamic updates in an application. Utilizing the parallel compute resources - 56 threads - in a standard multi-core machine, our implementation performs in some cases (a) up to 5x better than Graphone (b) up to 10x better than Ligra and (c) 40x better than Stinger for BFS, SSSP, and BC algorithms. Significantly, for an identically initialized data structure and an identical random orderly selection of graph operations, we achieve up to 80x lighter memory footprint compared to Stinger (Section 4). In comparative terms, the most recent counterpart of our work is GraphTinker [29], who report up to 4x speedup in comparison with Stinger. Thus, the presented algorithm outperforms its latest competitor.
- Finally, we present an amortized analysis (Section 5) to theoretically contrast the worst case cost of our method against that of Ligra and Stinger. To the best of our knowledge, this is the first work on amortized upper bound for concurrent dynamic graph operations.

## 2 Non-blocking Graph Data Structure

### Preliminaries

Our discussion uses a standard shared-memory model that supports atomic `read`, `write`, `fetch-and-add` (FAA), and `compare-and-swap` (CAS) instructions.

**Background on the Graph Operations.** A *graph* is represented as a pair  $G = (V, E)$ , where  $V$  is the set of *vertices* and  $E$  is the set of *edges*. An edge  $e \in E$ ,  $e := (u, v)$  represents a pair of vertices  $u, v \in V$ . In a *directed graph*<sup>b</sup>  $e := (u, v)$  is an ordered pair, thus has an associated direction: *emanating* (*outgoing*) from  $u$  and *terminating* (*incoming*) at  $v$ . We denote the set of outgoing edges from  $v$  by  $E_v$ . Thus,  $\cup_{v \in V} E_v = E$ . Each edge  $e \in E$  has a *weight*  $w_e$ . A node  $v \in V$  is said *reachable* from  $u \in V$ :  $v \leftrightarrow u$  if there are consecutive edges  $\{e_1, e_2, \dots, e_n\} \subseteq E$  such that  $e_1$  emanates from  $u$  and  $e_n$  terminates at  $v$ .

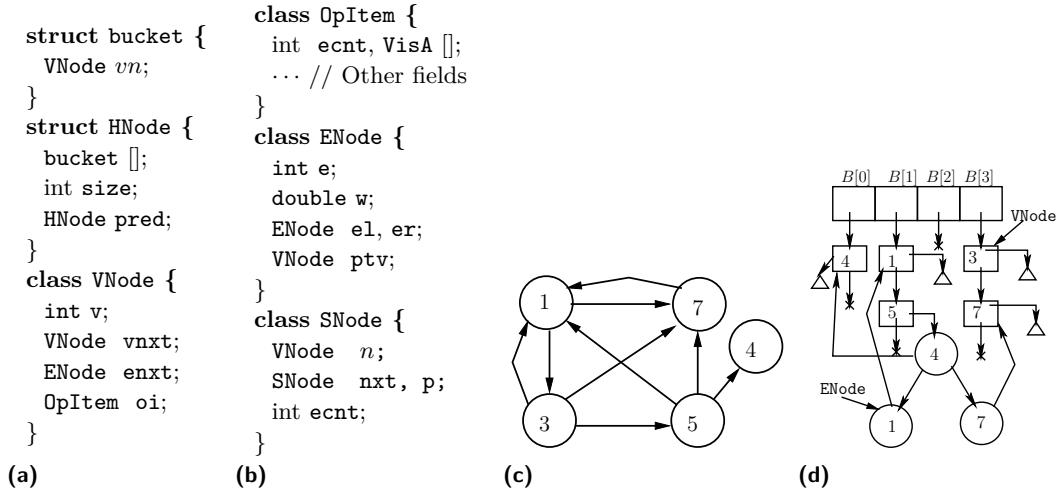
1. **Breadth First Search (BFS):** Given a query vertex  $v \in V$ , output each vertex  $u \in V - v$  reachable from  $v$ . The collection of vertices happens in a *BFS order*: those at a distance  $d_1$  from  $v$  are collected before those at a distance  $d_2 > d_1$ .
2. **Single Source Shortest Path (SSSP):** Given a vertex  $v \in V$ , find a shortest path with respect to total edge-weight from  $v$  to every other vertex  $u \in V - v$ . Note that, given a pair of nodes  $u, v \in V$ , the shortest path between  $u$  and  $v$  may not be unique.
3. **Betweenness Centrality (BC):** Given a vertex  $v \in V$ , compute  $BC(v) = \sum_{s, t \in V} \frac{\sigma(s, t|v)}{\sigma(s, t)}$ , where  $\sigma(s, t)$  is the number of shortest paths between vertices  $s, t \in V$  and  $\sigma(s, t|v)$  is that passing through  $v$ .  $BC(v)$  indicates the prominence of  $v$  in  $V$  and finds several applications where influence of an entity in a network is to be measured.

### The Abstract Data Type (ADT)

Consider a weighted directed graph  $G = (V, E)$  as defined before. A vertex  $v \in V$  has an immutable unique *key* drawn from a totally ordered universe. For brevity, we denote a vertex with key  $v$ :  $v(v)$  by  $v$  itself. Extending on the notations used in Section 1, we denote a directed edge with weight  $w$  from the vertex  $v_1$  to  $v_2$  as  $(v_1, v_2|w) \in E$ . We consider an ADT  $\mathcal{A}$  as a set of operations:  $\mathcal{A} = \{\text{PUTV}(v), \text{REMV}(v), \text{GETV}(v), \text{PUTE}(v_1, v_2|w), \text{REME}(v_1, v_2), \text{GETE}(v_1, v_2), \text{BFS}(v), \text{SSSP}(v), \text{BC}(v)\}$  on  $G$ .

1. A  $\text{PUTV}(v)$  updates  $V$  to  $V \cup v$  and returns `true` if  $v \notin V$ , otherwise it returns `false` without any update.
2. A  $\text{REMV}(v)$  updates  $V$  to  $V - v$  and returns `true` if  $v(v) \in V$ , otherwise it returns `false` without any update.
3. A  $\text{GETV}(v)$  returns `true` if  $v \in V$ , and `false` if  $v \notin V$ .
4. A  $\text{PUTE}(v_1, v_2|w)$ 
  1. updates  $E$  to  $E \cup (v_1, v_2|w)$  and returns  $\langle \text{true}, \infty \rangle$  if  $v_1 \in V \wedge v_2 \in V \wedge (v_1, v_2|\cdot) \notin E$ ,
  2. updates  $E$  to  $E - (v_1, v_2|z) \cup (v_1, v_2|w)$  and returns  $\langle \text{true}, z \rangle$  if  $(v_1, v_2|z) \in E$ ,
  3. returns  $\langle \text{false}, w \rangle$  if  $(v_1, v_2|w) \in E$  without updates,
  4. returns  $\langle \text{false}, \infty \rangle$  if  $v_1 \notin V \vee v_2 \notin V$  without updates.
5. A  $\text{REME}(v_1, v_2)$  updates  $E$  to  $E - (v_1, v_2|w)$  and returns  $\langle \text{true}, w \rangle$  if  $(v_1, v_2|w) \in E$ , otherwise it returns  $\langle \text{false}, \infty \rangle$  without any update.
6. A  $\text{GETE}(v_1, v_2)$  returns  $\langle \text{true}, w \rangle$  if  $(v_1, v_2|w) \in E$ , otherwise it returns  $\langle \text{false}, \infty \rangle$ .

<sup>b</sup> In this paper we confine the scope of discussion to directed graphs only.



■ **Figure 1** (a) and (b) Data structure components, (c) A sample directed graph, (d) Our implementation of (c).

7. A  $\text{BFS}(v)$ , if  $v \in V$ , returns a sequence of vertices reachable from  $v$  arranged in a BFS order as defined before. If  $v \notin V$ , it returns  $\text{NULL}$ .
8. An  $\text{SSSP}(v)$ , if  $v \in V$ , returns a set  $S(v) = \{d(v_i)\}_{v_i \in V}$ , where  $d(v_i)$  is the summation of the weights of the edges<sup>c</sup> on the shortest-path between  $v$  and  $v_i$  if  $v_i \leftrightarrow v$ , and  $d(v_i) = \infty$ , if  $v_i \not\leftrightarrow v$ . Note that  $d(v) = 0$ . There can be multiple paths between  $v$  and  $v_i$  with the same sum of edge-weights. If  $v \notin V$ , it returns  $\text{NULL}$ .
9. A  $\text{BC}(v)$  returns the betweenness centrality of  $v$  as defined before, if  $v \in V$ . It returns  $\text{NULL}$  if  $v \notin V$ .

A precondition for  $(v_1, v_2 | w) \in E$  is  $v_1 \in V \wedge v_2 \in V$ .

### Data Structure Components

To facilitate both an efficient traversal and lock-freedom, we build the data structure based on a composition of a lock-free hash-table implementing the vertex-list, and lock-free BSTs implementing the edge-lists. On a skeleton of this composition, we include the design components for efficient traversals and (partial) snapshots. This is a more efficient design as compared to Chatterjee et al.’s approach [12] where the component dictionaries are implemented using lock-free linked-lists only.

More specifically, the nodes of the vertex-list are instances of the class `VNode`, see Figure 1(a). A `VNode` contains the key of the corresponding vertex along with a pointer to a BST implementing its edge-list. The most important member of a `VNode` is a pointer to an instance of the class `OpItem`, which facilitates anchoring of the traversals as described above.

The `OpItem` class, see Figure 1(b), encapsulates an array `VisA` of the size equal to the number of threads in the system, a counter `ecnt`, and other algorithm specific indicators, which we describe in Section 3 while specifying the queries. An element of `VisA` simply keeps a count of the number of times the node is visited by a query performed by the corresponding

<sup>c</sup> We limit our discussion to positive edge-weights only.

thread. The counter `ecnt` is incremented every time an outgoing edge is added or removed at the vertex. This serves an important purpose of notifying a thread if the same edge is removed and added since the last visit.

The class `ENode`, see Figure 1(b), structures the nodes of an edge-list. It encapsulates a key, the edge-weight, the left- and right-child pointers and a pointer to the associated `VNode` where the edge terminates; the key in the `ENode` is that of the `VNode`; thus each `ENode` delegates a directed edge. The `VNodes` are bagged in a linked-list being referred to by a pointer from the `buckets`, see Figure 1(a). A resizable hash-table is constructed of the arrays of these `buckets`, wherein arrays are linked in the form of a linked-list of `HNodes`.

At the bare-bones level, our resizable vertex-list derives from the lock-free hash-table of Liu et al. [37], whereas the edge-lists extend the lock-free BST of Howley et al. [28]. We introduce the `OpItem` fields in hash-table nodes. To facilitate non-recursive traversal in the lock-free BST, we use stacks. As we explain later, aligning the operations of the hash-table to the state of `OpItem` therein brings in nontrivial challenges.

The last but a significant component of our design is the class `SNode`, see Figure 1(b). It encapsulates the information to validate a scan of the graph to output a consistent specialized partial snapshot. More specifically, it packs the pointers to `VNodes` visited during a scan along with two pointers `nxt` and `p` to keep track of the order of their visit. The field `ecnt` records the `ecnt` counter of the corresponding visited `VNode`, which enables checking if the visited `VNode` has had any addition or removal of an edge since the last visit.

## Non-blocking Data Structure Construction

Having these components in place, we construct a non-blocking graph data structure in a modular fashion. Refer to Figure 1(d) depicting a partial implementation of a sample directed graph shown in Figure 1(c). The `ENodes`, shown as circles in Figure 1(d), with their children and parent pointers make lock-free internal BSTs corresponding to the edge-lists. For simplicity we have only shown the outgoing edges of vertex 5 in Figure 1(d) while the edges of other vertices are represented by small triangles. Thus, whenever a vertex has outgoing edges, the corresponding `VNode`, shown as small rectangles therein, has a non-null pointer pointing to the root of a BST of `ENodes`. The `VNodes` themselves make sorted lock-free linked-lists connected to the buckets of a hash-table. The buckets are cells of a bucket-array that implement the lock-free hash-table. When required, we add/remove bucket-arrays for an unbounded resizable dynamic design. The lock-free `VNode`-lists have two sentinel `VNodes`: `vh` and `vt` initialized with keys  $-\infty$  and  $\infty$ , respectively.

We adopt the well-known technique of *pointer marking* – using a single-word CAS – via bit-stealing [28, 37] to perform lazy non-blocking removal of nodes. Concretely, on a common x86-64 architecture, memory has a 64-bit boundary and the last three least significant bits are unused; this allows us to use the last significant bit of a pointer to indicate first a *logical removal* of a node and thereafter detaching it from the data structure. Specifically, an `HNode`, a `VNode`, and an `ENode` is logically removed by marking its `pred`, `vnxt`, and `el` pointer, respectively. We call a node *alive* which is not logically removed.

### 3 PANIGRAHAM Framework

In this section, we describe a non-blocking algorithm that implements the ADT  $\mathcal{A}$ . The operations  $\mathcal{M} := \{\text{PUTV}, \text{REMV}, \text{GETV}, \text{PUTE}, \text{REME}, \text{GETE}\} \subset \mathcal{A}$  use the interface of the hash-table and BST with interesting non-trivial adaptation to our purpose. In the permitted space we describe the execution, correctness and progress property of the operations

```

1: Operation OP( $v$ )
2:    $tid \leftarrow \text{GETTHID}()$ ; // get thread-id
3:   if (ISMRKD( $v$ )) then
4:     return NULL; //Vertex is not present
5:   return SCAN( $v, tid$ ); //Invoke Scan


---


6: Method SCAN( $v, tid$ )
7:    $\text{list}(\text{SNode } ot, nt)$  ; //Trees to hold the nodes
8:    $ot \leftarrow \text{TREECOLLECT}(v, tid)$ ; //1st Collect
9:   while (true) do //Repeat the tree collection
10:     $nt \leftarrow \text{TREECOLLECT}(v, tid)$ ; //2nd Collect
11:    if (CMP TREE( $ot, nt$ )) then
12:      return  $nt$ ; //return if two collects are equal
13:     $ot \leftarrow nt$ ;


---


14: Method CMP TREE( $ot, nt$ )
15:   if ( $ot = \text{NULL} \vee nt = \text{NULL}$ ) then
16:     return false;
17:    $oit \leftarrow ot.\text{head}, nit \leftarrow nt.\text{head}$ ;
18:   while ( $oit \neq ot.\text{tail} \wedge nit \neq nt.\text{tail}$ ) do
19:     if ( $oit.n \neq nit.n \vee oit.\text{ecnt} \neq nit.\text{ecnt} \vee$   

 $oit.p \neq nit.p$ ) then
20:       return false; //Both the trees are not equal
21:      $oit \leftarrow oit.\text{nxt}; nit \leftarrow nit.\text{nxt}$ ;
22:     if ( $oit.n \neq nit.n \vee oit.\text{ecnt} \neq nit.\text{ecnt} \vee oit.p$   

 $\neq nit.p$ ) then //Both the trees are not equal
23:       return false ;
24:     else return true ; //Both the trees are equal


---


25: Method CHK VISIT( $adjn, tid, count$ )
26:   if ( $adjn.\text{oi.VisA}[tid] = count$ ) then
27:     return true;
28:   else return false ;


---


29: Method TREECOLLECT( $v, tid$ )
30:    $\text{queue}(\text{SNode } que)$  ; //Queue used for traversal
31:    $\text{list}(\text{SNode } st; cnt \leftarrow cnt + 1)$  ; //List to keep
of the visited nodes
32:    $v.\text{oi.VisA}[tid] \leftarrow cnt$ ;
33:    $sn \leftarrow \text{new CTNODE}(v, \text{NULL}, \text{NULL},$   

 $v.\text{oi.ecnt})$ ; //Create a new SNode
34:    $st.\text{ADD}(sn); que.\text{enqueue}(sn)$ ;
35:   while ( $\neg que.\text{empty}()$ ) do //Iterate all vertices
36:      $cvn \leftarrow que.\text{dequeue}()$ ; // Get the front node
37:     if (ISMRKD( $cvn$ )) then
38:       continue; // If marked then continue
39:      $itn \leftarrow cvn.n.\text{enxt}$ ; //Get the root ENode
40:      $\text{stack}(\text{ENode } S)$ ; // stack for inorder traversal
41:     /*Process all neighbors of  $cvn$  in the order of
inorder traversal, as the edge-list is a BST*/
42:     while ( $itn \vee \neg S.\text{empty}()$ ) do
43:       while ( $itn$ ) do
44:         if ( $\neg \text{ISMRKD}(itn)$ ) then
45:            $S.\text{push}(itn)$ ; // push the ENode
46:          $itn \leftarrow itn.\text{el}$ ;
47:          $itn \leftarrow S.\text{pop}()$ ;
48:         if ( $\neg \text{ISMRKD}(itn)$ ) then //Validate it
49:            $adjn \leftarrow itn.\text{ptv}$ ;
50:           if ( $\neg \text{ISMRKD}(adjn)$ ) then //Validate it
51:             if ( $\neg \text{CHK VISIT}(adjn, tid, cnt)$ ) then
52:                $adjn.\text{oi.VisA}[tid] \leftarrow cnt$ ; //Mark it
53:               //Create a new SNode
54:                $sn \leftarrow \text{new CTNODE}(adjn,$   

 $cvn, \text{NULL}, adjn.\text{oi.ecnt})$ ;
55:                $st.\text{ADD}(sn)$ ; //Insert  $sn$  to  $st$ 
56:                $que.\text{enqueue}(sn)$ ; //Push  $sn$  into the  $que$ 
57:              $itn \leftarrow itn.\text{er}$ ;
58:           return  $st$ ; //The tree is returned to the SCAN

```

■ **Figure 2** Framework interface operation for graph queries.

$\mathcal{Q} := \{\text{BFS, SSSP, BC}\} \subset \mathcal{A}$ . To de-clutter the presentation, we encapsulate the three queries in a unified framework. The framework comes with an interface operation OP. OP is specialized to the requirements of the three queries. The functionality of OP is presented in pseudo-code in Figures 2. Due to space constraints pseudo-code of the operations BFS, SSSP, and BC and detail descriptions are presented in the technical report [11].

Before describing the algorithm, it is important to specify its correctness and progress guarantee. In essence, we need to establish that during any execution the invariants corresponding to a consistent state of the data structure are satisfied, which are: (a) each edge-list maintains a BST order based on the ENode’s key  $e$ , and alive ENodes are reachable from  $\text{enxt}$  of the corresponding VNode, (b) a VNode that holds a pointer to a BST containing any alive ENode is itself alive, (c) each alive VNode is reachable from  $\text{vh}$  and vertex-lists connected to buckets are sorted based on the VNode’s keys  $v$ , and (d) an HNode which contains a bucket holding a pointer to an alive VNode is itself alive and an alive HNode is always connected to the linked-list of HNodes.

To prove *linearizability* [27], we describe the execution generated by the data structure as a collection of method invocation and response events. We assign an atomic step between the invocation and response as the *linearization point* (LP) of a method call (operation). Ordering the operations by their LPs provide a sequential history of the execution. We prove the correctness of the data structure by assigning a sequential history to an arbitrary

execution which is valid, i.e., it maintains the invariants. Furthermore, we argue that the data structure is non-blocking by showing that the queries would return in a finite number of steps if no update operation happens and hence is *obstruction-free* [26]. Moreover in an arbitrary execution at least one update operation returns in a finite number of steps and as a result is *lock-free* [26]. The details are provided in technical report [11].

**Pseudo-code convention.** We use  $p.x$  to denote the member field  $x$  of a class object pointer  $p$ . To indicate multiple return objects from an operation we use  $\langle x_1, x_2, \dots, x_n \rangle$ . To represent pointer-marking, we define three procedures: (a)  $\text{ISMRKD}(p)$  returns **true** if the last significant bit of the pointer  $p$  is set to 1, else, it returns **false**, (b)  $\text{MRK}(p)$  sets the last significant bit of  $p$  to 1, and (c)  $\text{UNMRK}(p)$  sets the same to 0. An invocation of  $\text{CVNODE}(v)$ ,  $\text{CENODE}(e)$  and  $\text{CTNODE}(v)$ , creates a new **VNode** with key  $v$ , a new **ENode** with key  $e$  and a new **SNode** with a **VNode**  $v(v)$  respectively. For a newly created **VNode**, **ENode** and **SNode** the pointer fields are initialized with **NULL** value.

The execution pipeline of **OP** is presented at lines 1 to 5 in Algorithm 2. **OP** intakes a query vertex  $v$ . It starts with checking if  $v$  is alive at Line 3. In the case  $v$  was *not alive*, it returns **NULL**. For this execution case, which results in **OP** returning **NULL**, the **LP** is at the atomic step (a) where **OP** is invoked in case  $v$  was not in the data structure at that point, and (b) where  $v$  was logically removed using a **CAS** in case it was alive at the invocation of **OP**.

Now, if  $v$  was alive, it proceeds to perform the method **SCAN**, Line 6 to 13. **SCAN** repeatedly performs (specialized partial) snapshot collection of the data structure along with comparing every consecutive pair of scans, stopping when a consecutive pair of collected snapshots are found identical. Snapshot collection is structured in the method **TREECOLLECT**, Line 29 to 59, whereas comparison of collected snapshots is performed by the method **CMP TREE**, Line 14 to 24.

Method **TREECOLLECT** performs a **BFS** traversal in the data structure to collect pointers to the traversed **VNodes**, thereby forming a tree. A cell of **VisA** corresponding to thread-id is marked on visiting it; notice that it is adaptation of the well-known use of node-dirty-bit for **BFS** [15]. The traversal over **VNodes** is facilitated by a queue: Line 30, whereas, exploring the outgoing edges at each **VNode**, equivalently, traversing over the **BST** corresponding to its edge-list uses a stack: Line 40. The snapshot collection for the queries **BFS** and **BC** are identical. For **SSSP**, where edge-weights are to be considered, the snapshot collection is optimized in each consecutive scan based on the last collection. At the core, the collected snapshot is a list of **SNodes**, where each **SNode** contains a pointer to a **VNode**, pointers to the next and previous **SNodes** and the value of the **ecnt** field of the **OpItem** of the **VNode**.

Method **CMP TREE** essentially compares two snapshots in three aspects: whether the collected **SNodes** contain (a) pointers to the same **VNodes** (b) have the same **SNode** being pointed by previous and next, and (c) have the same **ecnt**. The three checks ensure that a consistent snapshot is the one which has its collection lifetime *not concurrent* to (a) a vertex either added to or removed from it, (b) a path change by way of addition or removal of an edge, and, (c) an edge removed and then added back to the same position, respectively. Thus, at the completion of these checks, if two consecutive snapshots match, it is guaranteed to be unchanged during the time of the last two **TREECOLLECT** operations. Clearly, we can put a linearization point just after the atomic step where the last check is done: Line 19 or 22, where **CMP TREE** returns.

Now, it is clear that any  $q \in \mathcal{Q}$  does not engage in helping any other operation. Furthermore, an  $m \in \mathcal{M}$  does not help a  $q \in \mathcal{Q}$ . Thus, given an execution  $E$  as a collection of operation calls belonging to  $\mathcal{A}$ , by the fact that the data-structures hash-table and **BST** are

## 20:10 Non-Blocking Dynamic Unbounded Graphs

lock-free, and whenever no `PUTV`, `PUTE`, `REMV`, and `REME` happen, a  $q \in \mathcal{Q}$  returns, we infer that the presented algorithm is non-blocking. In Appendix A, we present the details of each of the operations.

Fundamentally, the functionalities of BFS, SSSP and BC queries are tailored by specialized construction of corresponding `OpItem` objects according to the requirements of their respective partial snapshots. As mentioned above, these queries are obstruction-free. In the technical report [11], we present the details of each of the queries.

### 4 Experiments

In this section, we describe the experimental evaluation of our non-blocking graph algorithms against three well-known existing batch analytics methods: (a) **Stinger** [18], (b) **Ligra** [45], and (c) **GraphOne** [34].

**Dataset.** We use (a) a standard synthetic dataset – **R-MAT** graphs [9] – with power-law distribution of degrees, and (b) real-life **SNAP**{EmailEuAll, Slashdot0811, socEpinions1, and WikiVot} [36] graph dataset.

**Algorithms.** While **Stinger** provides dynamic edge addition and removal and vertex removal operations, **Ligra** is built for static queries. However, these libraries do not allow concurrent updates with queries: we execute dynamic vertex and edge updates by intermittent sequential addition and removal. As explained earlier, we needed the repeated snapshot collection and validation methodology to guarantee linearizability of graph queries. However, if the consistency requirement is not as strong as linearizability, we can still have non-blocking progress if we collect the snapshots only once, i.e., we stop the scan algorithm after a single round of snapshot collection. At the cost of theoretical consistency, we gain a lot in terms of throughput, which is the primary demand of the analytics applications, who often go for approximate queries. Thus, the experiments include the following methods: (1) **PG-Cn**: Linearizable PANIGRAHAM, (2) **PG-Icn**: Inconsistent PANIGRAHAM, (3) **Ligra**, (4) **Stinger**, and (5) **GraphOne**. Note that, while all the libraries provide BFS queries, only **Ligra** supports SSSP and BC.

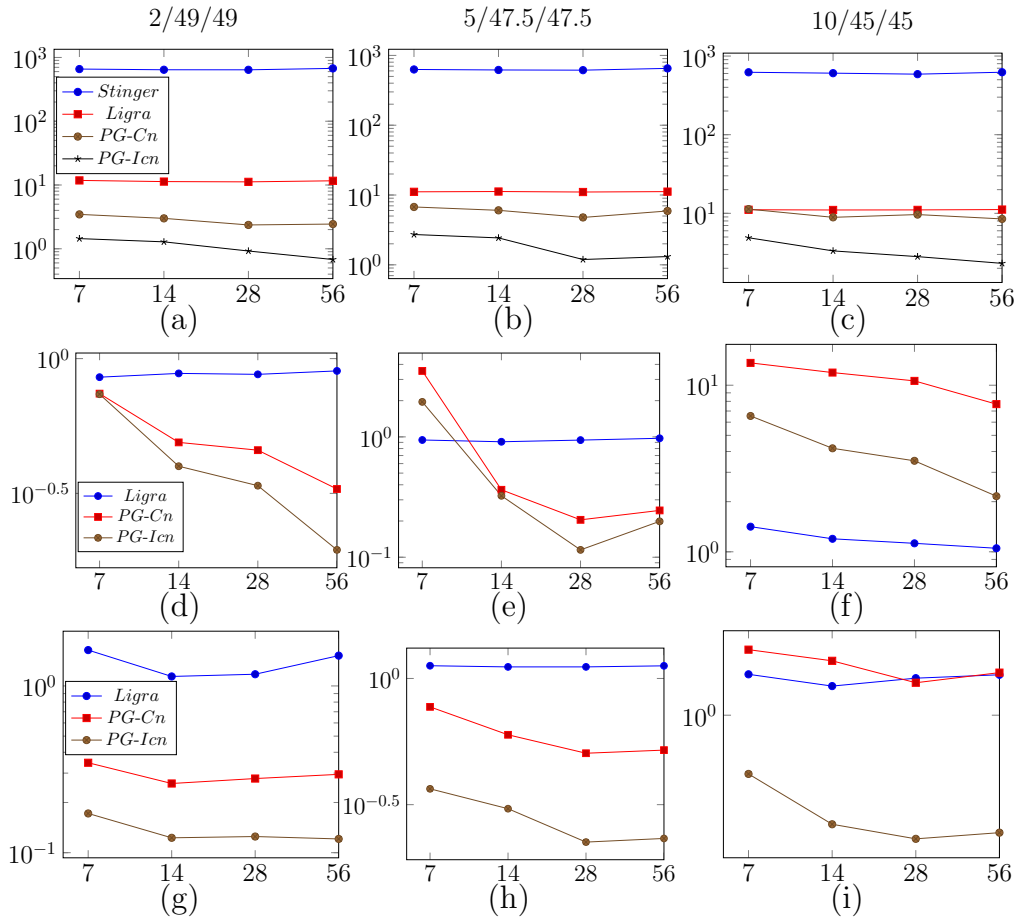
**The choice of the competitors.** One clear advantage of PANIGRAHAM over each of the lately developed dynamic graph frameworks, such as **GraphOne** [34] and **GraphTinker** [29], is that in a dynamic setting these frameworks do not provide any direct or intuitive method for vertex removal. The dynamic property of the graph in these frameworks is primarily with regards to the edges. While keeping the requirement for having support of dynamic vertex and edges, we zeroed on **Ligra** [18] and **Stinger** [45] for comparisons. We also compare the results of BFS on **GraphOne** [34] with concurrent `PUTE`, and `REME` operations by keeping a fixed number of vertices.

**Experimental Setup.** We conducted our experiments on a system with Intel(R) Xeon(R) E5-2690 v4 CPU packing 56 cores with a clock speed of 2.60GHz. There are 2 logical threads for each core and each having a private 64KB L1 and 256KB L2 cache. The 35840KB L3 cache is shared across the cores. The system has 32GB of RAM and 1TB of hard disk. It runs on a 64-bit Linux operating system. All implementations<sup>d</sup> are in C++ without garbage collection. We used Posix threads for multi-threaded implementation.

---

<sup>d</sup> The source code is available on <https://github.com/PDCRL/PANIGRAHAM>.



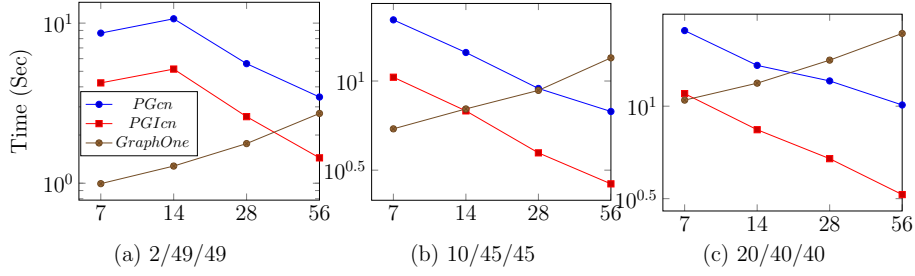


■ **Figure 3** Latency of the executions containing OP: (1) BFS ((a), (b), and (c)) on a graph of size  $|V|=131K$  and  $|E|=2.44M$ , (2) SSSP ((d), (e), and (f)) on a graph of size  $|V|=8K$  and  $|E|=80K$ , and (3) BC ((g), (h), and (i)) on a graph of size  $|V|=16K$  and  $|E|=160K$ . X-axis and Y-axis units are the number of threads and time in second, respectively.

The experiments start with a graph instance populating the data structure. At the execution initialization, we spawned a fixed set of threads (7, 14, 28 and 56). During the runtime, each thread randomly performed a set of operations chosen from a certain random workload distribution. The random workload pre-constructed and the same across all experiments. Each experiment was executed for 5 iterations. After ignoring the initial 5% operations for warm-up and we took the average of the remaining operations. We considered two evaluation metrics: (i) the latency: total time taken to complete the set of operations, after a fixed warm-up – 5% of the total number of operations, and (ii) the memory footprint.

**Workload Distribution.** In each micro-benchmark, first we loaded a graph instance, thereafter performed warm-up operations, followed by an end-to-end run of  $10^4$  operations in total, assigned in a uniform random order to the threads. We used a range of distributions over an ordered (family of) set of operations:  $\{OP, \text{Vertex-Updates}=\{\text{PUTV}, \text{REMV}\}, \text{Edge-Updates}=\{\text{PUTE}, \text{REME}\}\}$ . A sample label, say, 20/60/20 on a performance plot refers to a distribution  $\{OP : 20\%, \{\text{PUTV} : 30\%, \text{REMV} : 30\%\}, \{\text{PUTE} : 10\%, \text{REME} : 10\%\}\}$ .

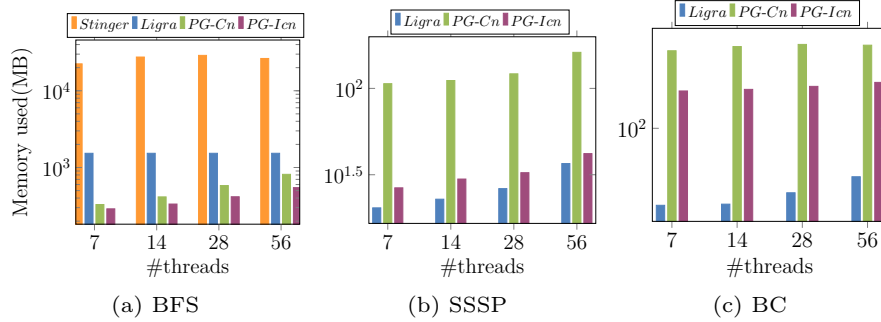
## 20:12 Non-Blocking Dynamic Unbounded Graphs



■ **Figure 4** Latency of the executions containing OP: BFS ((a), (b), and (c)) on a graph of size  $|V|= 65K$  and  $|E|= 500K$ . Total  $10^4$  operations were performed with given distributions. The distributions for each cases is: BFS/PUTE/REME, e.g., 2/49/49 : {BFS : 2%, PUTE : 49%, REME : 49%}. X-axis unit is the number of threads.

## Experimental Observations and Discussion

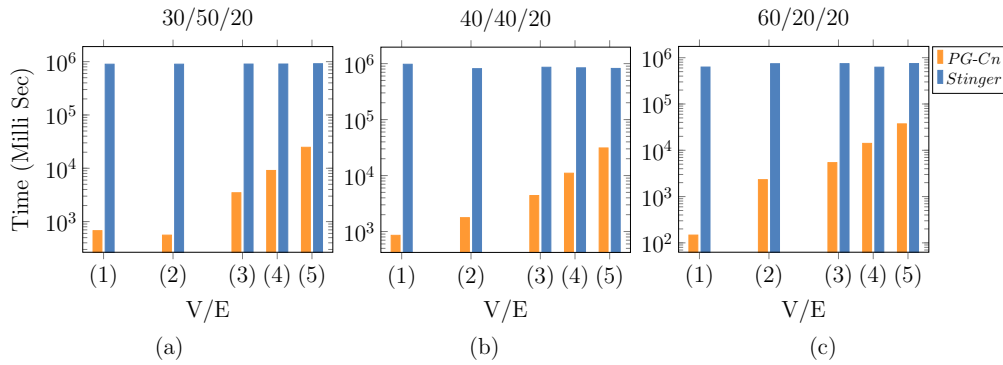
Figure 3 to 10 show the evaluation results. In the following we highlight the significant experimental observations.



■ **Figure 5** The memory footprint during the run-time corresponding to the workload distribution 10/45/45 as plotted in Figures 3(c), 3(f) and 3(i).

**Scalability.** See Figure 3; the concurrent methods scale well with the number of threads irrespective of the workload and graph size, whereas Stinger shows negligible scalability. With higher proportion of queries in the workload, Ligra starts scaling. This shows that concurrency in dynamic analytics is a natural way to scale-up.

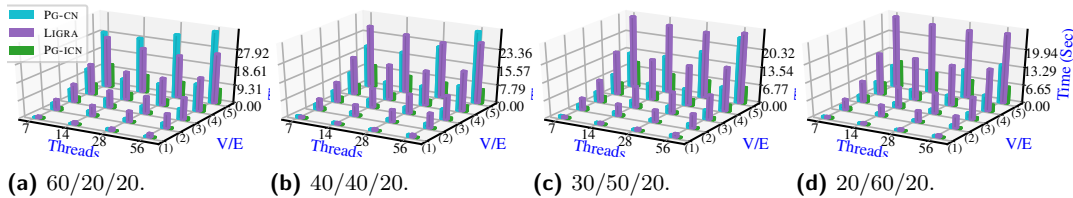
**GraphOne vs PANIGRAHAM.** GraphOne [34] does not allow vertex updates. Unlike Stinger and Ligra, wherein copying the allocated graph structure to a new memory-location was a workaround, the GraphOne interface does not let the structure of the allocated graph be retrieved, thus disallowing any obvious possibility of PUTV and REMV operations. Thus, the only experimental comparison of PG-Cn and PG-Icn with GraphOne was in regards to concurrent executions of BFS, PUTE and REME operations by fixing the number of vertices. Figure 4 depicts different workloads and with a graph having a fixed number of vertices. We observe that for the chosen graphs sizes and the common workload, that well represent a dynamic size, GraphOne exhibits severely limited scalability with the number of threads, in comparison to PG-Cn and PG-Icn.



■ **Figure 6** Consistent concurrent analytics of PG-Cn against Stinger: the execution latency of BFS is plotted for 56 threads for different graph-sizes as labeled on x-axis:  $\{(V/E), (1) : 1K/10K, (2) : 8K/80K, (3) : 16K/160K, (4) : 32K/320K, (5) : 65K/500K\}$ .

**Memory footprint.** Figure 5 shows that Stinger has approximately 80x heavier memory footprint in comparison with PG-Cn or PG-Icn for executions with BFS queries. The reason can be traced in the design of Stinger, whereby it pessimistically allocates a large chunk of memory. For SSSP and BC queries, wherein the `OpItem` object gets bigger to facilitate partial snapshot collection, Ligra gets advantage of compact CSR representation. However, in no case the allocated memory by PG-Cn or PG-Icn spills as drastically as Stinger.

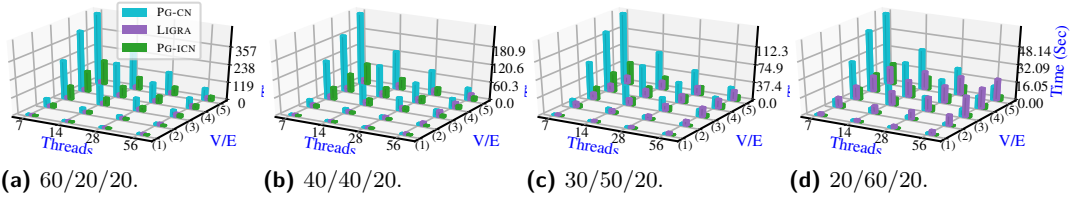
**Concurrency vs. Batch analytics.** Figure 6 shows that PG-Cn offers two to four orders of magnitude speed-up in comparison with state-of-the-art Stinger for a given standard system setting. It clearly implies that a concurrent analytics framework can vastly improve on the existing methods of batch analytics.



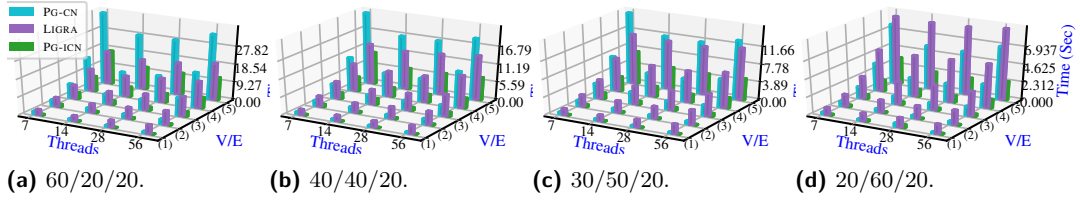
■ **Figure 7** Latency of the executions containing OP: BFS. In the 3D-plots, z-axis indicates the total time in seconds for an end-to-end run of  $10^4$  operations uniformly distributed according to the respective distributions. The dataset sizes as labeled on the y-axis are  $\{(V/E), \{(1) : 1K/10K, (2) : 8K/80K, (3) : 16K/160K, (4) : 32K/320K, (5) : 65K/500K\}$ .

**Overall advantage of Concurrency.** Having seen the comparative performance of Stinger-based batch analytics and the proposed consistent concurrent analytics framework, now we compare both the consistent and high performing inconsistent variants of PANIGRAHAM with a lightweight static framework Ligra adopted to the batch analytics setting (as noted earlier, Stinger and GraphOne do not support SSSP and BC queries). See Figures 7, 8, and 9. For smaller datasets, as well as for higher update workloads, both PG-Cn and PG-Icn outperform Ligra. As the query workload grows i.e., the overall workload gets closer to static, CSR based method exploits inline parallelization with lower cache misses, thus Ligra gets advantage. Still, PG-Icn decisively performs better than Ligra over the entire range of graph sizes and workload distribution. Notice that, in some cases, as we move from 28 to 56 threads, hyperthreading activates leading to cache thrashing, which limits CSR's optimization; in the same way, in some cases, for higher thread contention PG-Cn's performance also suffers.

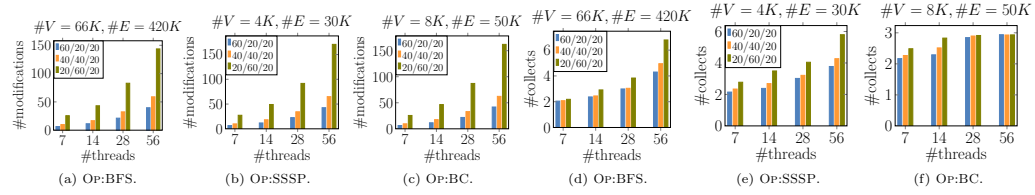
## 20:14 Non-Blocking Dynamic Unbounded Graphs



■ **Figure 8** End-to-end latency of the executions containing OP: SSSP. The plotting description is similar to that of Fig. 7. The graph sizes as labeled on the y-axis are  $\{(V/E), (1) : 1K/10K, (2) : 4K/30K, (3) : 8K/50K, (4) : 8K/70K, (5) : 8K/80K\}$ .



■ **Figure 9** End-to-end latency of the executions containing OP: BC. The plotting description is similar to that of Fig. 7. The graph sizes as labeled on the y-axis are  $\{(V/E), (1) : 1K/10K, (2) : 2K/20K, (3) : 4K/40K, (4) : 8K/80K, (5) : 16K/120K\}$ .



■ **Figure 10** Average number of concurrent modifications and scans during a query. Legends 60/20/20, etc. are identical to those in Fig. 7, 8 and 9.

## 5 Complexity Analysis

Given a graph  $G = (V, E)$ , denote  $|V| = n$ ,  $|E| = m$ ,  $\delta = \max_{v \in V} (\delta_v)$ , where  $\delta_v$  is the degree of vertex  $v$ . As PANIGRAHAM (PG) consists of a hash-table and BSTs, Ligra uses CSR format, and Stinger uses edge-lists to represent  $G$ , the worst-case cost of operations by each of them in a static setting are given in Table 1.

■ **Table 1** The static worst-case complexities.

Algo.	PUTV	REMV	PUTE	REME	GETV	GETE	BFS	SSSP	BC
PG	$O(n)$	$O(n)$	$O(n + \delta)$	$O(n + \delta)$	$O(n)$	$O(n + \delta)$	$O(n + m)$	$O(mn)$	$O(mn + n^2)$
Ligra	$O(n + m)$	$O(n + m)$	$O(m)$	$O(m)$	$O(1)$	$O(\log \delta)$	$O(n + m)$	$O(mn)$	$O(mn + n^2)$
Stinger	$O(n + m)$	$O(\delta)$	$O(\delta)$	$O(\delta)$	$O(1)$	$O(\delta)$	$O(n + m)$	–	–

The worst-case cost of PUTV/REMV/GETV for PG is due to the hash-table, and that of PUTE/REME/GETE is due to the BST and hash-table. The worst-case cost of PUTV/REMV for Ligra comes from copying and shifting the entire structure, whereas, that for PUTE/REME

comes from shifting the edge array. GETV and GETE in Ligra relate to lookup in an index and a sorted array, respectively. Stinger behaves similar to Ligra in terms vertex operations, however, the edge-lists facilitate the worst-case linear cost in maximum degree  $\delta$  only for the operations REMV/PUTE/REME/GETE here. The queries in each of the data-structure designs behave identically.

We define the *state* of a graph  $G$  as a tuple  $S_G = (n, m, \delta)$ , where  $n, m, \delta$  are as aforementioned. Essentially,  $S_G$  captures the size and shape of  $G$ . Now consider an execution – set of operation calls –  $X$  such that invocations and responses of *operations*  $\{o \in X\}$  form a valid *history*  $H$  [25]. Thus, for an  $o \in X$ ,  $type(o) \in \mathcal{A}$ , where  $type(o)$  denotes the type of  $o$  and  $\mathcal{A}$  is the ADT as described earlier. Denote the worst-case cost of  $o$ , given  $o$  is invoked at an atomic time point when state of  $G$  was  $S_G$  by  $W_{o, S_G}$ . The states of  $G$ , being tuples, are ordered by dictionary order. In a dynamic setting,  $W_{o, S_G}$  is upper-bounded by the cost of  $o$  as performed in a static setting over the *worst-case state*, during the lifetime of  $o$ , of  $G$  as defined in Lemma 1.

Let  $I_o$  and  $C_o$  be the *interval contention* [2] and *point contention* [3], respectively<sup>e</sup>, for an  $o \in X$ . We name the execution cases – PG-Cn, PG-Icn, Ligra, and Stinger – as in section 4. Notice that, for an execution of Ligra and Stinger,  $I_o = C_o = 1$  as there is no concurrency. For PG-Icn,  $C_o = 1$  as even though the operations are performed concurrently, they are essentially not obliged to maintain any consistency, thus, do not cause “restart” to their peers though they may cause “cost escalation” which is captured by  $I_o$ . Denote  $\tilde{I}_o = (I_o - 1)$ , the total number of concurrent operation calls *other than  $o$  itself* (those responsible for a possible cost escalation) that were invoked between the invocation and response of  $o$ . Lemma 1 is immediate:

► **Lemma 1.** *If an operation call  $o \in X$  is invoked at a state  $S_{G,o} = (n, m, \delta)$  of  $G$ , the worst-case state of  $G$  that  $o$  can encounter is  $\overline{S_{G,o}} = (O(n + \tilde{I}_o), O(m + \tilde{I}_o), O(\delta + \tilde{I}_o))$ .*

Denote  $X_{\mathcal{U}} = \{o \in X \mid type(o) \in \mathcal{U}, \mathcal{U} \subseteq \mathcal{A}\}$ , where  $\mathcal{A}$  is the ADT as defined in Section 2. Let  $I_{o, \mathcal{U}}$  and  $C_{o, \mathcal{U}}$  denote the interval and point contentions, respectively, of  $o$  pertaining to the operation calls  $o \in \{X_{\mathcal{U}} \cup \{o\}\}$ . Without loss of generality, we consider executions  $X$ , s.t.  $type(o) \in \mathcal{M} \cup \{q\} \forall o \in X$ , where  $\mathcal{M} = \{\text{PUTV}, \text{REMV}, \text{PUTE}, \text{REME}\} \subset \mathcal{A}$ ,  $q \in \mathcal{Q}$  and  $\mathcal{Q} = \{\text{BFS}, \text{SSSP}, \text{BC}\} \subset \mathcal{A}$ . This execution represents our experiments in section 4. The worst-case cost  $W_{o, S_{G,o}}$  of operation calls belonging to different  $type(o)$ , in a static setting, are as listed in Table 1. Denote  $\mathcal{M}_V = \{\text{PUTV}, \text{REMV}\}$ ,  $\mathcal{M}_E = \{\text{PUTE}, \text{REME}\}$ , and  $\delta_o$  as the degree of vertex  $v_1$ , s.t.  $o \in \{\text{PUTE}(v_1, v_2|w), \text{REME}(v_1, v_2)\}$ . Using the fact that an operation  $o \in X$  s.t.  $type(o) \in \mathcal{M}_V$  can be obstructed by only an operation  $o' \in E$  s.t.  $type(o') \in \mathcal{M}_V$  and similarly for the set  $\mathcal{M}_E$ , following the standard accounting method [13] an amortization over the update operations  $X_{\mathcal{M}}$  gives Lemma 2.

► **Lemma 2.** *The worst-case amortized cost per operation for an execution of  $X_{\mathcal{M}}$ , denoted as  $A_{X_{\mathcal{M}}}$  is*

$$O \left( \frac{C_{o, \mathcal{M}_V}}{|X_{\mathcal{M}}|} \sum_{o \in X_{\mathcal{M}_V}} W_{o, \overline{S_{G,o}}} + \frac{1}{|X_{\mathcal{M}}|} \sum_{o \in X_{\mathcal{M}_E}} W_{o, \overline{S_{G,o}}} + \frac{C_{o, \mathcal{M}_E}}{|X_{\mathcal{M}}|} \sum_{o \in X_{\mathcal{M}_E}} O(\delta_e) \right).$$

<sup>e</sup> We are slightly adapting the original definitions of interval and point contention, where these notions are defined for processes invoking  $o$ , to our terminologies.

## 20:16 Non-Blocking Dynamic Unbounded Graphs

Notice that the worst-case costs for individual operation calls  $W_{o, \overline{S_{G,o}}}$  consider a dynamic setting. Lemma 2 essentially infers that a careful accounting for amortization should consider only those concurrent operations that cause a CAS failure and thereby restart of an  $o \in X$ . Furthermore, an  $o \in X_{\mathcal{M}_E}$  restarts only from the vertex  $v_1$  as mentioned above. Using a similar technique, and the fact that the queries by PG-Icn do not restart, we have Theorem 3

► **Theorem 3.** *Denote*

$$A_X(\mathcal{M}) = \frac{C_{o, \mathcal{M}_V}}{|E|} \sum_{o \in X_{\mathcal{M}_V}} W_{o, \overline{S_{G,o}}} + \frac{1}{|X|} \sum_{o \in X_{\mathcal{M}_E}} W_{o, \overline{S_{G,o}}} + \frac{C_{o, \mathcal{M}_E}}{|E|} \sum_{o \in X_{\mathcal{M}_E}} O(\delta_e). \quad (1)$$

The worst-case amortized cost per operation  $A_X$  for an execution of  $o$  s.t.  $\text{type}(o) \in \mathcal{M} \cup \{q\} \forall o \in X$ , and  $q \in \mathcal{Q} = \{BFS, SSSP, BC\}$  is

1. For  $q \in \mathcal{Q}$  performed by PG-Icn,

$$A_X = A_X(\mathcal{M}) + \frac{1}{|X|} \sum_{o \in X_{\mathcal{Q}}} \left( W_{o, \overline{S_{G,o}}} + \widetilde{I_{o, \mathcal{M}}} \right). \quad (2)$$

2. For  $q \in \mathcal{Q}$  performed by PG-Cn,

$$A_X = A_X(\mathcal{M}) + \frac{C_{o, \mathcal{M}}}{|X|} \sum_{o \in X_{\mathcal{Q}}} \left( W_{o, \overline{S_{G,o}}} + \widetilde{I_{o, \mathcal{M}}} \right). \quad (3)$$

Now, different from the *concurrent analytics* by PANIGRAHAM, in a batch analytics setting, the updates and queries selected at a random order are essentially performed sequentially, thus we have Theorem 4.

► **Theorem 4.** *For  $q \in \mathcal{Q}$  performed by Ligra or Stinger is  $O\left(\frac{1}{|X|} \sum_{o \in X} W_{o, S_{G,o}}\right)$ .*

Plugging in the worst-case costs from Table 1 gives the amortized costs in terms of the parameters  $n, m, \delta$  of  $G$ .

► **Remark 5 (Observed contention).** Notice that the worst-case amortized cost per operation for PG-Cn can be tightened by more careful accounting as one restart of a query execution  $o \in X_{\mathcal{Q}}$  corresponds to all the modifications in  $G$  that might have happened during its scan phase. We experimentally obtained the average number of concurrent modifications and scans as in TREECOLLECT for the queries as shown in Figure 10. Clearly, the average number of scans before a linearized response is much less than the average number of concurrent modifications during the lifetime of a query.

► **Remark 6 (Parallel speedup).** Assuming that there are  $p$  non-faulty threads in the shared-memory system, and each atomic step can be executed in a unit *time-step*, the worst-case amortized number of time-steps per operation for an execution of both PG-Cn and PG-Icn is roughly  $A_X/p$ , where  $A_X$  is as given in Theorem 3. For Ligra and Stinger, the operation calls  $o \in X_{\mathcal{Q}}$  get speedup due to parallel executions, whereas  $o \in X_{\mathcal{M}}$  are executed sequentially. If the parallel execution of an  $o \in X_{\mathcal{Q}}$  has a speedup  $s \leq p$ , then the worst-case amortized number of time-steps per operation for an execution of Ligra or Stinger will be  $O\left(\frac{1}{|X|} \sum_{o \in X_{\mathcal{M}}} W_{o, S_{G,o}} + \frac{1}{s|X|} \sum_{o \in X_{\mathcal{Q}}} W_{o, S_{G,o}}\right)$ . Clearly, the theoretical insights from the amortized analysis is corroborated by our experiments where we observed that even for a moderately sized graph, PG-Icn performs better than Ligra, whereas, for smaller graphs despite of costly consistent queries, PG-Cn outperforms the batch analytic methods.

## 6 Conclusion

In this paper, we presented a novel framework of concurrent dynamic graph analytics **PANIGRAHAM**. We implemented commonly used graph algorithms: breadth-first search, single-source-shortest-path, and betweenness centrality over this framework. The presented framework is versatile enough such that it can be extended to other graph algorithms that process the *global* information in a graph and are usually found in graph-based analytics. We proved that the presented algorithms are non-blocking and linearizable. From the perspective of higher performance at the cost of consistency, we presented an inconsistent variant as well. We extensively evaluated a C++ implementation of the algorithms that shows scalability of the method with parallel resources. Another important contribution of this paper is an amortized analysis of the graph operations in a concurrent consistent non-blocking setting. To the best of our knowledge, this is the first work to provide amortized upper bound for concurrent dynamic graph operations. Unlike the well-known parallel batch analytics libraries, our framework honors the real-time order of updates and most significantly provides fully dynamic vertex additions, which has largely been unavailable previously. Its memory footprint is up to 80x lighter compared to Stinger and it provides up-to-three orders of magnitude better performance than Stinger.

The present work motivates two very important future works: (a) implementing lock-free variant of CSR representation of graphs to take advantage of cache efficiency and concurrency, and, (b) an amortized average-case analysis of these algorithm, which gives a more realistic picture of the implementations with respect to their theoretical behavior.

---

## References

- 1 Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic Snapshots of Shared Memory. *Journal of the ACM*, 40(4):873–890, 1993.
- 2 Yehuda Afek, Gideon Stupp, and Dan Touitou. Long Lived Adaptive Splitter and Applications. *Distributed Comput.*, 15(2):67–86, 2002.
- 3 H. Attiya and A. Fouren. Alg. Adapting to Point Contention. *J. ACM*, 50(4):444–468, 2003.
- 4 Greg Barnes. A Method for Implementing Lock-free Shared-data Structures. In *SPAA*, pages 261–270, 1993.
- 5 Omar Batarfi, Radwa El Shawi, Ayman G Fayoumi, Reza Nouri, Ahmed Barnawi, Sherif Sakr, et al. Large Scale Graph Processing Systems: Survey and an Experimental Evaluation. *Cluster Computing*, 18(3):1189–1213, 2015.
- 6 Trevor Brown, Faith Ellen, and Eric Ruppert. A general technique for non-blocking trees. In *PPoPP*, pages 329–342, 2014.
- 7 A. Buluç, J. R. Gilbert, and V. B. Shah. Implementing Sparse Matrices for Graph Algorithms. In *Graph Algorithms in the Language of Linear Algebra*, volume 22, pages 287–313. SIAM, 2011.
- 8 Hung Cao, Monica Wachowicz, and Sangwhan Cha. Developing an edge computing platform for real-time descriptive analytics. *2017 IEEE International Conference on Big Data (Big Data)*, pages 4546–4554, 2017.
- 9 Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-MAT: A recursive model for graph mining. In *SDM*, pages 442–446, 2004.
- 10 Bapi Chatterjee, Nhan Nguyen, and Philippas Tsigas. Efficient Lock-free Binary Search Trees. In *PODC*, pages 322–331, 2014.
- 11 Bapi Chatterjee, Sathya Peri, and Muktikanta Sa. Dynamic Graph Operations: A Consistent Non-blocking Approach. *CoRR*, abs/2003.01697, 2020.

- 12 Bapi Chatterjee, Sathya Peri, Muktikanta Sa, and Nandini Singhal. A Simple and Practical Concurrent Non-blocking Unbounded Graph with Linearizable Reachability Queries. In *ICDCN*, pages 168–177, 2019.
- 13 Bapi Chatterjee, Ivan Walulya, and Philippos Tsigas. Help-optimal and Language-portable Lock-free Concurrent Data Structures. In *ICPP*, pages 360–369, 2016.
- 14 Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. Kineograph: taking the pulse of a fast-changing and connected world. In *EuroSys*, pages 85–98, 2012.
- 15 Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.
- 16 Nhan Nguyen Dang and Philippos Tsigas. Progress guarantees when composing lock-free objects. In *Euro-Par*, pages 148–159, 2011.
- 17 Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. Low-Latency Graph Streaming Using Compressed Purely-Functional Trees. In *PLDI*, pages 918–934, 2019.
- 18 D. Ediger, R. McColl, J. Riedy, and D. A. Bader. STINGER: High Performance Data Structure for Streaming Graphs. In *HPEC*, 2012.
- 19 Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *PODC*, pages 131–140, 2010.
- 20 Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30, 2012.
- 21 Timothy L. Harris. A Pragmatic Implementation of Non-blocking Linked-Lists. In *DISC*, pages 300–314, 2001.
- 22 Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer III, and Nir Shavit. A Lazy Concurrent List-Based Set Algorithm. *Parallel Processing Letters*, 17(4):411–424, 2007.
- 23 Danny Hendler, Nir Shavit, and Lena Yerushalmi. A Scalable Lock-free Stack Algorithm. In *SPAA*, pages 206–215, 2004.
- 24 Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-Free Synchronization: Double-Ended Queues as an Example. In *(ICDCS)*, pages 522–529, 2003.
- 25 Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan K., 2008.
- 26 Maurice Herlihy and Nir Shavit. On the Nature of Progress. In *OPODIS*, pages 313–328, 2011.
- 27 Maurice Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.
- 28 Shane V. Howley and Jeremy Jones. A Non-blocking Internal Binary Search Tree. In *SPAA*, pages 161–171, 2012.
- 29 Wole Jaiyeoba and K. Skadron. Graphtinker: A high performance data structure for dynamic graph processing. *IPDPS*, pages 1030–1041, 2019.
- 30 Nikolaos D. Kallimanis and Eleni Kanellou. Wait-Free Concurrent Graph Objects with Dynamic Traversals. In *OPODIS*, pages 1–27, 2015.
- 31 Miray Kas, Kathleen M. Carley, and L. Richard Carley. Incremental Closeness Centrality for Dynamically Changing Social Networks. In *ASONAM 2013*, pages 1250–1258, 2013.
- 32 Alex Kogan and Erez Petrank. Wait-Free Queues With Multiple Enqueuers and Dequeuers. In *PPOPP*, pages 223–234, 2011.
- 33 Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic Parallelism Requires Abstractions. In *PLDI*, pages 211–222, 2007.
- 34 P. Kumar and H. Huang. Graphone: A data store for real-time analytics on evolving graphs. In *FAST*, 2019.
- 35 Edya Ladan-Mozes and Nir Shavit. An Optimistic Approach to Lock-free FIFO Queues. *Distributed Computing*, 20(5):323–341, 2008.
- 36 Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford Large Network Dataset Collection. <http://snap.stanford.edu/data>, June 2014.



- 37 Yujie Liu, Kunlong Zhang, and Michael Spear. Dynamic-sized Nonblocking Hash Tables. In *PODC*, pages 242–251, 2014.
- 38 Maged M. Michael. High Performance Dynamic Lock-Free Hash Tables and List-Based Sets. In *SPAA*, pages 73–82, 2002.
- 39 Aravind Natarajan and Neeraj Mittal. Fast concurrent lock-free binary search trees. In *PPoPP*, pages 317–328, 2014.
- 40 K A Obukhova, Iryna Zhuravska, and Volodymyr Burenko. Diagnostics of power consumption of a mobile device multi-core processor with detail of each core utilization. *TCSET*, pages 368–372, 2020.
- 41 P. Pan, C. Li, and M. Guo. Congraplus: Towards efficient processing of concurrent graph queries on numa machines. *IEEE TPDS*, 30(9):1990–2002, 2019.
- 42 Peitian Pan and Chao Li. Congra: Towards efficient processing of concurrent graph queries on shared-memory machines. In *ICCD*, pages 217–224, 2017.
- 43 Arunmoezhi Ramachandran and Neeraj Mittal. A Fast Lock-Free Internal Binary Search Tree. In *ICDCN*, pages 37:1–37:10, 2015.
- 44 Alberto G. Rossi, D. Blake, A. Timmermann, I. Tonks, and R. Wermers. Network centrality and delegated investment performance. *Journal of Financial Economics*, 128:183–206, 2018.
- 45 Julian Shun and Guy E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *PPoPP*, pages 135–146, 2013.
- 46 Julian Shun, Laxman Dhulipala, and Guy E. Blelloch. Smaller and Faster: Parallel Processing of Compressed Graphs with Ligra+. In *DCC*, pages 403–412, 2015.
- 47 Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.
- 48 Shahar Timnat, Anastasia Braginsky, Alex Kogan, and Erez Petrank. Wait-Free Linked-Lists. In *OPODIS*, pages 330–344, 2012.
- 49 John D. Valois. Lock-Free Linked Lists Using Compare-and-Swap. In *PODC*, pages 214–222, 1995.
- 50 Kunlong Zhang, Yujiao Zhao, Yajun Yang, Yujie Liu, and Michael F. Spear. Practical Non-blocking Unordered Lists. In *DISC*, pages 239–253, 2013.
- 51 J. Zhao, Y. Zhang, X. Liao, L. He, B. He, H. Jin, H. Liu, and Y. Chen. Graphm: An efficient storage system for high throughput of concurrent graph processing. In *SC*, 2019.

## **A** The Non-blocking Graph Algorithm

In this section, we present a detailed implementation of our non-blocking directed graph algorithm. The non-blocking graph composes on the basic structures of the dynamic non-blocking hash table [37] and non-blocking internal binary search tree [28]. For a self-contained reading, we present the algorithms of non-blocking hash table and BST. Because it derives and builds on the earlier works [37] and [28], many keywords in our presentation are identical to theirs. One key difference between our non-blocking BST design from [28] is that we maintain a mutable edge-weight in each BST node, thereby not only the implementation requires extra steps but also we need to discuss extra cases in order to argue the correctness of our design. Furthermore, we also perform non-recursive traversals in the BST for snapshot collections, which were already discussed as part of the graph queries. The pseudo-codes pertaining to the non-blocking hash-table are presented in Figure 12, whereas those for the non-blocking BST are presented in Figures 13.

### **A.1** Structures

The declarations of the object structures that we use to build the data structure are listed in Figure 12 and 13. The structures `FSet`, `FSetOp`, and `HNode` are used to build the *vertex-list*, whereas `Node`, `RelocateOp`, and `ChildCASOp` are the component-objects of the *edge-list*. The

```

1: Operation PUTV(v)
2:   return HASHADD(v);
-----
3: Operation REMV(v)
4:   return HASHREM(v);
-----
5: Operation GETV(v)
6:    $\langle st, v \rangle \leftarrow \text{HASHCON}(v)$ ;
7:   if (st = true) then
8:     return  $\langle \text{true}, v \rangle$ ;
9:   else
10:    return  $\langle \text{false}, \text{NULL} \rangle$ ;
-----
11: Operation GETE(v1, v2)
12:    $\langle u, v, st \rangle \leftarrow \text{CONVPLUS}(v_1, v_2)$ ;
13:   if (st = false) then
14:     return  $\langle \text{false}, \infty \rangle$ ;
15:    $\langle st, e \rangle \leftarrow \text{BSTCON}(v_2, v.\text{enxt})$ ;
16:   if (st = FOUND  $\wedge \neg \text{HASHCON}(v_1) \wedge \neg \text{HASHCON}(v_2)$ 
17:    $\wedge (e.\text{ptv} = u)$ ) then
18:     z  $\leftarrow e.w$ ;
19:     return  $\langle \text{true}, z \rangle$ ;
20:   else return  $\langle \text{false}, \infty \rangle$ ;
-----
21: Method CONVPLUS(v1, v2)
22:    $\langle st1, u \rangle \leftarrow \text{HASHCON}(v_1)$ ; //Modified GETV, re-
23:   returns status along with ref
24:    $\langle st2, v \rangle \leftarrow \text{HASHCON}(v_2)$ ;
25:   if (st1 = true  $\wedge$  st2 = true) then
26:     return  $\langle u, v, \text{true} \rangle$ ;
27:   else return  $\langle u, u, \text{false} \rangle$ ;
-----
28: Operation PUTE(v1, v2|w)
29:    $\langle u, v, st \rangle \leftarrow \text{CONVPLUS}(v_1, v_2)$ ;
30:   if (st = false) then return  $\langle \text{false}, \infty \rangle$ ;
31:   while (true) do
32:     if ( ISMRKD(u)  $\vee$  ISMRKD(v)) then
33:       return  $\langle \text{false}, \infty \rangle$ ;
34:     st  $\leftarrow \text{FIND}(v_2, pe, peOp, ce, ceOp, u.\text{enxt})$ ;
35:     if (GETFLAG(pe) = MARKED) then
36:       continue;
37:     if (st = FOUND) then
38:       if (ce.w = w) then return  $\langle \text{false}, w \rangle$ ;
39:       else
40:         z  $\leftarrow ce.w$ ;
41:         CAS(ce.w, z, w);
42:         u.ecnt.FetchAndAdd (1);
43:         return  $\langle \text{true}, z \rangle$ ;
-----
44:   else
45:     ne  $\leftarrow \text{CENODE}(v_2, w)$ ;
46:     ne.ptv  $\leftarrow v$ ;
47:     Boolean ifLeft  $\leftarrow (st = \text{NOTFOUND\_L})$ ;
48:     ENode old  $\leftarrow \text{ifLeft} ? ce.\text{left} : ce.\text{right}$ ;
49:     casOp  $\leftarrow \text{new ChildCASOp}(\text{ifLeft}, \text{old}, ne)$ ;
50:     if (CAS(ce.op, ceOp, FLAG(casOp, CHILDCAS)))
51:     then
52:       u.ecnt.FetchAndAdd (1);
53:       HELPCAS(casOp, ce);
54:       return  $\langle \text{true}, \infty \rangle$ ;
-----
55: Operation REME(v1, v2)
56:    $\langle u, v, st \rangle \leftarrow \text{CONVPLUS}(v_1, v_2)$ ;
57:   if (st = false) then
58:     return  $\langle \text{false}, \infty \rangle$ ;
59:   while (true) do
60:     if (ISMRKD(u)  $\vee$  ISMRKD(v)) then
61:       return  $\langle \text{false}, \infty \rangle$ ;
62:     st  $\leftarrow \text{FIND}(v_2, pe, peOp, ce, ceOp, u.\text{enxt})$ ;
63:     if (st  $\neq$  FOUND) then
64:       return  $\langle \text{false}, \infty \rangle$ ;
65:     if (ISNULL(curr.right)  $\vee$  ISNULL(ce.left))
66:     then
67:       if (CAS(ce.op, ceOp, FLAG(ceOp, MARKED)))
68:       then
69:         u.ecnt.FetchAndAdd (1);
70:         HELPMARKED(pe, peOp, ce);
71:         z  $\leftarrow ce.w$ ;
72:         break;
73:     else
74:       st  $\leftarrow \text{FIND}(v_2, pe, peOp, ce, ceOp, u.\text{enxt})$ ;
75:       if ((st = ABORT)  $\vee$  (ce.op  $\neq$  ceOp)) then
76:         continue;
77:       relocOp  $\leftarrow \text{new RelocateOp}(ce, ceOp, v_2,$ 
78:       replace.e);
79:       if (CAS(relocOp, replaceOp, FLAG(relocOp,
80:       RELOCATE))) then
81:         u.ecnt.FetchAndAdd (1);
82:         if (HELPRELOCATE(relocOp, pe, peOp, replace))
83:         then
84:           z  $\leftarrow ce.w$ ;
85:           break;
86:         return  $\langle \text{true}, z \rangle$ ;

```

■ **Figure 11** Pseudocodes of PUTV, REMV, GETV, PUTE, REME, GETE and CONVPLUS.

structure **FSet**, a freezable set of **VNodes** that serves as a building block of the non-blocking hash table. An **FSet** object builds a **VNode** set with **PUTV**, **REMV** and **GETV** operations, and in addition, provides a **FREEZE** method that makes the object immutable. The changes of an **FSet** object can be either addition or removal of a **VNode**. For simplicity, we encode **PUTV** and **REMV** operation as **FSetOp** objects. The **FSetOp** has a state *optype* (**PUTV** or **REMV**), the key value, *done* a boolean field that shows the operation was applied or not, and *resp* a boolean field that holds the return value.

The vertex-list is a dynamically resizable non-blocking hash table constructed with the instances of **VNodes**, and it is a linked-list of **HNodes** (Hash Table Node). The **HNode** composed of an array of buckets of **FSet** objects, the *size* field stores the array length and the predecessor **HNode** is pointed to by the *pred* pointer. The head of the **HNode** is pointed to by a shared **Head** pointer.

For clarity, we assume that a **RESIZE** method grows (doubles) or shrinks (halves) the *size* of the **HNode** which amount to modifying the length of the bucket array. The hash function uses modular arithmetic for indexing in the hash table, e.g.  $\text{index} = \text{key} \bmod \text{size}$ .

Based on the boolean parameter taken by `RESIZE` method, it decides the hash table either to grow or shrink. The `INITBKT` method ensures all `VNodes` are physically present in the buckets. It relocates the `HNodes` to the hash table which are in the predecessor's list.

The  $i^{th}$  bucket of a given `HNode`  $h$  is initialized by `INITBKT` method, by splitting or merging the buckets of  $h$ 's predecessor `HNode`  $s$ , if  $s$  exists. The sizes of  $h$  and  $s$  are compared and then this method decides whether  $h$  is shrinking or expanding with reference to  $s$ . Then it freezes the respective bucket(s) of  $s$  before copying the `VNodes`. If  $h$  halves the size of  $s$ , then  $i^{th}$  and  $(i + h.size)^{th}$  buckets of  $s$  are merged together to form the  $i^{th}$  bucket of  $h$ . Otherwise,  $h$  doubles the size of  $s$ , then approximately half of the `VNodes` in the  $(i \bmod h.size)^{th}$  bucket of  $s$  relocate to the  $i^{th}$  bucket of  $h$ . To avoid any races with the other helping threads while splitting or merging of buckets a `CAS` is used (Line 137).

The `ENode` structure is similar to that of a lock-free BST [28] with an additional edge weight `w` and a pointer field `ptv` which points to the corresponding `VNode`. This helps direct access to its `VNode` while doing a BFS traversal and also helps in deletion of the incoming edges. The operation `op` field stores if any changes are being made, which affects the `ENode`. To avoid the overhead of another field in the node structure, we use bit-manipulation: last significant bits of a pointer  $p$ , which are unused because of the memory-alignment of the shared-memory system, are used to store information about the state of the pointer shared by concurrent threads and executing an operation that would potentially update the pointer of the pointer. More specifically, in case of an x86-64 bit architecture, memory has a 64-bit boundary and the last three least significant bits are unused. So, we use the last two significant bits, which are enough for our purpose, of the pointer to store auxiliary data. We define four different methods to change an `ENode` pointer: `ISNULL( $p$ )` returns `true` if the last two significant bits of  $p$  make 00, which indicates no ongoing operation, otherwise, it returns `false`; `ISMRKD( $p$ )` returns `true` if the last two significant bits of  $p$  are set to 01, else it returns `false`, which indicates the node is no longer in the tree and it should be *physically* deleted; `ISCHILDCAS( $p$ )` returns `true` if last two bits of  $p$  are set to 10, which indicates one of the child node is being modified, else it returns `false`; `ISRELOCATE( $p$ )` returns `true` if the last two bits of  $p$  make 11, which indicates that the `ENode` is undergoing a node relocation operation.

A `ChildCASOp` object holds sufficient information for another thread to finish an operation that made changes to one of the child – right or left – pointers of a node. A node's `op` field holds a flag indicating an active `ChildCASOp` operation. Similarly, a `RelocateOp` object holds sufficient information for another thread to finish an operation that removes the key of a node with both the children and replaces it with the next largest key. To replace the next largest key, we need the pointer to the node whose key is to be removed, the data stored in the node's `op` field, the key to replacement and the key being removed. As we did in case of a `ChildCASOp`, the `op` field of a node holds a flag with a `RELOCATE` state indicating an active `RelocateOp` operation.

## A.2 The Vertex Operations

The working of the non-blocking vertex operations `PUTV`, `REMV`, and `GETV` are presented in Figure 11. A `PUTV( $v$ )` operation, at Lines 1 to 2, invokes `HASHADD( $v$ )` to perform an insertion of a `VNode`  $v$  in the hash table. A `REMV( $v$ )` operation at lines 3 to 4 invokes `HASHREM( $v$ )` to perform a deletion of `VNode`  $v$  from the hash table. The method `APPLY`, which tries to modify the corresponding buckets, is called by both `HASHADD` and `HASHREM`, see Line 109 and 114. It first creates a new `FSetOp` object consisting of the modification request, and then constantly tries to apply the request to the respective bucket  $b$ , see Lines

138 to 146. Before applying the changes to the bucket it checks whether  $b$  is `NULL`; if it is, `INITBKT` method is invoked to initialize the bucket (Line 144). At the end, the return value is stored in the `resp` field.

The algorithm and the resizing hash table are orthogonal to each other, so we used heuristic policies to resize the hash table. As a classical heuristic we use a `HASHADD` operation that checks for the size of the hash table with some *threshold* value, if it exceeds the threshold the size of the table is doubled. Similarly, a `HASHREM` checks the threshold value, if it falls below threshold, it shrinks the hash table size to halves.

A `GETV(v)` operation, at Lines 5 to 10, invokes `HASHCON(v)` to search a `VNode`  $v$  in the hash table. It starts by searching the given key  $v$  in the bucket  $b$ . If  $b$  is `NULL`, it reads  $t$ 's predecessor (Line 122)  $s$  and then starts searching on it. At this point it could return an incorrect result as `HASHCON` is concurrently running with resizing of  $s$ . So, a double check at Line 123 is required to test whether  $s$  is `NULL` between Lines 120 and 122. Then, we re-read that bucket of  $t$  (Line 124 or 126), which must be initialized before  $s$  becomes `NULL`, and then we perform the search in that bucket. If  $b$  is not `NULL`, then we simply return the presence of the corresponding `VNode` in the bucket  $b$ . Note that, at any point in time there are at most two `HNodes`: only one when no resizing happens and another to support resizing – halving or doubling – of the hash table.

### A.3 The Edge Operations

The non-blocking graph edge operations – `PUTE`, `REME`, and `GETE` – are presented in Figure 11. Before describing these operations, we detail the implementation of `FIND` method, which is used by them. It is shown in Figure 13. The method `FIND`, at Lines 199 to 227, tries to locate the position of the key by traversing down the edge-list of a `VNode`. It returns the position in  $pe$  and  $ce$ , and their corresponding `op` values in  $peOp$  and  $ceOp$  respectively. The result of the method `FIND` can be one of the four values: (1) `FOUND`: if the key is present in the tree, (2) `NOTFOUND_L`: if the key is not in the tree but might have been placed at the left child of  $ce$  if it was added by some other threads, (3) `NOTFOUND_R`: similar to `NOTFOUND_L` but for the right child of  $ce$ , and (4) `ABORT`: if the search in a subtree is unable to return a usable result.

A `PUTE(v1, v2|w)` operation, at Lines 26 to 51, begins by validating the presence of  $v_1$  and  $v_2$  in the vertex-list. If the validations fails, it returns  $\langle \text{false}, \infty \rangle$  (Line 28). Once the validation succeeds, `PUTE` operation invokes `FIND` method in the edge-list of the vertex with key  $v_1$  to locate the position of the key  $v_2$ . The position is returned in the variables  $pe$  and  $ce$ , and their corresponding `op` values are stored in the  $peOp$  and  $ceOp$  respectively. On that, `PUTE` checks whether an `ENode` with the key  $v_2$  is present. If it is present containing the same edge weight value  $w$ , it implies that an edge with the exact same weight is already present, therefore `PUTE` returns  $\langle \text{false}, \infty \rangle$  (Line 36). However, if it is present with a different edge weight, say  $z$ , `PUTE` updates  $ce$ 's old weight  $z$  to the new weight  $w$  and returns  $\langle \text{true}, z \rangle$  (Line 38). We update the edge-weight using a `CAS` to ensure the correct return in case there were multiple concurrent `PUTE` operations trying to update the same edge. Notice that, here we are *not* freezing the `ENode` in anyway while updating its weight. The linearizability is still ensured, which we discuss in the next section.

If the key  $v_2$  is not present in the tree, a new `ENode` and a `ChildCASOp` object are created. Then using `CAS` the object is inserted logically into  $ce$ 's `op` field (Line 48). If the `CAS` succeeds, it implies that  $ce$ 's `op` field hadn't been modified since the first read. Which in turn indicates that all other fields of  $ce$  were also not changed by any other concurrent

thread. Hence, the CAS on one of the  $ce$ 's child pointer should not fail. Thereafter, using a call to `HELPCHILDCAS` method the new `ENode`  $ne$  is physically added to the tree. This can be done by any thread that sees the ongoing operation in  $ce$ 's `op` field.

A `REME( $v_1, v_2$ )` operation, at Lines 52 to 78, similarly begins by validating the presence of  $v_1$  and  $v_2$  in the vertex-list. If the validation fails, it returns  $\langle \text{false}, \infty \rangle$ . Once the validation succeeds, it invokes `FIND` method in the edge-list of the vertex having key  $v_1$  to locate the position of the key  $v_2$ . If the key is not present it returns  $\langle \text{false}, \infty \rangle$ . If the key is present, one of the two paths is followed. The first path at Lines 63 to 67 is followed if the node has less than two children. In case the node has both its children present a second path at Lines 69 to 77 is followed. The first path is relatively simpler to handle, as single CAS instruction is used to mark the node from the state `NONE` to `MARKED` at this point the node is considered as logically deleted from the tree. After a successful CAS, a `HELPMARKED` method is invoked to perform the physical deletion. It uses a `ChildCASOp` to replace  $pe$ 's child pointer to  $ce$ 's with either a pointer to  $ce$ 's only child pointer, or a `NULL` pointer if  $ce$  is a leaf node.

The second path is more difficult to handle, as the node has both the children. Firstly, `FIND` method only locates the children but an extra `FIND` (Line 69) method is invoked to locate the node with the next largest key. If the `FIND` method returns `ABORT`, which indicates that  $ce$ 's `op` field was modified after the first search, so the entire `REME` operation is restarted. After a successful search, a `RelocateOp` object  $replace$  is created (Line 72) to replace  $ce$ 's key  $v_2$  with the node returned. This operation added to  $replace$ 's `op` field safeguards it against a concurrent deletion while the `REME` operation is running by virtue of the use of a CAS (Line 73). Then `HELPRELOCATE` method is invoked to insert `RelocateOp` into the node with  $v_2$ 's `op` field. This is done using a CAS, after a successful CAS the node is considered as logically removed from the tree. Until the result of the operation is known the initial state is set to `ONGOING`. If any other thread either sees that the operation is completed by way of performing all the required CAS executions or takes steps to perform those CAS operations itself, it will set the operation state from `ONGOING` to `SUCCESSFUL`, using a CAS. If it has seen other value, it sets the operation state from `ONGOING` to `FAILED`. After the successful state change, a CAS is used to update the key to new value and a second CAS is used to delete the ongoing `RelocateOp` from the same node. Then next part of the `HELPRELOCATE` method performs cleanup on  $replace$  by either marking it if the relocation was successful or clearing its `op` field if it has failed. If the operation is successful and  $ce$  is marked, `HELPMARKED` method is invoked to excise  $ce$  from the tree. At the end `REME` returns  $\langle \text{true}, ce.w \rangle$

Similar to `PUTE` and `REME`, a `GETE( $v_1, v_2$ )` operation, at Lines 11 to 19, begins by validating the presence of  $v_1$  and  $v_2$  in the vertex-list. If the validation fail, it returns  $\langle \text{false}, \infty \rangle$ . Once the validation succeeds, it invokes `FIND` method in the edge-list of the vertex with key  $v_1$  to locate the position of the key  $v_2$ . If it finds  $v_2$ , it checks if both the vertices are not marked and also the  $ceOp$  not marked; on ensuring that it returns  $\langle \text{true}, ce.w \rangle$ , otherwise, it returns  $\langle \text{false}, \infty \rangle$ .

```

struct FSetNode {int set; boolean ok; }
struct FSet { FSetNode node; }
struct FSetOp {int optype, key; boolean resp; }
struct HNode {FSet buckets;int size;HNode pred;}

79: Method GETRESPONSE(op)
80: return op.resp;

81: Method HASMEMBER(b, k)
82: o ← b.node; // local copy of b
83: return k ∈ o.set;

84: Method INVOKE(b, op)
85: o ← b.node; // local copy of b
86: while (o.ok) do
87:   if (op.optype = ADD) then
88:     resp ← op.key ∉ o.set;
89:     set ← o.set ∪ {op.key};
90:   else
91:     if (op.optype = REMOVE) then
92:       resp ← op.key ∈ o.set;
93:       set ← o.set \ {op.key};
94:     n ← new FSetNode(set, true);
95:     if (CAS (b.node, o, n)); then
96:       op.resp ← resp;
97:       return true;
98:     o ← b.node;
99: return false;

100: Method FREEZE(b)
101: o ← b.node; // local copy of b
102: while (o.ok) do
103:   n ← new FSetNode(o.set, false);
104:   if (CAS (b.node, o, n)); then
105:     break;
106:   o ← b.node;
107: return o.set

108: Operation HASHADD(key)
109: resp ← APPLY(ADD, key); ]

110: if (heuristic-policy) then
111:   RESIZE (true);
112: return resp;

113: Operation HASHREM(key)
114: resp ← APPLY(REMOVE, key);
115: if (heuristic-policy) then
116:   RESIZE (false);
117: return resp;

118: Operation HASHCON(key)
119: t ← Head;
120: b ← t.buckets[key mod t.size ];
121: if (b = NULL) then
122:   s ← t.pred;
123:   if (s ≠ NULL ) then
124:     b ← s.buckets[key mod s.size];
125:   else
126:     b ← t.buckets[key mod t.size];
127: return HASMEMBER (b, key);

128: Method RESIZE(grow)
129: t ← Head;
130: if (t.size > 1 ∨ grow = true) then
131:   for (i ← 0 to t.size-1) do
132:     INITBKT(t, i);
133:   t.pred ← NULL;
134:   size ← grow ? t.size * 2 : t.size/2;
135:   buckets ← new FSet[size ];
136:   t' ← new HNode(buckets, size, t);
137:   CAS(Head, t, t');

138: Method APPLY(optype, key)
139: op ← new FSetOp(optype, key, false, -);
140: while (true) do
141:   t ← Head;
142:   b ← t.buckets[key mod t.size ];
143:   if (b = NULL) then
144:     b ← INITBKT(t, key.mod t.size);
145:   if (INVOKE(b, op)) then
146:     return GETRESPONSE (op);

```

■ **Figure 12** Structure of FSet, FSetOp and HNode. Pseudocodes of INVOKE, FREEZE, ADD, and REMOVE methods based on dynamic sized non-blocking hash table[37].

```

147: Method INITBKT(t, key)
148: b ← t.buckets[key];
149: s ← t.pred;
150: if (b = NULL ∧ s ≠ NULL) then
151:   if (t.size = s.size) then
152:     m ← s.buckets[i mod s.size];
153:     set ← FREEZE(m) ∩ { x | x mod t.size
= i };
154:   else
155:     m ← s.buckets[i];
156:     m' ← s.buckets[i + s.size];
157:     set ← FREEZE(m) ∪ FREEZE(m');
158:     b' ← new FSet(set, true);
159:     CAS(t.buckets[i], NULL, b');
160:   return t.buckets[i];

struct Node{int key;Operation op; Node left,right;}
struct RelocateOp {int state,removeKey,replaceKey;
Node dest; Operation destOp; }
struct ChildCASOp {boolean ifLeft;
Node expected, update; }

161: Operation ADD(key)
162: Node pred, curr, newNode;
163: Operation predOp, currOp, casOp;
164: int result;
165: while (true) do
166:   result ← FIND(key, pred, predOp, curr, currOp,
root);
167:   if (result = FOUND) then return false;
168:   newNode ← new Node(key);
169:   Boolean ifLeft ← (result = NOTFOUND_L);
170:   Node old ← ifLeft ? curr.left : curr.right;
171:   casOp ← new ChildCASOp(ifLeft, old, newNode);
172:   if (CAS(curr.op, currOp, FLAG(casOp, CHILDCAS)))
then
173:     HELPCILDCAS(casOp, curr);
174:     return true;
175: Operation REMOVE(key)
176: Node pred, curr, replace;
177: Operation predOp, currOp, replaceOp, relocOp;
178: while (true) do
179:   if (FIND(key, pred, predOp, curr, currOp, root)
≠ FOUND) then
180:     return false;
181:   if (ISNULL(curr.right ∨ ISNULL(curr.left)))
then
182:     if (CAS(curr.op, currOp, FLAG(currOp, MARKED)))
then
183:       HELPMARKED(pred, predOp, curr);
184:       return true;
185:   else
186:     if ((FIND(key, pred, predOp, replace,
replaceOp, curr) = ABORT) ∨ (curr.op ≠ currOp)
then
187:       continue;
188:     relocOp ← new RelocateOp(curr, currOp,
key, replace.key);
189:     if (CAS(replace.op, replaceOp, FLAG(relocOp,
RELOCATE))) then
190:       if (HELPRELOCATE(relocOp, pred, predOp,
replace)) then
191:         return true;
192: Operation CONTAINS(key)
193: Node pred, curr;
194: Operation predOp, currOp;
195: if (FIND(key, pred, predOp, curr, currOp, root)
= FOUND) then
196:   return true;
197: else
198:   return false;

199: Method FIND(key, pred, predOp, curr, currOp,
root)
200: int result, currKey; Node next, lastRight;
201: Operation lastRightOp; result ← NOTFOUND_R;
202: curr ← root; currOp ← curr.op;
203: if (GETFLAG(currOp) ≠ NULL) then
204:   if (root = root) then
205:     HELPCILDCAS(UNFLAG(currOp), curr);
206:     goto Line 201;
207:   else return ABORT;
208:   next ← curr.right; lastRight ← curr;
209:   lastRightOp ← currOp;
210:   while (¬ ISNULL(next)) do
211:     pred ← curr; predOp ← currOp;
212:     curr ← next; currOp ← curr.op;
213:     if (GETFLAG(currOp) ≠ NULL) then
214:       HELPLIST(pred, predOp, curr, currOp);
215:       goto Line 201;
216:     currKey ← curr.key;
217:     if (key < currKey) then
218:       result ← NOTFOUND_L; next ← curr.left;
219:     else
220:       if (key > currKey) then
221:         result ← NOTFOUND_R; next ← curr.right;
222:         lastRight ← curr; lastRightOp ← currOp;
223:       else
224:         result ← FOUND; break;
225:   if ((result ≠ FOUND) ∧ (lastRightOp ≠
lastRight.op)) then goto Line 201;
226:   if (curr.op ≠ currOp) then goto Line 201;
227:   return result;

```

■ **Figure 13** Structure of Node, RelocateOp and ChildCASOp. Pseudocodes of ADD, REMOVE, CONTAINS and FIND methods based on non-blocking binary search tree[28]. Pseudocodes of CONTAINS, RESIZE, APPLY and INITBKT methods based on dynamic sized non-blocking hash table[37].





# Explicit Space-Time Tradeoffs for Proof Labeling Schemes in Graphs with Small Separators

Orr Fischer ✉

Tel-Aviv University, Israel

Rotem Oshman ✉

Tel-Aviv University, Israel

Dana Shamir ✉

Tel-Aviv University, Israel

---

## Abstract

In distributed verification, our goal is to verify that the network configuration satisfies some desired property, using pre-computed information stored at each network node. This is formally modeled as a *proof labeling scheme* (PLS): a prover assigns to each node a *certificate*, and then the nodes exchange their certificates with their neighbors and decide whether to accept or reject the configuration. Subsequent work has shown that in some specific cases, allowing more rounds of communication – so that nodes can communicate further across the network – can yield shorter certificates, trading off the *space* required to store the certificate against the *time* required for verification. Such tradeoffs were previously known for trees, cycles, and grids, or for proof labeling schemes where all nodes receive the same certificate.

In this work we show that in large classes of graphs, every one-round PLS can be transformed into a multi-round PLS with shorter certificates. We give two constructions: given a 1-round PLS with certificates of  $\ell$  bits, in graphs families with balanced edge separators of size  $s(n)$ , we construct a  $t$ -round PLS with certificates of size  $\tilde{O}(\ell \cdot s(n)/t)$ , and in graph families with an excluded minor and maximum degree  $\Delta$ , we construct a  $t$ -round PLS with certificates of size  $\tilde{O}(\ell \cdot \Delta/\sqrt{t})$ . Our constructions are explicit, and we use erasure codes to exploit the larger neighborhood viewed by each node in a  $t$ -round PLS.

**2012 ACM Subject Classification** Networks; Theory of computation → Distributed algorithms

**Keywords and phrases** proof-labeling schemes, space-time tradeoffs, families with excluded minor

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2021.21

**Funding** Research funded by the Israel Science Foundation, Grant No. 2801/20, and also supported by Len Blavatnik and the Blavatnik Family foundation.

## 1 Introduction

Distributed proofs are a mechanism that allows a distributed network to verify that its current configuration is legal: each node is equipped with a pre-computed *certificate*, which serves as part of a global proof that the network configuration is legal. To verify the proof, nodes examine the certificates in their neighborhoods and decide whether to *accept* or *reject* the proof.

Most of the literature on distributed proofs assumes that in the verification phase, each node can only view the certificates and the network structure in some constant-sized neighborhood around itself: for example, in *proof labeling schemes* [16], nodes only see the certificates of their immediate neighbors, and in *locally-checkable proofs* [10], nodes can see their  $r$ -neighborhood for some constant  $r$ . However, under this restriction it is known that some graph predicates require very large certificates, up to  $\Omega(n^2)$  bits per node [10]. This gives rise to the following question, proposed by [18, 6]: can we trade *time* for *space*? In other words, if we allow the verification phase of the proof to run for more rounds and collect information about a larger neighborhood, can we decrease the certificate size?



© Orr Fischer, Rotem Oshman, and Dana Shamir;  
licensed under Creative Commons License CC-BY 4.0

25th International Conference on Principles of Distributed Systems (OPODIS 2021).

Editors: Quentin Bramas, Vincent Gramoli, and Alessia Milani; Article No. 21; pp. 21:1–21:22

Leibniz International Proceedings in Informatics



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

A distributed proof where the verifier runs in  $t$  rounds is called a  $t$ -proof-labeling scheme, or  $t$ -PLS for short [18]. Our goal when proving space-time tradeoffs is to show that for some problem or class of problems, as  $t$  grows, we can construct a  $t$ -PLS with smaller certificates. Several such tradeoffs are shown in [18, 6]: for example, in [18] it is shown that every graph property on  $n$ -vertex graphs can be verified using certificates of size  $O(n^2/t)$ , and in [6] it is shown that in trees, cycles and grids, we can save a factor of  $O(t)$  in the certificate size of any proof-labeling scheme. However, the general case remains open [6]: is it true that for any network predicate we might wish to verify, the size of the certificate scales down as  $t$  grows? In the current paper we take a step in this direction, and show that for a large class of graphs the answer is positive.

Our main contributions are the following:

- We show that *erasure codes* are a useful building block for constructing  $t$ -PLS, yielding simple, explicit constructions with improved parameters.
- We show that in two large classes of graphs – namely, any graph family that excludes a fixed minor, and any graph family that admits a small balanced edge separator<sup>1</sup> – we can transform a 1-PLS into a  $t$ -PLS with reduced certificate size, depending on the exact parameters of the graph. These families include, for example, planar graphs (and more generally bounded-genus graphs), minor-closed graph families, bounded-treewidth graphs, and others.

Our proof for graph families with an excluded minor generalizes to any graph family with *polynomial expansion*, but for lack of space, we defer the more general case to the full version of the paper.

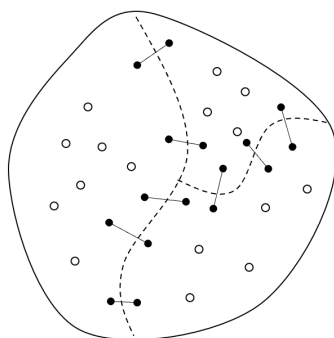
**Using erasure codes to construct  $t$ -proof-labeling schemes.** Suppose we are given a 1-PLS  $\Pi$ , and wish to construct from it a  $t$ -PLS  $\Pi'$  with shorter certificates. A natural approach, used in [18, 6], is to take the certificate  $a_v$  computed under  $\Pi$  for each node  $v$ , “chop it” into pieces  $a_v^1, \dots, a_v^k$  of length  $|a_v|/k$  each (where  $k$  is a parameter of the construction), and distribute the pieces to  $k$  nodes in  $v$ 's  $t$ -neighborhood. In the  $t$ -round verifier of  $\Pi'$ , node  $v$  collects the pieces, reconstructs its certificate  $a_v$ , and uses the original verifier of  $\Pi$  to decide whether to accept or reject.

The main difficulty when designing a  $t$ -PLS of this type is to decide, for each node  $v$ , which nodes in  $v$ 's vicinity should receive which pieces of  $v$ 's certificate, so that no single node  $u$  is given too many certificate pieces to store. Several such constructions are given in [18, 6], each tailored for a specific problem or class of graphs, and one construction in [6] is non-explicit (i.e., the probabilistic method is used). Erasure codes offer a simple mechanism for simplifying this process: an erasure code encodes a string  $w$  into pieces  $c_1, \dots, c_m$ , such that from any sufficiently large subset  $c_{i_1}, \dots, c_{i_d}$  of pieces, we can reconstruct the original string  $w$ . This allows us not to worry about *which* nodes receive *which* pieces of a given certificate: any subset of sufficiently many pieces allows us to reconstruct the certificate in full. In Section 4 we show that using erasure codes, we can simplify, unify and slightly improve two constructions from [18, 6], which transform a 1-PLS where all nodes receive the same certificate (a *uniform PLS* [6]) into a  $t$ -PLS with shorter certificates.

Using codes has other advantages as well: for example, if we use error-correcting codes (of which erasure codes are a special case) to encode the certificates of the nodes, we can gain some degree of resilience against *benign data corruption*, a scenario where the network configuration is still legal but the certificates of some nodes have become corrupted. In

---

<sup>1</sup> We formally define the notion of balanced edge separators later in the paper, but informally, a graph is said to have small balanced edge separators if there is a small set of edges, whose removal partitions the graph into connected components that are not too large.



■ **Figure 1** A partition of the graph, with border nodes indicated in black.

Section 4 we define the notion of *resilient  $t$ -proof-labeling schemes*, and show that by compiling a 1-PLS into a  $t$ -PLS using an error-correcting code, we can withstand up to  $O(b(t))$  corrupted certificates in each  $t$ -neighborhood, assuming the  $t$ -neighborhood is of size at least  $b(t)$ .

**Partition-based  $t$ -proof-labeling schemes.** In [18, 6] it is shown that in trees, cycles and grids, we can transform any 1-PLS with  $\ell$ -bit certificates into a  $t$ -PLS with certificates of size  $O(\ell/t)$ . The idea underlying the three constructions (for trees, cycles and grids) is similar, and we refer to it as a *partition-based  $t$ -PLS*: we partition the graph into units  $U_1, \dots, U_k$ , such that each unit

- Has diameter  $O(t)$ ,
- Contains at least  $r$  nodes, where  $r$  is a parameter of the construction, and
- Contains few *border nodes* – nodes that have neighbors outside the unit (see Fig. 1).

In the  $t$ -PLS, we take the 1-PLS certificate  $a_v$  of each border node  $v$ , and split  $a_v$  into smaller pieces  $a_v^1, \dots, a_v^s$ , where  $s = \Theta(t)$ .<sup>2</sup> We distribute the pieces across the nodes of  $v$ 's unit,<sup>3</sup> in such a way that each node receives  $O(1)$  certificate pieces in total (from all the border nodes). The certificates of non-border nodes are simply ignored. In the resulting  $t$ -PLS, the certificate size is reduced by a factor of  $\Theta(t)$ .

To verify the proof, each border node  $v$  first reconstructs its original certificate  $a_v$  by collecting the  $s$  certificate-parts from its unit. Next, in each unit  $U_i$ , we check whether there is an assignment  $A_i$  of certificates to the non-border nodes of  $U_i$  which would cause all nodes of the unit to accept (including the border nodes, using the certificates they reconstructed for themselves); if so, the nodes of the unit accept, and otherwise they reject. The soundness of this scheme follows from the fact that since the certificates of the border nodes are fixed, we can “stitch together” the assignments  $A_1, \dots, A_k$  that we found for the units  $U_1, \dots, U_k$  into a global certificate assignment that would be accepted by the original 1-PLS.

In Section 5 we define a general scheme for transforming a 1-PLS into a  $t$ -PLS, which codifies the strategy outlined above, and which we then instantiate for graph families that have small separators.

<sup>2</sup> We describe here the parameters used in the constructions of [6]; our constructions use different values.

<sup>3</sup> In the case of trees, the construction from [6] may distribute the pieces across the nodes of an adjacent unit, not necessarily  $v$ 's unit.

**Constructing  $t$ -proof-labeling schemes for graphs with small separators.** The constructions given in [6] for trees, cycles and grids exploit the fact that graphs of these classes can be partitioned into vertex-disjoint units such that if a unit  $U_i$  contains  $b$  border nodes, then the unit has at least  $\Omega(b \cdot t)$  nodes in total, allowing us to distribute pieces of the certificates of the border nodes across the unit efficiently. What other classes of graphs can be partitioned into units with large “area” and a small “perimeter”? It is natural to consider graphs with *bounded expansion*, such as planar graphs, or graphs with bounded treewidth. We show that indeed, there are two such classes of graphs for which we can construct a  $t$ -PLS with shorter certificates compared to a 1-PLS. In Section 7, we handle graph families that admit a *small balanced edge separator* (see Section 7 for the definition), and prove:

► **Theorem 1.** *Fix a subgraph-closed family  $\mathcal{G}$  and a function  $s : \mathbb{N} \rightarrow \mathbb{N}$ , such that any graph  $G \in \mathcal{G}$  has a balanced edge separator of size  $s(|V(G)|)$ . Let  $\Pi$  be a 1-PLS recognizing some predicate  $\mathcal{P}$ . Then for every  $t \geq 2$ , there exists a  $t$ -PLS  $\Pi'$  for  $\mathcal{P}$ , such that for every configuration  $(G, I) \in \mathcal{P}$  of diameter at least  $t$ ,*

$$\text{cost}(\Pi', (G, I)) = \tilde{O}\left(\frac{\text{cost}(\Pi, (G, I)) \cdot s(n)}{t}\right).$$

Here,  $\text{cost}(\Pi, (G, I))$  denotes the length of the certificate assigned by the PLS  $\Pi$  to the nodes of the graph  $G$ , when the input to each node is given by  $I : V(G) \rightarrow \mathcal{I}$  (see Section 3 for the formal definitions).

Graphs that have small balanced edge separators include low-degree graphs with small treewidth; for example, outerplanar graphs have treewidth 2, and are known to admit a balanced edge separator of size at most  $O(\Delta)$ , where  $\Delta$  is the maximum degree.

In Section 8 we handle graph families that excludes some constant-sized minor, and show:

► **Theorem 2.** *Fix a family  $\mathcal{G}$  of graphs that exclude some fixed minor  $H$ . Let  $\Pi$  be a 1-PLS recognizing some predicate  $\mathcal{P}$ . Then for every  $t \geq 2$ , there exists a  $t$ -PLS  $\Pi'$  recognizing  $\mathcal{P}$ , such that for every configuration  $(G, I) \in \mathcal{P}$  of diameter at least  $t$  and maximum degree  $\Delta$ ,*

$$\text{cost}(\Pi', (G, I)) = \tilde{O}\left(\frac{\text{cost}(\Pi, (G, I)) \cdot \Delta}{\sqrt{t}}\right).$$

Many interesting graph families can be described by a set of forbidden (excluded) minors; perhaps the most famous example is *planar graphs*, and more generally, graphs of fixed genus.

Strictly speaking, the two constructions above could be implemented without using erasure codes, but using erasure codes simplifies them.

We conjecture that the connection between the existence of an efficient  $t$ -PLS and bounded expansion goes both ways, namely, in graphs that are *good expanders*,<sup>4</sup> a  $t$ -PLS cannot always have certificate size significantly smaller than a 1-PLS for the same predicate.

## 2 Related Work

Distributed proofs, proof-labeling schemes have been extensively studied under various modeling assumptions (e.g. [2, 4, 5, 12, 13, 15, 14, 16, 19, 21, 7, 11, 17, 10]). In the current section we discuss only prior work that is directly relevant to the current paper; we refer to the surveys of Feuilloley and Fraigniaud [3] and of Suomela [22] for a more complete overview of the field.

<sup>4</sup> An *expander* is a graph where every sufficiently-small set  $S$  of vertices has a *large boundary* – e.g.,  $S$  has many outgoing edges, or many adjacent nodes, depending on the precise definition used.

While almost all work on proof-labeling schemes assumes that the verifier runs for a single round, Göös et al. [10] considered a model where the verifier may run for  $r$  rounds, where  $r$  is a constant. Among other results, they showed a *universal 1-PLS* which uses  $O(\min(n^2, m \log n))$ -bit certificates by giving each node an encoding of the entire graph; this can be used to locally check any property  $\mathcal{P}$ . On the other hand, they showed that there are properties that require a certificate size of  $\Omega(n^2)$  bits to be checked locally, such as the existence of a non-trivial automorphism.

Ostrovsky et al. [18] introduced the notion of a  $t$ -PLS, in which the verification procedure is allowed to use  $t$  rounds of communication instead of only one. They analyze the tradeoffs between time (number of rounds), certificate size, and total communication, and show that the certificate size of a universal  $t$ -PLS is  $O(\min(n^2, m \log n)/t)$ . In the universal  $t$ -PLS of [18], the graph is divided into blocks, and the representation of the graph is distributed between the nodes in each block, such that each node can collect the entire graph representation from its  $t$ -neighborhood. They also construct an optimal  $t$ -PLS that determines whether a graph is acyclic using  $O(\log n/t)$ -bit certificates, and prove a matching lower bound.

Feuilleley et al. [6] gives additional general tradeoffs for  $t$ -proof-labeling schemes. First, they show near-linear scaling of the certificate size for any proof labeling scheme in trees, cycles and grids (as outlined in Section 1). In the current paper we use a similar but more general approach in larger classes of graphs. Feuilleley et al. [6] also show that every *uniform 1-PLS* (a PLS in which the prover gives the same certificate to all the nodes) can be transformed into a  $t$ -PLS with certificates smaller by a factor of  $O(\log(n)/b(t))$ ; here,  $b(t)$  is the minimum number of nodes in a  $t$ -neighborhood of a node in the graph. Their construction uses the probabilistic method: they show that if each node stores each bit of the original certificate with probability roughly  $\log(n)/b(t)$ , then there is non-zero probability that each node will have all bits of the certificate in its  $t$ -neighborhood, which allows all nodes to recover the original certificate. In the current paper we use erasure codes to simplify the scheme, eliminate the multiplicative factor of  $\log(n)$ , and obtain an explicit construction. Finally, [6, 18] also construct  $t$ -PLS for specific problems (e.g., shortest paths and cycle-freeness).

### 3 Preliminaries

**Graph notation.** Given a graph  $G$ , we let  $V(G)$  and  $E(G)$  denote the vertex set and edge set of  $G$ , respectively. We let  $H \subset G$  denote the fact that  $H$  is a subgraph of  $G$  (that is,  $V(H) \subseteq V(G)$  and  $E(H) \subseteq E(G)$ ). We sometimes abuse notation by writing  $v \in G$  instead of  $v \in V(G)$ .

The neighborhood of a node  $v \in V(G)$  is denoted  $N(v)$ . The  $t$ -neighborhood of a node  $v \in V(G)$ , i.e., the set of all the nodes with distance at most  $t$  from  $v$ , is denoted  $B_t(v)$  (“the ball of size  $t$  around  $v$ ”). We say that a graph  $G$  has growth  $b : \mathbb{N} \rightarrow \mathbb{N}$  if for every  $v \in V$  and  $t \geq 1$ ,  $|B_t(v)| \geq b(t)$ .

An  $\mathcal{I}$ -*configuration* (or *configuration* for short) is a pair  $(G, I)$ , where  $G = (V, E)$  is a graph, and  $I : V \rightarrow \mathcal{I}$  is an assignment of inputs to the nodes of  $G$ . Given a family  $\mathcal{G}$  of graphs, we denote by  $\mathcal{G}_{\mathcal{I}}$  the set of all  $\mathcal{I}$ -configurations  $(G, I)$  where  $G \in \mathcal{G}$  and  $I : V \rightarrow \mathcal{I}$ .

**Proof labeling schemes.** Let  $\mathcal{G}_{\mathcal{I}}$  be a family of configurations, let  $\mathcal{P} \subseteq \mathcal{G}_{\mathcal{I}}$  be a predicate, and let  $t > 1$ . A  $t$ -*proof labeling scheme* ( $t$ -PLS) for  $\mathcal{P}$  is a pair  $\Pi = (\mathbf{Prv}, \mathbf{Ver})$ , where

- **Prv**, the *prover*, is a mapping that takes a configuration  $(G, I) \in \mathcal{P}$  and produces a certificate assignment  $\mathbf{Prv}(G, I) = \{a_v\}_{v \in V(G)}$  for the nodes of  $G$ , where  $a_v \in \{0, 1\}^*$  for each  $v \in V(G)$ .

- **Ver**, the *verifier*, is a  $t$ -round deterministic<sup>5</sup> distributed algorithm that takes as input the certificate  $a_v \in \{0, 1\}^*$  at each node  $v$ , and outputs a Boolean value. We let  $\mathbf{Ver}(G, I, a, v)$  denote the output of the algorithm at node  $v \in V(G)$ , when executed in configuration  $(G, I)$ , with certificates  $a = \{a_v\}_{v \in V(G)}$ .

We require:

- **Soundness**: for every configuration  $(G, I) \in \mathcal{G}_{\mathcal{I}}$  and certificate assignment  $a = \{a_v\}_{v \in V(G)}$ , if  $\mathbf{Ver}(G, I, a, v) = 1$  for all  $v \in V(G)$ , then  $(G, I) \in \mathcal{P}$ .
- **Completeness**: for every configuration  $(G, I) \in \mathcal{P}$  and for every node  $v \in V(G)$  we have  $\mathbf{Ver}(G, I, \mathbf{Prv}(G, I), v) = 1$ .

We note that the verifier may send arbitrarily large messages (i.e., the verifier is a  $t$ -round LOCAL algorithm).

A *1-PLS* is a restricted type of proof labeling scheme  $\Pi = (\mathbf{Prv}, \mathbf{Ver})$ , following the original definition from [16]: the prover is the same as in the case of  $t$ -PLS for  $t > 1$  above, but the verifier is restricted to a single round, where each node  $v$  sends its certificate  $a_v$  to all of its neighbors. In the case of a 1-PLS, we let  $\mathbf{Ver}(v, I(v), N(v), a_v, \{a_u\}_{u \in N(v)})$  denote the output of the verifier at node  $v$ , when node  $v$ 's input is  $I(v)$ , its neighborhood is  $N(v)$ , its certificate is  $a_v$ , and its neighbors' certificates are  $\{a_u\}_{u \in N(v)}$ .

We say a  $t$ -PLS is *uniform* if it gives the same certificates to all nodes in the graph, i.e., for every  $(G, I) \in \mathcal{G}_{\mathcal{I}}$  and  $v, u \in V(G)$ ,  $\mathbf{Prv}(G, I)(v) = \mathbf{Prv}(G, I)(u)$ .

Given a  $t$ -PLS  $\Pi = (\mathbf{Prv}, \mathbf{Ver})$ , the *cost* of  $\Pi$  in a configuration  $(G, I)$ , denoted  $\text{cost}(\Pi, (G, I))$ , is the length of the longest certificate in  $\mathbf{Prv}(G, I)$ . If  $\Pi$  is a  $t$ -PLS for a predicate  $\mathcal{P}$ , we define the cost of  $\Pi$  in a family of configurations  $\mathcal{G}_{\mathcal{I}}$  as follows:

$$\text{cost}(\Pi, \mathcal{P}, n) = \max \{ \text{cost}(\Pi, (G, I)) : (G, I) \in \mathcal{P}, |V(G)| = n \}$$

Note that the cost is based only on configurations that satisfy  $\mathcal{P}$ , as the prover has no particular obligation otherwise.

**Initial knowledge.** Our techniques require the nodes to have unique identifiers, but generally the nodes do not need to know in advance the size of the graph or the value of any other graph parameter; accordingly, the  $t$ -PLS we construct in Section 4 does not assume that nodes know the size of the graph. However, to simplify the presentation of the remainder of our results, following Section 4 we do assume that the size  $n$  of the graph is known to all the nodes. This is not essential, but it avoids some technical details.

**Erasure codes.** An erasure code is a type of code that allows us to recover from the erasure of some symbols in the codeword:

► **Definition 3.** *Given a prime number  $q \in \mathbb{N}$  and parameters  $n, k, d \in \mathbb{N}$ , an  $(n, k, d)_q$ -erasure code is a pair  $(\text{Enc}, \text{Dec})$ , where*

- $\text{Enc} : \mathbb{F}_q^k \rightarrow \mathbb{F}_q^n$  is an encoder,
- $\text{Dec} : (\mathbb{F}_q \cup \{?\})^n \rightarrow \mathbb{F}_q^k$  is a decoder,
- For each  $w \in \mathbb{F}_q^k$ , if  $c \in (\mathbb{F}_q \cup \{?\})^n$  is obtained from  $\text{Enc}(w)$  by replacing fewer than  $d$  symbols by '?', then  $\text{Dec}(c) = w$ .

<sup>5</sup> It is interesting to explore the case where the verifier is *randomized*, as this is known to help in some contexts [8], but in the current paper the verifiers that we construct are deterministic.

Note that  $n$  is used for both the size of the graphs we are working with and the length of the code, and this suits our purposes, because whenever we apply an erasure code, the length of the code will be the number of nodes in the graph: for convenience, we always produce one piece for each node in the graph, even though not all the pieces are used in every construction.

The codes that we use in the current paper are  $(n, k, n - k + 1)_q$ -erasure codes, which are optimal in the number of erasures that they can tolerate. For such a code to exist, it suffices to have  $k \leq n \leq q$  [20]. The celebrated *Reed-Solomon code* is an example of such a code.

Definition 3 assumes that the decoder is given a string of length  $n$  where at most  $d$  symbols have been *erased*, i.e., replaced by '?'. For our purposes here, it is more convenient to think of the decoder as a function that takes a set of at most  $n$  *pieces*, which are numbered symbols of the form  $(i, c) \in [n] \times \mathbb{F}_q$ , and reconstructs a word  $w \in \mathbb{F}_q^k$ . Formally, given a set  $S = \{(i_1, c_1), \dots, (i_{n'}, c_{n'})\}$  such that  $n' \leq n$  and  $i_j \neq i_{j'}$  for every  $j \neq j'$ , we abuse notation by writing  $Dec(S)$  to represent the output of the following procedure: let  $c' \in (\mathbb{F}_q \cup \{?\})^n$  be the string where

$$c'_j = \begin{cases} c_j & \text{if } (j, c_j) \in S, \\ ? & \text{otherwise.} \end{cases}$$

Then we return  $Dec(c')$ . (Note that this is well-defined, because we assumed that there is at most one value  $c_j$  such that  $(j, c_j) \in S$  for each  $j$ .)

**From binary certificates to codewords over  $\mathbb{F}_q$ .** In the current paper we use erasure codes to encode certificates that are represented as binary strings. To do so, we view the certificate as a string over some finite field  $\mathbb{F}_q$ , where  $q$  is a sufficiently large prime number. The size  $q$  of the field we must take depends on the total number  $n$  of pieces, the length  $\ell$  of the binary certificate, and the number  $m \leq n$  of pieces that suffice to reconstruct the certificate: given  $n, \ell, m \in \mathbb{N}$ , let

$$q(n, \ell, m) = \max\left(n, 2^{\lceil \ell/m \rceil}\right). \quad (1)$$

Let  $\ell' = \lceil \ell / \log q_{n, \ell, m} \rceil$  be the number of  $\mathbb{F}_{q_{n, \ell, m}}$ -elements required to represent an  $\ell$ -bit string. By choice of  $q_{n, \ell, m}$  we have  $\ell' \leq m \leq n \leq q_{n, \ell, m}$ .

Throughout the paper, we fix a family  $\{C_{n, \ell, m}\}_{n, \ell, m \in \mathbb{N}}$  of erasure codes, where each  $C_{n, \ell, m} = (Enc_{n, \ell, m}, Dec_{n, \ell, m})$  is an  $(n, \ell', n - \ell' + 1)_{q_{n, \ell, m}}$ -erasure code. (Such a code exists, because  $\ell' \leq m \leq n \leq q_{n, \ell, m}$ .) To encode a certificate  $a \in \{0, 1\}^\ell$ , we view  $a$  as an  $\ell'$ -symbol string over  $\mathbb{F}_{q_{n, \ell, m}}$ , and apply the encoder  $Enc_{n, \ell, m}$ . To reconstruct the certificate from  $m$  or more pieces, we apply the decoder  $Dec_{n, \ell, m}$  to obtain an  $\ell'$ -symbol string over  $\mathbb{F}_{q_{n, \ell, m}}$ , which we then view as an  $\ell$ -bit binary string.

When using the code  $C_{n, \ell, m}$ , the length (in bits) of each piece that we produce is

$$\log q(n, \ell, m) = O((\ell/m) + \log n), \quad (2)$$

matching the intuition that  $m$  pieces of size roughly  $\ell/m$  are necessary and sufficient to reconstruct an  $\ell$ -bit string.

#### 4 Space-Time Tradeoff for Uniform Proof Labeling Schemes

Recall that a *uniform* proof labeling scheme is one where the prover assigns the same label to all nodes. To illustrate the usefulness of erasure codes, in this we use them to transform a uniform 1-PLS II for a graph family with growth  $b : \mathbb{N} \rightarrow \mathbb{N}$ , into a  $t$ -PLS for the same

predicate using certificates that are smaller by a factor of roughly  $b(t-1)$ . This is a simpler, tighter and explicit proof for a similar claim from [6],<sup>6</sup> and it also generalizes the universal  $t$ -PLS from [18].

In this section we do not assume that the nodes of the graph know the size  $n$  of the graph or the growth function  $b$ . (Since the size of the certificates does depend on these parameters, we assume that certificates are encoded in some predetermined variable-length encoding.) Instead, we ask the prover to provide  $n$  and  $b$  to each node; the prover may lie, but we can show that this does not affect the soundness of our construction.

► **Theorem 4.** *Let  $\mathcal{G}_{\mathcal{I}}$  be a family of configurations,  $\mathcal{P} \subseteq \mathcal{G}_{\mathcal{I}}$  be a predicate and  $\Pi$  be a uniform 1-PLS for  $\mathcal{P}$ . Then for every  $t > 1$ , there exists a  $t$ -PLS  $\Pi'$  for  $\mathcal{P}$ , such that for every configuration  $(G, I) \in \mathcal{P}$  with  $n$  nodes and growth  $b : \mathbb{N} \rightarrow \mathbb{N}$ ,*

$$\text{cost}(\Pi', (G, I)) = O\left(\frac{\text{cost}(\Pi, \mathcal{P}, n)}{b(t-1)} + \log n\right).$$

**Proof.** Let  $\Pi = (\mathbf{Prv}, \mathbf{Ver})$  be the uniform 1-PLS and fix  $t > 1$ . Let  $k(n) = \text{cost}(\Pi, \mathcal{P}, n)$ . We define the  $t$ -PLS  $\Pi' = (\mathbf{Prv}', \mathbf{Ver}')$  as follows.

Given an  $n$ -node configuration  $(G, I) \in \mathcal{P}$  with growth factor  $b : \mathbb{N} \rightarrow \mathbb{N}$ , let  $a \in \{0, 1\}^{k(n)}$  be the certificate assigned by  $\mathbf{Prv}$  to the nodes  $\{v_1, \dots, v_n\}$  of  $G$ . For convenience we denote  $k = k(n)$ ,  $b = b(t-1)$  and  $q = q(n, k, b)$ . We use the erasure code  $C_{n,k,b} = (\text{Enc}_{n,k,b}, \text{Dec}_{n,k,b})$  to generate the encoding  $\text{Enc}_{n,k,b}(a) = (c_1, \dots, c_n) \in \mathbb{F}_q^n$ .

The new prover  $\mathbf{Prv}'$  assigns each node  $v$  of  $G$  the certificate  $a'_v = (i, c_i, b, n)$ . The length of the certificate is  $O(k/b + \log n)$ : encoding the index  $i \in [n]$ , the ball size  $b = b(t-1) \leq n$ , and the graph size  $n$  requires  $O(\log n)$  bits. The piece  $c_i \in \mathbb{F}_q$  is of length  $O(k/b + \log n)$ , by (2).

Next we describe the verifier  $\mathbf{Ver}'$ . Each node  $v$  uses  $t$  rounds of communication to learn the entire ball  $B_t(v)$  of radius  $t$  around itself, including all certificates of the nodes in  $B_t(v)$ . Node  $v$  verifies that:

- All certificates in the ball are well-formed 4-tuples.
- All certificates agree on the last two values (the claimed values of  $b$  and  $n$ ), and
- For any two certificates  $(i_1, c_{i_1}, b_1, n_1), (i_2, c_{i_2}, b_2, n_2)$  collected, we have  $i_1 \neq i_2$ .

In the sequel, we denote by  $s_x = (i_x, c_{i_x}, \tilde{b}, \tilde{n})$  the certificate of node  $x \in B_t(v)$ . As we said above, the values  $\tilde{b}, \tilde{n}$  are the same across all certificates in  $B_t(v)$ , otherwise  $v$  rejects. Also let  $\tilde{k} = k(\tilde{n})$  and  $\tilde{q} = q(\tilde{n}, \tilde{k}, \tilde{b})$ .

For each  $u \in N(v) \cup \{v\}$ , let  $S_u = \{(i_x, c_{i_x}) : x \in B_{t-1}(u)\}$  denote the pieces in the  $(t-1)$ -ball around node  $u$ . Node  $v$  verifies that  $|S_u| \geq \tilde{b}$ ,  $i_x \in [\tilde{n}]$ , and  $c_{i_x}$  is an element of  $\mathbb{F}_{\tilde{q}}$ . Next, for each  $u \in N(v) \cup \{v\}$ , node  $v$  uses the decoder  $\text{Dec}_{\tilde{n}, \tilde{k}, \tilde{b}}$  to reconstruct from  $S_u$  a certificate  $a_u = \text{Dec}_{\tilde{n}, \tilde{k}, \tilde{b}}(S_u) \in \{0, 1\}^{\tilde{k}}$ .

Finally, node  $v$  verifies that:  $a_u = a_v$  for every  $u \in N(v)$ , and that the original verifier  $\mathbf{Ver}(v, I(v), N(v), a_v, \{a_u\}_{u \in N(v)})$  accepts. If so, node  $v$  accepts, and otherwise it rejects.

For lack of space, we omit the detailed proof that  $\Pi'$  is sound and complete here. Completeness is straightforward; soundness is proven by showing that if all nodes accept, then all have reconstructed the same certificate  $a \in \{0, 1\}^k$  for themselves and their neighbors, and the original verifier  $\mathbf{Ver}$  accepts this certificate. ◀

<sup>6</sup> In [6] it is claimed that the factor saved is  $b(t)$ , but we believe it should be  $b(t-1)$ , as the verifier requires one extra round to make sure that all nodes have reconstructed the same certificate.



**Bounding the message size.** In the universal  $t$ -PLS construction from [18], the verifier uses messages whose size is bounded by the size of the certificate, whereas our construction above (and the corresponding one from [6]) requires nodes to learn their entire  $t$ -neighborhood, which cannot be done using small messages. However, our construction is easily modified to work with messages whose size is bounded by the certificate size: instead of learning the entire neighborhood, we use pipelining to have each node collect  $t$  pieces from its  $t$ -neighborhood, allowing us to save a factor of  $\Theta(t)$  in the certificate size (instead of  $b(t-1)$ ). A bit of additional effort is required to ensure that all nodes reconstruct the same certificate; the details are deferred to the full version of the paper.

**Handling benign data corruption.** If we replace erasure codes by their more powerful cousins, *error-correcting codes*, we gain the ability to withstand some bounded number of *corrupted certificates* in every neighborhood. Since proof-labeling schemes are intended to enable fault-tolerance and self-stabilization, this can be useful. We introduce the notion of a *resilient  $t$ -PLS*:

► **Definition 5.** Let  $f : \mathbb{N} \rightarrow \mathbb{N}$ ,  $\mathcal{G}_{\mathcal{I}}$  a family of graph configurations and  $\mathcal{P}$  a Boolean predicate over  $\mathcal{G}_{\mathcal{I}}$ . An  $f$ -resilient  $t$ -PLS for  $\mathcal{P}$  is a  $t$ -PLS  $\Pi = (\mathbf{Prv}, \mathbf{Ver})$  satisfying:

- Soundness, defined as in Section 3.
- $f$ -resilient completeness: for every configuration  $(G, I) \in \mathcal{P}$  and node  $v \in V(G)$ , if  $a'$  is a certificate assignment that agrees with  $\mathbf{Prv}(G, I)$  on all but at most  $f(r)$  certificates in  $B_r(v)$  for every  $r \leq t$ , then  $\mathbf{Ver}(G, I, a', v) = 1$ .

In the full version of the paper, we show that using error-correcting codes, we can withstand  $f(t)$  potential corruptions in every  $t$ -neighborhood, and still save a factor of roughly  $b(t-1) - 2f(t-1)$  in the certificate size.

## 5 Partition-Based $t$ -Proof-Labeling Schemes

Next, we introduce our constructions for  $t$ -PLS in graphs with small separators. We begin by defining a class of  $t$ -PLS, called *partition-based schemes*, which codify the idea underlying our constructions and several constructions from [18, 6], and in the following sections we give the details of our two concrete constructions.

In a partition-based scheme, we partition the graph into vertex-disjoint connected components  $C_1, \dots, C_m$ , each with a small boundary  $Y_i \subseteq C_i$  separating it from the rest of the graph, and distribute the certificates of the boundary nodes across the nodes in their component. The diameter of each component must be less than  $t$ , so that during the verification phase, each node in the component can learn the entire component, and the boundary nodes can recover the pieces of their certificate. Formally, we require the following properties for each component  $C_j$  in the partition:

- (P-1) The induced subgraph  $G[C_j]$  is connected and has diameter at most  $r$ , where  $r < t-1$  is a parameter of the construction.
- (P-2) Only boundary nodes may have neighbors outside their own component: if  $u \in C_j$  has a neighbor  $v \in N(u)$  such that  $v \notin C_j$ , then  $u, v \in \bigcup_{i \in [m]} Y_i$ .

Given a 1-PLS  $\Pi = (\mathbf{Prv}, \mathbf{Ver})$  with certificate length  $k$ , and a partition satisfying the conditions above, we construct a  $t$ -PLS along the following outline.

**Partition-based provers.** Let  $k = k(n)$  and  $q = q(n, k, d)$ , where  $n$  is the size of the input graph (which we now assume is known to the nodes),  $k(n)$  is the certificate length of  $\Pi$  in graphs of size  $n$ , and  $d$  is a parameter of our construction.

## 21:10 Explicit Space-Time Tradeoffs for PLS in Graphs with Small Separators

Our new prover  $\mathbf{Prv}'$  begins by computing, for each boundary node  $x \in \bigcup_j Y_j$ , the certificate  $a_x$  that the original prover  $\mathbf{Prv}$  would give to node  $x$ . The prover then uses  $C_{n,k,d}$  to encode  $a_x$  across some nodes  $U_x \subseteq B_t(x)$  in  $x$ 's vicinity, giving each node  $v \in U_x$  a piece of the encoding of  $a_x$ . We refer to  $a_x$  as *the  $\mathbf{Prv}$ -certificate of  $x$* , to distinguish it from the certificate assigned by the new prover  $\mathbf{Prv}'$ . The  $\mathbf{Prv}$ -certificates of non-boundary nodes are not used; the verifier will guess them.

In addition to the  $\mathbf{Prv}$ -certificates of the boundary nodes,  $\mathbf{Prv}'$  specifies the partition into components, by telling each node the index of the component to which it belongs, and whether or not it is a boundary node. Thus, the certificate that  $\mathbf{Prv}'$  assigns to each node  $v \in V$  is of the form  $(cid_v, s_v, P_v)$ , where

- $cid_v \in \mathbb{N}$  is the index of the component to which  $v$  belongs,
- $s_v \in \{0, 1\}$  indicates whether or not  $v$  is a boundary node, and
- $P_v \subseteq V \times \mathbb{N} \times \mathbb{F}_q$  is a collection of pieces of  $\mathbf{Prv}$ -certificates for some boundary nodes in  $v$ 's vicinity. Each piece is of the form  $(x, i, c_i)$ , where  $x$  is the ID of a boundary node,  $i$  is the index of the piece, and  $c_i$  is the  $i$ -th piece in the encoding  $Enc_{n,k,d}(a_x) = (c_1, \dots, c_n)$ .

The actual construction of the prover  $\mathbf{Prv}'$  depends on the graph family we want to handle: specifically,  $\mathbf{Prv}'$  must be able to compute a “good” partition for graphs from the family, and it must be able to assign pieces of the original certificates of the boundary nodes, in such a way that no node receives too many pieces (to keep the certificates of  $\mathbf{Prv}'$  short). In the current paper we construct two partition-based provers:

- In Section 7 we construct a prover for graph families with a small edge separator, and
- In Section 8 we construct a prover for graph families that exclude some fixed minor.

**The partition-based verifier.** We define a *single verifier*  $\mathbf{Ver}_{r,d}^t$ , parameterized by:

- The number of rounds  $t$  that the verifier may use,
- An upper bound  $r < t - 1$  on the diameter of each component  $C_i$  in the honest prover's partition,
- The parameter  $d$  of the erasure codes  $C_{n,k(n),d}$  used by the honest prover.

The verifier  $\mathbf{Ver}_{r,d}^t$  is sound whenever the original 1-PLS verifier  $\mathbf{Ver}$  on which it is based is sound. To yield a sound and complete  $t$ -PLS, it can be combined with any partition-based prover that matches the parameters  $r, d, t$  and distributes certificate pieces in a way that each node can reconstruct what it needs.

We formally define  $\mathbf{Ver}_{r,d}^t$  in the next section, but on a high level, it operates as follows: each node  $v$  learns its  $\Theta(t)$ -neighborhood and the certificates in it, and verifies that the component containing  $v$  is well-formed and has diameter at most  $r$ . Node  $v$  uses the certificate-pieces in its  $\Theta(t)$ -neighborhood to reconstruct an “original certificate”  $a_x$ , purportedly computed by the original prover  $\mathbf{Prv}$ , for each boundary node  $x$  in  $v$ 's component. It then checks if there exists an assignment of “original certificates” to the non-boundary nodes in  $v$ 's component, that would cause all nodes in the component to accept; if so,  $v$  accepts, and otherwise  $v$  rejects.

The soundness of this construction stems from the fact that we can “stitch together” the original certificates reconstructed or guessed for the various components, into a certificate assignment that is accepted by the original verifier  $\mathbf{Ver}$ .

## 6 The Partition-Based Verifier

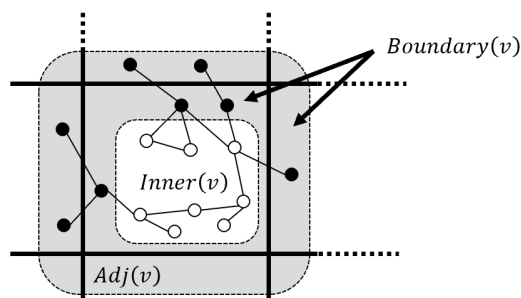
In this section we define the partition-based  $t$ -round verifier  $\mathbf{Ver}_{r,d}^t$ , assuming we are given a 1-PLS verifier  $\mathbf{Ver}$ .

As explained in Section 5, we assume that every node  $v \in V$  is given a certificate of the form  $(cid_v, s_v, P_v)$ , where  $cid_v \in \mathbb{N}$ ,  $s_v \in \{0, 1\}$ , and  $P_v \subseteq \mathcal{V} \times [n] \times \mathbb{F}_q$ , where  $\mathcal{V}$  is the domain from which UIDs are drawn. We say that node  $v$  is a *boundary node* if  $s_v = 1$ , and otherwise node  $v$  is an *inner node*.

Each node  $v$  spends the first  $t - 1$  rounds learning its entire  $(t - 1)$ -neighborhood  $B_{t-1}(v)$ , including the graph structure, the certificates and the inputs of the nodes. Define the following subsets of  $B_{t-1}(v)$  (see Fig. 2):

- $\tilde{C}(v) = \{u \in B_{t-1}(v) : cid_u = cid_v\}$ . These are the nodes that the prover has indicated belong to the same component as  $v$ .
- $Adj(v) = (B_{t-1}(v) \cap N(\tilde{C}(v))) \setminus \tilde{C}(v)$ . These are the nodes adjacent to  $\tilde{C}(v)$  which node  $v$  can see in its  $(t - 1)$ -neighborhood.
- $Inner(v) = \{u \in \tilde{C}(v) : s_u = 0\}$ ,
- $Boundary(v) = (\tilde{C}(v) \setminus Inner(v)) \cup Adj(v)$ ,
- $All(v) = Inner(v) \cup Boundary(v)$ .

Intuitively, the nodes in  $Inner(v)$  are the nodes for which  $v$  will “guess” certificates, and the nodes in  $Boundary(v)$  are the nodes for which  $v$  will reconstruct certificates from pieces given by the prover.



■ **Figure 2** The sets  $Inner(v)$ ,  $Boundary(v)$  and  $Adj(v)$  computed by the verifier at node  $v$ . The bold lines indicate the partition into components,  $\{\tilde{C}(u)\}_{u \in V}$ .

Node  $v$  verifies the following conditions:

- (V-1) The subgraph induced by  $\tilde{C}(v)$  on  $v$ 's  $(t - 1)$ -neighborhood is connected and has diameter at most  $r$ .
- (V-2) All neighbors of  $v$  that are inner nodes are in  $\tilde{C}(v)$ , that is, for every  $u \in N(v)$ , if  $s_u = 0$ , then  $cid_u = cid_v$ .

The requirements (V-1) and (V-2) mirror our requirements (P-1) and (P-2) from the partition-based prover (see Section 5): if the prover constructed a partition satisfying (P-1) and (P-2), then (V-1) and (V-2) will be satisfied at all nodes. We point out that (V-2), when verified at all nodes, implies that every node in  $Boundary(v)$  is a boundary node: if  $x \in (\tilde{C}(v) \setminus Inner(v))$  then this is immediate, and if  $x \in Adj(v)$ , then  $x \notin \tilde{C}(v)$  but  $x$  has a neighbor  $y \in \tilde{C}(v)$ . If  $x$  were not a boundary node, then  $y$  would reject when verifying (V-2).

The next step is for node  $v$  to reconstruct original certificates for the nodes in  $Boundary(v)$ , as follows: for each  $x \in Boundary(v)$ , let  $A_v(x) = \{(x, i_x^u, c_x^u) \in P_u : u \in B_{t-1}(v)\}$  be the pieces associated with node  $x$  that node  $v$  can see in its  $(t - 1)$ -neighborhood. Node  $v$  verifies that  $|A_v(x)| \geq d$ , and that for every  $u \neq u'$  in  $B_{t-1}(v)$  we have  $i_x^u \neq i_x^{u'}$ . Then, node  $v$  applies the decoder  $Dec_{n,k,d}$  to  $A_v(x)$  to reconstruct a string  $\tilde{a}_v(x) \in \{0, 1\}^k$ .

After reconstructing a certificate  $\tilde{a}_v(x)$  for each  $x \in Boundary(v)$ , node  $v$  sends the list  $\{(x, \tilde{a}_v(x)) : x \in Boundary(v)\}$  to all of its neighbors, and receives from each neighbor  $u \in N(v)$  a list  $\{(x, \tilde{a}_u(x)) : x \in Boundary(u)\}$ . Node  $v$  verifies that for every  $x \in Boundary(v) \cap Boundary(u)$  the same certificate was reconstructed by  $v$  and  $u$ , that is,  $\tilde{a}_v(x) = \tilde{a}_u(x)$ .

Following the previous step, node  $v$  has an assignment  $\tilde{a}_v : \text{Boundary}(v) \rightarrow \{0, 1\}^k$  of certificates to the boundary nodes. It now checks if there exists an extension of  $\tilde{a}_v$  to all nodes in  $All(v)$ , such that the original verifier  $\mathbf{Ver}$  accepts  $\tilde{a}_v$  at every node in  $\tilde{C}(v)$ . Formally, we require that  $\mathbf{Ver}(u, I(u), N(u), \tilde{a}_v(u), \{\tilde{a}_v(w)\}_{w \in N(u)}) = 1$  for every  $u \in \tilde{C}(v)$ . If such an extension exists, then node  $v$  accepts, and otherwise it rejects.

In Appendix A, we prove that our verifier is sound, by showing that if  $\mathbf{Ver}_{r,d}^t$  accepts at all nodes, then there is a  $\mathbf{Prv}$ -certificate assignment that causes the original verifier  $\mathbf{Ver}$  to accept at all nodes.

## 7 A Partition-Based $t$ -PLS for Graphs with Small Edge Separators

In this section we construct a partition-based prover  $\mathbf{Prv}_{sep}^t$  for graph families that admit *small balanced edge separators*:

► **Definition 6.** Fix  $\alpha \in (1/2, 1)$ . A set  $S$  of edges  $S \subseteq E$  is an  $\alpha$ -edge separator for a graph  $G = (V, E)$  if the graph  $G' = (V, E \setminus S)$  that remains after the edges in  $S$  are removed has no connected components of size larger than  $\alpha|V|$ .

We note that in our construction we always choose a *minimal* balanced edge separator, that is, an edge separator  $S$  such that the removal of any edge  $e \in S$  would yield a set  $S \setminus \{e\}$  that is not a balanced edge separator. This will be important for correctness.

**The cost of an edge separator.** We define the *cost* of an edge separator  $S \subseteq E$  to be the number of nodes incident to edges in  $S$ :  $\text{cost}(S) = |\{u \in V : \exists v(u, v) \in S\}|$ .

We note that this differs from the standard definition – usually, one tries to find a set  $S \subseteq E$  of minimum cardinality (i.e., the smallest number of edges). However, in our construction, it is the *number of nodes incident to  $S$*  that matters, not the size of  $S$  itself, because the prover will need to specify a certificate for every node incident to  $S$ . Our notion of cost is never greater than the standard notion (the number of edges), but it can be smaller.

Given a graph family  $\mathcal{G}$  that is closed under taking subgraphs, we say that  $\mathcal{G}$  *admits a balanced edge separator of cost  $s$* :  $\mathbb{N} \rightarrow \mathbb{N}$  (where  $s$  is a non-decreasing function) if every graph  $G \in \mathcal{G}$  has a minimal  $(2/3)$ -balanced edge separator of cost at most  $s(|V(G)|)$ .

For a graph family that admits balanced edge separators of cost  $s$ , we construct a partition-based prover  $\mathbf{Prv}_{sep}^t$ , as follows.

**The decomposition tree.** Let  $d = \lfloor t/2 \rfloor - 1$ . We recursively decompose the input graph  $G$  into smaller subgraphs using balanced edge separators, until only components of size smaller than  $d$  remain; these components are the partition that will be used by our prover.

The recursive decomposition is represented by a labeled tree  $T$ , where

- Each vertex of  $T$  is a connected subgraph of  $G$ , and the root of  $T$  is  $G$  itself.
- Each non-leaf vertex  $C \subseteq G$  is labeled by a minimal  $(2/3)$ -balanced edge separator  $S_C \subseteq E(C)$  of cost  $\leq s(|V(C)|)$ .
- If a vertex  $C$  of  $T$  has size  $|V(C)| \geq d$ , then  $C$  is an inner vertex, and its children are the maximal connected components of the graph  $(V(C), E(C) \setminus S_C)$ .
- If a vertex  $C$  of  $T$  has size  $|V(C)| < d$ , then  $C$  is a leaf of  $T$ . For convenience, in this case we let  $S_C = \emptyset$ .

For each tree vertex  $C$ , let  $X_C \subseteq V(C)$  denote the nodes incident to an edge in  $S_C$ . The nodes in  $X_C$  will be boundary nodes in our partition.

It is easy to see that the depth of  $T$  is  $O(\log n)$ : at each inner vertex  $C$ , the edge set  $S_C$  is a  $(2/3)$ -balanced edge separator, so the children of  $C$  have size at most  $(2/3)|V(C)|$ . We state several additional properties of the decomposition, which will be needed for our  $t$ -PLS; the proofs of these properties appear in Appendix B.

In the sequel, we say that  $C$  is an *ancestor* of  $C'$  if  $C$  is on the path from  $C'$  to the root of  $T$ , including the case  $C = C'$ .

► **Lemma 7.** *Each tree vertex  $C$  is an induced subgraph of  $G$ , and the leafs  $L_1, \dots, L_k$  of  $T$  induce a partition  $V(L_1), \dots, V(L_k)$  of the vertices  $V(G)$ .*

For a node  $v \in V$ , let  $C(v)$  denote the leaf  $C$  of  $T$  such that  $v \in V(C)$ .

► **Lemma 8.** *For each  $v \in V$ ,  $C(v)$  is a connected subgraph of  $G$  of size at most  $d - 1$ .*

► **Corollary 9.** *The construction satisfies (P-1), with  $r = d - 1$ : for every node  $v \in V$ , the induced subgraph  $G[C(v)]$  is connected and has diameter at most  $d - 1 < t - 1$ .*

► **Lemma 10.** *For every two vertices  $C, C'$  of  $T$ ,*

- *If  $C$  is an ancestor of  $C'$  in  $T$ , then  $C' \subseteq C$ ;*
- *If  $C$  is not an ancestor of  $C'$  and  $C'$  is also not an ancestor of  $C$ , then  $C$  and  $C'$  are vertex-disjoint.*

Let  $S = \bigcup_{C \in T} S_C$ ,  $X = \bigcup_{C \in T} X_C$ . The nodes in  $X$  serve as the *boundary nodes* from Section 5.

► **Lemma 11.** *The construction satisfies condition (P-2): for every node  $u \in V$  and neighbor  $v \in N(u)$ , if  $C(u) \neq C(v)$ , then  $u, v \in X$ .*

Next, for a node  $v \in V$ , let

$$Up(v) = \bigcup_{\text{ancestors } C' \text{ of } C(v)} X_{C'}, \text{ and } \quad Down(x) = \{v \in V : x \in Up(v)\}.$$

In our  $t$ -PLS, every node  $v$  receives a piece of the certificate for each node in  $Up(v)$ . Because  $T$  has logarithmic depth, and the separator taken out at each vertex has size at most  $s(n)$ , we have:

► **Lemma 12.** *For each node  $v \in V$  we have  $|Up(v)| \leq s(n) \log n$ .*

On the other hand, it is also important that each node  $x \in X$  have at least  $\Omega(t)$  nodes to which we can give pieces of  $x$ 's certificate. If  $x \in X$ , then there is some inner vertex  $C$  of  $T$  such that  $x \in X_C$ . By definition,  $V(C) \subseteq Down(x)$ . Since inner vertices are connected subgraphs of size at least  $d$ , we have:

► **Lemma 13.** *For each  $x \in X$  we have  $|Down(x) \cap B_d(x)| \geq d$ .*

**The prover.** We now define the prover  $\mathbf{Prv}_{sep}^t$ , given a 1-PLS prover  $\mathbf{Prv}$  for some predicate  $\mathcal{P}$ . Fix a configuration  $(G, I) \in \mathcal{P}$ , and let  $a_v$  be the certificate assigned by  $\mathbf{Prv}$  to  $v \in V$ . Let  $n = |V|$ ,  $k = k(n)$  be the length of the certificates produced by  $\mathbf{Prv}$ , and let  $q = q(n, k, d)$  (from Section 3). Finally, let us fix an erasure code  $C_{n,k,d} = (Enc, Dec)$  over  $\mathbb{F}_q$ .

Using the decomposition tree  $T$  of  $G$ , our new prover assigns certificates as follows: first, the prover assigns a unique identifier  $ID(C) \in [n]$  to each leaf  $C$  of  $T$ . Then, the prover encodes the certificate of each node  $x \in X$  across the nodes in  $Down(x)$ : the prover computes the encoding  $Enc(a_x) = (c_x^1, \dots, c_x^n)$  of  $x$ 's certificate  $a_x$ , and assigns a unique

index  $\ell_x(u) \in [|Down(x)|]$  to each node  $u \in Down(x)$ . We refer to a triplet  $(x, \ell, c_x^\ell)$  as *the  $\ell$ -th piece of  $x$ 's certificate*. Each node  $v \in Down(x)$  receives the piece  $(x, \ell_x(v), c_x^{\ell_x(v)})$  (among other information).<sup>7</sup>

The certificate of node  $v \in V$  is  $(cid_v, s_v, P_v) \in [n] \times \{0, 1\} \times 2^{V \times [n] \times \mathbb{F}_q}$ , where

- $cid_v = ID(C(v))$ , and
- $s_v = 1$  iff  $v \in X$ ,
- $P_v$  is a collection of certificate pieces:  $P_v = \{(x, \ell_x(v), c_x^{\ell_x(v)}) : x \in Up(v)\}$ .

Since  $|Up(v)| \leq s(n) \log n$  and  $\log q = O(k(n)/d + \log n)$ , the size of each certificate is bounded by  $O(\log n + ((k(n)/d) + \log n)s(n) \log n) = \tilde{O}(k(n)s(n)/t)$ .

In Appendix B, we prove that the  $t$ -PLS  $(\mathbf{Prv}_{sep}^t, \mathbf{Ver}_{r,d}^t)$  is complete. Since we have already shown that  $\mathbf{Ver}_{r,d}^t$  preserves the soundness of the original verifier  $\mathbf{Ver}$ , together this proves Theorem 1 from Section 1.

## 8 A Partition-Based $t$ -PLS for Graph Families with an Excluded Minor

In this section we show that graph families with an excluded constant-sized minor admit a partition into components  $C_0, \dots, C_m$  of size  $r = O(t)$ , such that the boundary  $Y_i \subseteq C_i$  of each component  $C_i$  satisfies  $|Y_i|/|C_i| = O(\Delta/\sqrt{r})$ . (Here,  $\Delta$  is the maximum degree in the graph.) This allows the prover to split the certificate of each boundary node  $y \in Y_i$  into roughly  $\sqrt{r}/\Delta$  pieces, and assign them to the nodes of  $C_i$ , so that each node receives only a single certificate piece in total.

Our construction is based on the  *$r$ -region decomposition* of Frederickson [9], which takes a planar graph  $G = (V, E)$  and produces  $O(n/r)$  sets of nodes,  $R_1, \dots, R_m \subseteq V$ , called *regions*. The regions are not necessarily vertex-disjoint; nodes that appear in more than one region are called *border nodes*.<sup>8</sup> The regions  $R_1, \dots, R_m$  satisfy:

- For each edge  $\{u, v\} \in E$ , there is some region  $R_i$  such that  $u, v \in R_i$ ,
- $|R_i| \leq r$  for each  $i$ , and
- Each region contains at most  $O(\sqrt{r})$  border nodes:  $|R_i \cap \bigcup_{j \neq i} R_j| = O(\sqrt{r})$  for each  $i$ .

We note that  $G$  need not be a connected graph for Frederickson's construction, and the regions  $R_1, \dots, R_m$  need not induce connected subgraphs.

Although Frederickson originally stated his results only for planar graphs, the  $r$ -region decomposition from [9] relies only on the existence of a *balanced weighted vertex separator of size  $O(\sqrt{n})$*  (we omit the definition of this object here, as it is not needed directly for our construction). Later, [1] showed that any graph family with a constant-size excluded minor  $H$  admits a balanced weighted separator of size  $O(c_H \sqrt{n})$ , where  $c_H$  is a constant depending on the minor  $H$ . Thus, Frederickson's  $r$ -region decomposition applies to any graph family with an excluded minor of constant size.<sup>9</sup>

The  $r$ -region decomposition is an excellent starting point for constructing a partition for our prover, but there are two issues we need to solve:

<sup>7</sup> Note that while the encoding  $Enc(a_x)$  of  $x$ 's certificate comprises  $n$  pieces, only the first  $|Down(x)|$  pieces will actually be used, and the rest are not given to any graph node. This is fine: because  $|Down(x)| \geq d$  (Lemma 13), we can still recover  $a_x$  from the pieces that were actually used.

<sup>8</sup> In [9] these nodes are called *boundary nodes*, but here we refer to them as *border nodes*, as we use *boundary nodes* to mean something different in the context of Sections 5, 6.

<sup>9</sup> In fact, Frederickson's decomposition can be extended to any graph family with *polynomial expansion*, as it is known that such graph families have sublinear vertex separators. This allows us to extend the construction given here to any graph family with polynomial expansion, yielding a  $t$ -PLS with that saves a factor of  $O(\Delta/t^\delta)$  for some  $\delta \in (1/2, 1)$ . The details are deferred to the full version of the paper.

- The subgraph  $G[R_i]$  induced by a region  $R_i$  is not necessarily connected, and
- Regions can be arbitrarily small, so even though each region has size at most  $r$  and contains  $O(\sqrt{r})$  border nodes, the ratio between the size of the region and the number of border nodes in it is unbounded. This means there may not be enough nodes inside the region to allow us to partition the certificates of the border nodes across them.

To address these issues, we start from an  $r$ -region decomposition of the graph, and show that we can carve out at least one component  $C$  of size  $O(r)$ , which is connected and contains at most  $O(|C|/\sqrt{r})$  border nodes. We recurse on the connected components that remain after  $C$  is removed, until we have covered the original graph in its entirety.

► **Lemma 14.** *Let  $\mathcal{G}$  be a family of graphs that exclude a fixed minor  $H$ . Then there exists a constant  $\alpha \in \mathbb{R}^+$  such that for each  $r \leq n$ , every graph  $G \in \mathcal{G}$  admits a partition  $C_0, \dots, C_m$  of  $V(G)$ , along with border sets  $X_0 \subseteq C_0, \dots, X_m \subseteq C_m$ , such that*

**(R-1)** *Condition (P-1) is satisfied: for each  $i$ ,  $G[C_i]$  is a connected subgraph and  $|C_i| \leq r$ .*

**(R-2)** *For each  $i \in [m]$ ,  $u \in C_i$  and  $v \in V \setminus C_i$  such that  $\{u, v\} \in E$ , either  $u \in \bigcup_{j < i} X_j$  or  $v \in \bigcup_{j < i} X_j$  (or both).<sup>10</sup>*

**(R-3)** *For each  $i \in [m]$  we have  $|C_i|/|X_i| \geq \alpha\sqrt{r}$  (or  $|X_i| = 0$ ).*

**Proof sketch.** We describe how the partition is constructed; the proof that the conditions of the lemma are satisfied is deferred to Appendix C.

Let  $c_1, c_2 \geq 1$  be the constants from Frederickson's construction, so that for every graph over at least  $r$  vertices, there is an  $r$ -region decomposition comprising at most  $c_1 n/r$  regions, each of size at most  $r$ , and each containing at most  $c_2 \sqrt{r}$  border nodes.

Let  $\alpha = 1/(c_1 c_2)$ . We build our partition recursively, by carving out a sequence of components  $C_0, \dots, C_m$ , each with a border set  $X_i \subseteq C_i$ , satisfying the conditions of the lemma. Suppose we have already found and removed the first  $i \geq 0$  components, and let  $G_i = G[V_i]$  be the remaining graph, where  $V_i = V \setminus \bigcup_{j < i} C_j$ .

If  $|V_i| < r$ , we cannot apply the region decomposition to  $G_i$ , as it is too small. In this case we let  $C_i$  be some maximal connected component of  $G_i$ , and let  $X_i = \emptyset$ . It is easy to see that the conditions of the lemma are satisfied.

If  $|V_i| \geq r$ , then let  $R_1, \dots, R_m$  be an  $r$ -region decomposition of  $G_i$ , comprising at most  $c_1 |V_i|/r$  regions, each of size at most  $r$  and containing at most  $c_2 \sqrt{r}$  border nodes (here, "border nodes" means nodes that appear in more than one region  $R_1, \dots, R_m$ ; we do not consider regions removed in previous steps of the construction). Since we have at most  $c_1 |V_i|/r$  regions that together cover all nodes of  $V_i$ , and since each region contains at most  $r$  nodes, there is some region  $R^*$  such that  $r/c_1 \leq |R^*| \leq r$ . Let  $R^- = \bigcup_{R' \neq R^*} R'$  be the union of all the regions except  $R^*$ , and let  $W_1, \dots, W_s$  be the maximal connected components of the induced subgraph  $G_i[R^*]$ . We claim that there is some maximal component  $W_j$  that has

$$\frac{|W_j|}{|W_j \cap R^-|} \geq \frac{\sqrt{r}}{c_1 c_2}, \quad (3)$$

otherwise the total number of border nodes in  $R^*$  would be too large:

$$|R^* \cap R^-| = \sum_{j=1}^s |W_j \cap R^-| > \sum_{j=1}^s |W_j| \cdot \frac{c_1 c_2}{\sqrt{r}} = \frac{c_1 c_2}{\sqrt{r}} |R^*| \geq \frac{c_1 c_2}{\sqrt{r}} \frac{r}{c_1} = c_2 \sqrt{r},$$

a contradiction to the fact that every region contains at most  $c_2 \sqrt{r}$  border nodes.

<sup>10</sup>Note that this condition is weaker than (P-2).

We define the component  $C_i$  to be some maximal connected component  $W_j$  satisfying (3), and we let  $X_i = W_j \cap R^-$ , the border nodes in  $W_j$ . The proof that the conditions of the lemma are satisfied appears in Appendix C. ◀

**The prover.** We are now ready to define the prover  $\mathbf{Prv}_{minor}^t$ , given a 1-PLS  $(\mathbf{Prv}, \mathbf{Ver})$  for some predicate  $\mathcal{P}$ .

Let  $\mathcal{G}$  be a family of graphs with an excluded minor  $H$ , and let  $\alpha$  be the constant from Lemma 14. Let  $r = \lfloor t/2 \rfloor - 1$  and let  $d = \alpha\sqrt{r}/(\Delta + 1)$ . Let  $C_{n,k,d} = (Enc_{n,k,d}, Dec_{n,k,d})$  where  $k = k(n)$  is the certificate size of the original PLS and let  $q = q(n, k, d)$  be the prime from Section 3.

Given a configuration  $(G, I) \in \mathcal{G}_{\mathcal{I}}$  over  $n$  nodes with maximum degree  $\Delta$ , the prover  $\mathbf{Prv}_{minor}^t$  computes the certificates  $\{a_v\}_{v \in V}$  assigned by  $\mathbf{Prv}$  to  $(G, I)$ , and the partition  $C_0, \dots, C_m, X_0, \dots, X_m$  from Lemma 14.

For each component  $C_i$  with border  $X_i \subseteq C_i$ , let  $Y_i = X_i \cup N(X_i)$ . The prover partitions  $C_i$  into  $|Y_i|$  sets,  $\{D_y\}_{y \in Y_i}$ , each of size at least  $d = \alpha\sqrt{r}/(\Delta + 1)$ . This is possible, because

$$|Y_i| \leq (\Delta + 1)|X_i| \leq (\Delta + 1)|C_i|/(\alpha\sqrt{r}).$$

For each node  $y \in Y_i$ , the prover computes the encoding  $Enc_{n,k,d}(a_y) = (c_y^1, \dots, c_y^n)$  of  $a_y$  into  $n$  pieces. The prover also assigns a unique index  $\ell_y(v) \in [d]$  to each node  $v \in D_y$ , and node  $v \in D_y$  will then receive the piece  $(y, \ell_y(v), c_y^{\ell_y(v)})$  (among other information). The certificate of node  $v \in C_i$  is given by  $(i, s_v, P_v)$ , where

- $s_v = 1$  iff  $v \in Y_i$ ,
- $P_v = \left\{ (y, \ell_y(v), c_y^{\ell_y(v)}) : v \in D_y \right\}$ . This consists of a single certificate piece, because the sets  $D_y, D_{y'}$  are disjoint for  $y \neq y'$ .

The certificate size is bounded by  $O(\log n + \log q) = O((k/d) + \log n) = \tilde{O}(k\Delta/\sqrt{t})$ .

This completes the definition of  $\mathbf{Prv}_{minor}^t$ . The proof that the  $t$ -PLS  $(\mathbf{Prv}_{minor}^t, \mathbf{Ver}_{r,d,q}^t)$  is complete appears in Appendix C. Together, this proves Theorem 2 from Section 1.

---

## References

- 1 Noga Alon, Paul D. Seymour, and Robin Thomas. A separator theorem for graphs with an excluded minor and its applications. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, pages 293–299, 1990.
- 2 Keren Censor-Hillel, Ami Paz, and Mor Perry. Approximate proof-labeling schemes. In *Structural Information and Communication Complexity - 24th International Colloquium (SIROCCO)*, volume 10641, pages 71–89, 2017.
- 3 Laurent Feuilloley and Pierre Fraigniaud. Survey of distributed decision. *Bull. EATCS*, 119, 2016.
- 4 Laurent Feuilloley and Pierre Fraigniaud. Error-sensitive proof-labeling schemes. In *31st International Symposium on Distributed Computing*, volume 91, pages 16:1–16:15, 2017.
- 5 Laurent Feuilloley, Pierre Fraigniaud, and Juho Hirvonen. A hierarchy of local decision. *Theor. Comput. Sci.*, 856:51–67, 2021.
- 6 Laurent Feuilloley, Pierre Fraigniaud, Juho Hirvonen, Ami Paz, and Mor Perry. Redundancy in distributed proofs. *Distributed Computing*, 34(2):113–132, 2021.
- 7 Laurent Feuilloley, Pierre Fraigniaud, Pedro Montealegre, Ivan Rapaport, Éric Rémila, and Ioan Todinca. Compact distributed certification of planar graphs. *Algorithmica*, 83(7):2215–2244, 2021.
- 8 Pierre Fraigniaud, Boaz Patt-Shamir, and Mor Perry. Randomized proof-labeling schemes. *Distributed Computing*, 32, 2019.



- 9 Greg N. Frederickson. Fast algorithms for shortest paths in planar graphs, with applications. *SIAM Journal on Computing*, 16(6):1004–1022, 1987.
- 10 Mika Göös and Jukka Suomela. Locally checkable proofs in distributed computing. *Theory Comput.*, 12(1):1–33, 2016.
- 11 Gillat Kol, Rotem Oshman, and Raghuvansh R. Saxena. Interactive distributed proofs. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 255–264, 2018.
- 12 Liah Kor, Amos Korman, and David Peleg. Tight bounds for distributed minimum-weight spanning tree verification. *Theory Comput. Syst.*, 53(2):318–340, 2013.
- 13 Janne H. Korhonen and Jukka Suomela. Towards a complexity theory for the congested clique. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 163–172, 2018.
- 14 Amos Korman and Shay Kutten. Distributed verification of minimum spanning trees. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 26–34, 2006.
- 15 Amos Korman and Shay Kutten. On distributed verification. In *Distributed Computing and Networking, 8th International Conference, ICDCN*, volume 4308, pages 100–114, 2006.
- 16 Amos Korman, Shay Kutten, and David Peleg. Proof labeling schemes. *Distributed Comput.*, 22(4):215–233, 2010.
- 17 Moni Naor, Merav Parter, and Eylon Yogev. The power of distributed verifiers in interactive proofs. In *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1096–115, 2020.
- 18 Rafail Ostrovsky, Mor Perry, and Will Rosenbaum. Space-time tradeoffs for distributed verification. In *International Colloquium on Structural Information and Communication Complexity*, pages 53–70. Springer, 2017.
- 19 Boaz Patt-Shamir and Mor Perry. Proof-labeling schemes: Broadcast, unicast and in between. In *Stabilization, Safety, and Security of Distributed Systems - 19th International Symposium (SSS)*, volume 10616, pages 1–17, 2017.
- 20 Ron M. Roth. *Introduction to coding theory*. Cambridge University Press, 2006.
- 21 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.
- 22 Jukka Suomela. Survey of local algorithms. *ACM Comput. Surv.*, 45(2):24:1–24:40, 2013.

## A Soundness of the Partition-Based Verifier

Fix a certificate assignment  $\{(cid_v, s_v, P_v) : v \in V\}$  that is accepted by all nodes under  $\mathbf{Ver}_{r,d}^t$ , and let us construct a certificate assignment  $\{a_v : v \in V\} \subseteq \{0, 1\}^k$  that is accepted by all nodes under the original verifier  $\mathbf{Ver}$ : for each node  $v \in V$ , let  $z_v \in \tilde{C}(v)$  be the node that has the smallest ID in  $\tilde{C}(v)$ . Then we define  $a_v = \tilde{a}_{z_v}(v)$ , where  $\tilde{a}_{z_v}$  is the certificate assignment found by  $z_v$  during the verification phase (i.e., the certificate assignment that makes  $\mathbf{Ver}$  accept at all nodes in  $\tilde{C}(z_v)$ ). We now prove that our stitched-together certificate assignment  $\{a_v\}_{v \in V}$  is indeed accepted by  $\mathbf{Ver}$  at all nodes.

First, it is not hard to see that if  $u$  is in the component  $\tilde{C}(v)$ , then  $u$  and  $v$  agree on the structure of the component:

► **Lemma 15.** *If  $u \in \tilde{C}(v)$  then  $\tilde{C}(u) = \tilde{C}(v)$ ,  $\text{Boundary}(u) = \text{Boundary}(v)$ , and  $\text{Adj}(u) = \text{Adj}(v)$ .*

**Proof.** Node  $v$  verifies that  $\tilde{C}(v)$  induces a connected component of size at most  $r$  inside  $B_{t-1}(v)$ . Since  $u \in \tilde{C}(v)$ , for every  $w \in \tilde{C}(v)$  there is a path of length at most  $r$  between  $u$  and  $w$ ; therefore  $\tilde{C}(v) \subseteq B_r(u) \subseteq B_{t-1}(u)$ , and since all nodes  $w \in \tilde{C}(v)$  have  $cid_w = cid_v = cid_u$ , it follows that  $\tilde{C}(v) \subseteq \tilde{C}(u)$ . In particular,  $v \in \tilde{C}(u)$ , and the same reasoning now shows that  $\tilde{C}(u) \subseteq \tilde{C}(v)$  as well. Together we have  $\tilde{C}(v) = \tilde{C}(u)$ .

## 21:18 Explicit Space-Time Tradeoffs for PLS in Graphs with Small Separators

Now consider  $Adj(v)$ . By definition,  $w \in Adj(v)$  iff  $w \in (B_{t-1}(v) \cap N(\tilde{C}(v))) \setminus \tilde{C}(v)$ , that is,  $w \notin \tilde{C}(v)$  and there is a node  $x \in \tilde{C}(v)$  such that  $w \in N(x)$ . Node  $v$  has verified that  $\tilde{C}(v)$  induces a connected component of size at most  $r$  inside  $B_{t-1}(v)$ , so there is a path of length at most  $r < t - 1$  between node  $x$  and node  $u$ , implying that  $w \in N(x) \subseteq B_{t-1}(u)$ . Together with the fact that  $\tilde{C}(u) = \tilde{C}(v)$ , which we showed above, we see that  $w \in (B_{t-1}(u) \cap N(\tilde{C}(u))) \setminus \tilde{C}(u) = Adj(u)$ . Containment in the other direction is similar, with the roles of  $u$  and  $v$  exchanged.

Finally, what we showed above implies that  $Boundary(u) = Boundary(v)$ : by definition,  $Boundary(u) = (\tilde{C}(u) \setminus Inner(u)) \cup Adj(u)$ . We already showed that  $\tilde{C}(u) = \tilde{C}(v)$  and  $Adj(u) = Adj(v)$ . It is immediate that  $Inner(u) = Inner(v)$ , as  $Inner(u)$  comprises all the inner nodes in  $\tilde{C}(u)$ , and similarly for  $v$ . The claim follows.  $\blacktriangleleft$

Next, to show that we stitched together the certificates of the various components in a consistent manner, it is crucial to prove that all nodes agree on the certificates they reconstructed for all boundary nodes they have in common:

► **Lemma 16.** *Let  $u, v \in V$ , and let  $x \in Boundary(u) \cap Boundary(v)$ . Then  $\tilde{a}_u(x) = \tilde{a}_v(x)$ .*

**Proof.** We show that there is a path  $\pi_u$  between  $u$  and  $x$ , and a path  $\pi_v$  between  $v$  and  $x$ , such that all nodes  $w$  on both paths have  $x \in Boundary(w)$ . Since every node  $w$  verifies that it agrees with its neighbors on certificates they reconstructed for the nodes in  $Boundary(w)$ , this implies the claim.

We prove the existence of the path  $\pi_u$  between  $u$  and  $x$ ;  $\pi_v$  is similar. Consider first the case where  $x \in \tilde{C}(u)$ . Node  $u$  has verified that  $\tilde{C}(u)$  is connected and that  $|\tilde{C}(u)| \leq r$ , so there is a path  $\pi_u = u_0, \dots, u_\ell$  from  $u$  to  $x$ , such that  $\ell \leq r$ ,  $u_0 = u, u_\ell = x$ , and  $u_0, \dots, u_\ell \in \tilde{C}(u)$ . By Lemma 15, each path node  $u_i$  has  $Boundary(u_i) = Boundary(u)$ ; and since  $x \in Boundary(u)$ , this shows that every node  $u_i$  in the path has  $x \in Boundary(u_i)$ .

Now suppose that  $x \notin \tilde{C}(u)$ . Then we must have  $x \in Adj(u)$ , as  $x \in Boundary(u) = (\tilde{C}(u) \setminus Inner(u)) \cup Adj(u)$ . By definition of  $Adj(u)$ , node  $x$  has a neighbor  $y \in N(x) \cap \tilde{C}(u)$ . Since  $\tilde{C}(u)$  is connected and  $|\tilde{C}(u)| \leq r$ , there is a path  $\pi_u = u_0, \dots, u_\ell, u_{\ell+1}$  from  $u$  to  $x$ , such that  $\ell \leq r$ ,  $u_0 = u, u_\ell = y, u_{\ell+1} = x$ , and  $u_0, \dots, u_\ell \in \tilde{C}(u)$ . By Lemma 15, each path node  $u_i$  where  $i \leq \ell$  has  $Adj(u_i) = Adj(u)$  and hence  $x \in Adj(u_i) \subseteq Boundary(u_i)$ . And of course the last node,  $u_{\ell+1} = x$ , also has  $x \in Boundary(x)$ , as  $x \in \tilde{C}(x) \setminus Inner(x)$ .  $\blacktriangleleft$

Finally, we show that the “representative”  $z_v$ , whose certificate assignment  $\tilde{a}_z$  we used to define  $a_v$ , in fact has  $v$  in its component  $\tilde{C}(z_v)$ :

► **Lemma 17.** *For each  $v \in V$  we have  $v \in \tilde{C}(z_v)$ .*

**Proof.** Let  $z = z_v$ . By definition,  $z \in \tilde{C}(v)$ , so by Lemma 15,  $\tilde{C}(v) = \tilde{C}(z)$ . Since  $v \in \tilde{C}(v)$ , we also have  $v \in \tilde{C}(z)$ .  $\blacktriangleleft$

We can now conclude that under the original verifier **Ver**, all nodes accept the certificate assignment  $a$  that we defined:

► **Lemma 18.** *For each node  $v \in V$  we have  $\mathbf{Ver}(v, I(v), N(v), a_v, \{a_u\}_{u \in N(v)}) = 1$ .*

**Proof.** Let  $v \in V$ , and let  $z = z_v \in \tilde{C}(v)$ . Since  $v \in \tilde{C}(z)$  (Lemma 17), node  $z$  verifies that  $v$  accepts  $\tilde{a}_z$ : formally, node  $z$  verifies that  $\mathbf{Ver}(v, I(v), N(v), \tilde{a}_z(v), \{\tilde{a}_z(u)\}_{u \in N(v)}) = 1$ . We now prove that the certificate assignments  $\tilde{a}_z$  and  $a$  agree on the certificates of  $v$  and all of its neighbors: that is,  $\tilde{a}_z(u) = a_u$  for each node  $u \in \{v\} \cup N(v)$ .

- For  $v$  itself, we defined  $a_v = \tilde{a}_z(v)$ .
- For a neighbor  $u \in N(v) \cap \tilde{C}(v)$  inside  $v$ 's component, Lemma 15 shows that  $\tilde{C}(u) = \tilde{C}(v)$ . In particular,  $z_u = z_v = z$  (the node with the smallest ID in the component), and therefore  $a_u = \tilde{a}_{z_u}(u) = \tilde{a}_z(u)$ .
- For a neighbor  $u \in N(v) \setminus \tilde{C}(v)$  outside  $v$ 's component, node  $u$  must be a boundary node ( $s_u = 1$ ), otherwise condition (V-2) would cause  $u$  to reject. Also,  $u \in \text{Adj}(v)$ , by definition of  $\text{Adj}(v)$ . Lemma 15 shows that  $\text{Adj}(z) = \text{Adj}(v)$ . and hence  $u \in \text{Adj}(z) \subseteq \text{Boundary}(z)$ . Now let  $z' = z_u$  be the node “responsible” for  $u$ ; we defined  $a_u = \tilde{a}_{z'}(u)$ . By Lemma 17 we have  $u \in \tilde{C}(z')$ , and since  $u$  is a boundary node,  $u \in \text{Boundary}(z')$ . We now have that  $u \in \text{Boundary}(z) \cap \text{Boundary}(z')$ , and therefore, by Lemma 16,  $a_u = \tilde{a}_{z'}(u) = \tilde{a}_z(u)$ .

Since node  $z$  has verified that  $\mathbf{Ver}(v, I(v), N(v), \tilde{a}_z(v), \{\tilde{a}_z(u)\}_{u \in N(v)}) = 1$ , and we have shown that  $a_u = \tilde{a}_z(u)$  for every  $u \in \{v\} \cup N(v)$ , we immediately see that the verifier at node  $v$  accepts the assignment  $a$ , i.e.,  $\mathbf{Ver}(v, I(v), N(v), a_v, \{a_u\}_{u \in N(v)}) = 1$ . ◀

## B Missing Proofs from Section 7

**Proof of Lemma 7.** It is easy to see that at each level of the tree, the vertices  $C_1, \dots, C_k$  at that level induce a partition  $V(C_1), \dots, V(C_k)$  of  $V(G)$ , because the children of each vertex are the connected components that remain after removing some edges (but no vertices are removed). Thus, in particular, the leaves of  $T$  induce a partition of  $V(G)$ .

Next we show that each vertex  $C$  is an induced subgraph of  $G$ , by induction on the distance from the root to  $C$ . The base case is immediate, because the root of the tree is  $G$  itself. For the induction step, suppose that  $C$  is an induced subgraph of  $G$ , and let  $D$  be a child of  $C$ . Suppose for the sake of contradiction that there is an edge  $\{u, v\} \in E(G)$ , such that  $u, v \in V(D)$ , but  $\{u, v\} \notin E(D)$ . Recall that by definition of the tree,  $D$  is a maximal connected component of the graph  $(V(C), E(C) \setminus S_C)$ . Since  $D \subseteq C$ , we have  $u, v \in V(C)$ , and since  $C$  is an induced subgraph of  $G$ , we must have  $\{u, v\} \in E(C)$ ; therefore  $\{u, v\}$  must have been removed in the recursive step at  $C$  as part of the edge separator, i.e.,  $\{u, v\} \in S_C$ . But since  $u, v$  are in the same connected component  $C$  of the graph  $(V(C), E(C) \setminus S_C)$ , there is some path  $\pi_{u,v}$  between  $u$  and  $v$  that does not include any edge from  $S_C$ . We argue that this contradicts the minimality of  $S_C$ , as taking out only the edges  $S_C \setminus \{\{u, v\}\}$  yields exactly the same maximal connected components as taking out all of  $S_C$ .

To see this, let us abuse notation slightly, and denote by  $D \setminus F$  the graph  $(V(D), E(D) \setminus F)$ . We claim that for any two nodes  $x, y \in D$ , node  $x$  is reachable from  $y$  in  $D \setminus S_C$  iff it is reachable from  $y$  in  $D \setminus (S_C \setminus \{\{u, v\}\})$ : clearly, if  $x$  is reachable from  $y$  in  $D \setminus S_C$ , then it is also reachable from  $y$  in  $D \setminus (S_C \setminus \{\{u, v\}\})$ . For the other direction, if  $x$  is reachable from  $y$  in  $D \setminus (S_C \setminus \{\{u, v\}\})$ , then we can also reach  $x$  from  $y$  by a path that does not include any edge of  $S_C$ : given a path that avoids all edges in  $S_C \setminus \{\{u, v\}\}$  but does include  $\{u, v\}$ , we simply replace  $\{u, v\}$  by the path  $\pi_{u,v}$  whose existence we showed above, which does not include any edges of  $S_C$ . ◀

**Proof of Lemma 8.** Since  $C(v)$  is a leaf of the decomposition tree,  $|C(v)| \leq d - 1$ . Also, every vertex of the decomposition tree is a connected subgraph of  $G$ . ◀

**Proof of Lemma 10.** If  $C$  is an ancestor of  $C'$ , an easy induction on the distance between  $C$  and  $C'$  in  $T$  shows that  $C'$  is a subgraph of  $C$ : the children of every inner vertex  $D$  of  $T$  are subgraphs of  $D$ . If  $C, C'$  are not ancestors of one another, then let  $D$  be the lowest common ancestor of  $C$  and  $C'$ . We have  $D \neq C, D \neq C'$ . Let  $D'$  be the child of  $D$  that is an ancestor of  $C$ , and let  $D''$  be the child of  $D$  that is an ancestor of  $C'$ . Then  $D', D''$  are

vertex-disjoint, as they are both maximal connected components of the graph obtained from  $D$  by removing the edges in  $S_D$ . Since we have already shown that  $C \subseteq D'$  and  $C' \subseteq D''$ , it follows that  $C$  and  $C'$  are also vertex-disjoint.  $\blacktriangleleft$

**Proof of Lemma 11.** Let  $D$  be the lowest common ancestor of  $C(u)$  and  $C(v)$  in  $T$ , and let  $D_u, D_v$  be the children of  $D$  that are ancestors of  $C(u), C(v)$ , respectively. Of course,  $D_u \neq D_v$ . By Lemma 10, we have  $C(u) \subseteq D_u \subseteq D$  and  $C(v) \subseteq D_v \subseteq D$ , and hence  $u \in D_u$ ,  $v \in D_v$ , and  $u, v \in D$ . And since  $D$  is an induced subgraph of  $G$  (Lemma 7), and we know that  $\{u, v\} \in E$  (as  $v \in N(u)$ ), it must be that  $\{u, v\} \in E(D)$ . But  $D_u$  and  $D_v$  are maximal connected components of  $(V(D), E(D) \setminus S_D)$ , and so the edge  $\{u, v\}$  must be in  $S_D$ , otherwise  $u, v$  would be in the same child of  $D$ . It follows that  $u, v \in X_D \subseteq X$ .  $\blacktriangleleft$

**Proof of Lemma 12.**  $T$  has depth  $O(\log n)$ , and at every ancestor  $C'$  of  $C(v)$  we have  $|X_{C'}| \leq s(|C'|) \leq s(n)$ . The claim follows.  $\blacktriangleleft$

**Proof of Lemma 13.** Since  $x \in X$ , there is some inner vertex  $C$  such that  $x \in X_C$ , and by definition, for every  $v \in C$  we have  $x \in Up(v)$ . Therefore  $V(C) \subseteq Down(x)$ . In particular, since  $C$  is a connected subgraph of size at least  $d$ , and since  $x \in X_C \subseteq V(C)$ , we have  $|V(C) \cap B_d(x)| \geq d$ . Therefore  $|Down(x) \cap B_d(x)| \geq |V(C) \cap B_d(x)| \geq d$ .  $\blacktriangleleft$

**Completeness of the prover  $\text{Prv}_{sep}^t$ .** To show that the honest prover  $\text{Prv}_{sep}^t$  causes all nodes to accept, we relate the values that nodes compute during their verification to the decomposition tree computed by the prover. We continue to refer to nodes as 'inner' or 'boundary' nodes, except that now, since we are working with the honest prover, we know that  $v$  is a boundary node (i.e.,  $s_v = 1$ ) iff  $v \in X$ .

► **Lemma 19.** For each inner node  $v$  we have  $\tilde{C}(v) = C(v)$ .

**Proof.** Recall that  $\tilde{C}(v)$  is the set of nodes  $u$  in  $B_{t-1}(v)$  that have  $cid_u = cid_v$ . By definition, the prover assigns  $cid_u = ID(C(u))$  to each node  $u$ . Therefore,  $cid_u = cid_v$  iff  $C(u) = C(v)$ , that is, iff  $u \in C(v)$ . It follows that  $\tilde{C}(v) = C(v) \cap B_{t-1}(v)$ . But by Lemma 8 we have  $C(v) \subseteq B_d(v)$ , implying that  $C(v) \cap B_{t-1}(v) = C(v)$ , and therefore,  $\tilde{C}(v) = C(v)$ .  $\blacktriangleleft$

► **Lemma 20.** Conditions (V-1) and (V-2) are satisfied at every node  $v \in V$ .

**Proof.** Fix  $v \in V$ , and let us verify that the conditions hold.

**V-1:** By Lemma 19 we have  $\tilde{C}(v) = C(v)$ , and since  $C(v)$  is a leaf of the decomposition tree, it is a connected component of size  $|C(v)| \leq r = d - 1$ . Thus,  $\tilde{C}(v)$  induces a connected component of size at most  $r$  in  $B_{t-1}(v)$ .

**V-2:** Let  $u \in N(v)$  be an inner node ( $s_u = 0$ ). Then  $u \notin X$ , and by Lemma 11, every neighbor of  $u$  must be in the same component,  $C(u) = C(v)$ . This implies that  $cid_u = cid_v$ .  $\blacktriangleleft$

Next we show that the checks associated with the reconstruction of the boundary nodes' certificates succeed:

► **Lemma 21.** Let  $v \in V$ . For each  $x \in \text{Boundary}(v)$  we have  $|A_v(x)| \geq d$ , and all piece indices that appear in  $A_v(x)$  are distinct. Moreover, the reconstructed certificate  $\tilde{a}_v(x)$  is the true certificate  $a_x$ .

**Proof.** First, note that  $x$  is a boundary node ( $s_x = 1$ ): if  $x \in \tilde{C}(v) \setminus \text{Inner}(v)$  then this is immediate by definition of  $\text{Inner}(v)$ . Otherwise, we have  $x \in \text{Adj}(v)$ , meaning that  $x \notin \tilde{C}(v)$  and there is some neighbor  $y \in \tilde{C}(v)$  such that  $x \in N(y)$ . Inner nodes cannot have neighbors

from a different component (by (V-2), which as we proved in Lemma 20 is satisfied). Therefore  $x$  is a boundary node, and the prover disperses pieces of  $a_x$  across all the nodes in  $Down(x)$ .

By Lemma 13, there are at least  $d$  nodes in  $Down(x) \cap B_d(x)$ . All these nodes are in  $B_{t-1}(v)$ : because  $|\tilde{C}(v)| \leq r = d - 1$  (Lemma 8) and  $x \in N(\tilde{C})$ , we have  $x \in B_d(v)$ , and therefore  $B_d(x) \subseteq B_{2d}(v) \subseteq B_{t-1}(v)$  (as  $d = \lfloor t/2 \rfloor - 1$ ). Consequently,  $A_v(x)$  includes at least  $d$  distinct pieces of  $a_x$ , and the decoder  $D$  reconstructs  $a_x$  successfully. ◀

► **Lemma 22.** *The verifier  $\mathbf{Ver}_{r,d}^t$  accepts at all nodes  $v \in V$ .*

**Proof.** We have already shown that conditions (V-1) and (V-2) are satisfied at all nodes, and that the reconstruction step succeeds. It remains to prove that for every node  $v \in V$ , there is some certificate assignment  $\tilde{a}_v$  that agrees with the certificates that  $v$  reconstructed for each boundary node  $x \in Boundary(v)$ , and which causes  $\mathbf{Ver}$  to accept at all nodes in  $C(v)$ . Naturally, we define  $\tilde{a}_v(u) = a_u$  for each  $u \in All(v)$ . By Lemma 21, this indeed agrees with the certificates that  $v$  reconstructed for its boundary nodes.

Since we know that  $\mathbf{Ver}(u, I(u), N(u), a_u, \{a_w\}_{w \in N(u)}) = 1$  for all  $u \in V$ , we get that for every  $u \in C(v)$ , we have  $\mathbf{Ver}(u, I(u), N(u), \tilde{a}_v(u), \{\tilde{a}_v(w)\}_{w \in N(u)}) = 1$ . ◀

## C Missing Proofs from Section 8

**Conditions (R-1)–(R-3) in the case where  $|V_i| < r$ .** We prove that  $C_i, X_i$  satisfy the requirements:

- (R-1) (P-1): clearly,  $|C_i| \leq |V_i| < r$ , and  $C_i$  is a maximal connected component of  $G_i$ , which is an induced subgraph of  $G$ . Therefore  $G[C_i]$  is a connected subgraph of size at most  $r$ , as required.
- (R-2) Let  $u \in C_i$  and  $v \in V \setminus C_i$  be such that  $\{u, v\} \in E$ . It cannot be that  $v \in V_i$ , because  $V_i$  is an induced subgraph of  $G$ , so this would imply that  $\{u, v\} \in E[G_i]$ , contradicting the fact that  $C_i$  is a maximal connected component of  $G_i$ . Thus,  $v \notin V_i$ , meaning that  $v \in C_j$  for some  $j < i$ . In addition, since  $C_j \cap V_i = \emptyset$  and  $u \in C_i \subseteq V_i$ , we have  $u \notin C_j$ . Because  $C_j$  satisfies the requirements of the lemma, this implies that either  $u \in \bigcup_{j' < j} X_{j'}$  or  $v \in \bigcup_{j' < j} X_{j'}$ . And since  $j < i$ , the requirement is satisfied for  $C_i$  as well.
- (R-3) We have  $|X_i| = 0$ .

**Conditions (R-1)–(R-3) in the case where  $|V_i| \geq r$ .** Let us prove that the requirements are satisfied:

- (R-1) We have  $|C_i| \leq |R^*| \leq r$ , and since  $C_i$  is a maximal connected component of  $G_i[R^*]$ , and  $G_i$  is itself an induced subgraph of  $G$ , we see that  $G[C_i]$  is a connected subgraph of size at most  $r$ , as required.
- (R-2) Let  $u \in C_i$  and  $v \in V \setminus C_i$  such that  $\{u, v\} \in E$ . There are two cases:
  - $v \in G_i$ : note that since  $C_i$  is a maximal connected component of  $G_i[R^*]$  and  $G_i$  is itself an induced subgraph of  $G$ , we must have  $v \notin R^*$ , otherwise the presence of the edge  $\{u, v\} \in E$  would mean that  $u, v$  are both in the same maximal connected component. Thus, there is some region  $R' \neq R$  that covers the edge  $\{u, v\}$ , that is,  $u, v \in R'$ . But this implies that  $u \in R^* \cap R'$ , so  $u$  is a border node, and  $u \in X_i$ .
  - $v \notin G_i$ : then since  $G_i$  is the graph induced by  $V_i = V \setminus \bigcup_{i' < i} C_{i'}$ , there is some component  $C_{i'}$ ,  $i' < i$ , such that  $v \in C_{i'}$ . Since  $C_{i'}$  satisfies (R-2), we have either  $u \in \bigcup_{i'' < i'} X_{i''}$  or  $v \in \bigcup_{i'' < i'} X_{i''}$ , and since  $i' < i$ , this implies that (R-2) is satisfied for  $C_i$  as well.
- (R-3) Holds by choice of  $C_i$  as a maximal connected component  $W_j$  that satisfies (3).

## 21:22 Explicit Space-Time Tradeoffs for PLS in Graphs with Small Separators

**Completeness of the prover  $\mathbf{Prv}_{minor}^t$ .** Suppose that  $(G, I) \in \mathcal{P}$ , and let us show that all nodes accept the honest prover's certificates. As in Section 7, we do so by relating the values that nodes compute during their verification to the partition the prover computed.

► **Lemma 23.** *For each node  $v \in C_i$  we have  $\tilde{C}(v) = C_i$ .*

**Proof.** By definition, the prover assigns the same index  $cid_u = i$  to all nodes  $u \in C_i$ . Therefore  $\tilde{C}(v) = C(v) \cap B_{t-1}(v)$ . In addition, we know that  $|C_i| \leq r < t - 1$  and that  $G[C_i]$  is connected, so  $C_i \subseteq B_{t-1}$ , implying that  $C_i \cap B_{t-1}(v) = C_i$ , and therefore,  $\tilde{C}(v) = C_i$ . ◀

► **Corollary 24.** *Condition (V-1) is satisfied at all nodes.*

**Proof.** For each node  $v$ , Lemma 23 implies that  $\tilde{C}(v) = C_i$  for some part  $C_i$  in the partition. By the conditions of Lemma 14,  $|C_i| \leq r$  and  $G[C_i]$  is connected, as required. ◀

► **Lemma 25.** *Condition (V-2) is satisfied at all nodes.*

**Proof.** Fix  $v \in V$ , and let  $u \in N(v)$  be such that  $cid_u \neq cid_v$ . By Lemma 23, there are two distinct parts  $C_i \neq C_j$  such that  $u \in C_i$  and  $v \in C_j$ . Thus, from condition (R-2) of Lemma 14, either  $u \in X_i$  or  $v \in X_j$  (or both). Since the prover marks all border nodes and also their neighbors, we therefore have  $s_u = s_v = 1$ . ◀

Finally, we show that each boundary node has enough certificate-pieces in its vicinity:

**Lemma 21, for the current construction.** As in the proof of Lemma 21 for  $\mathbf{Prv}_{sep}^t$ , for any  $v \in V$  and  $x \in \text{Boundary}(v)$ , we must have  $s_x = 1$ . (This part of the proof depends only on the verifier, which is the same in both proof systems.) Thus, there is some  $i$  such that  $x \in Y_i$ , and the prover distributes pieces of  $x$ 's proof across the nodes in  $D_i$ . Recall that  $|D_i| = d$ . Moreover,  $D_i \subseteq C_i \subseteq B_r(x)$ , and  $x \in B_{r+1}(v)$  (because  $x \in \text{Boundary}(v) \subseteq \tilde{C}(v) \cup N(\tilde{C}(v))$  and  $\tilde{C}(v)$  is a connected subgraph comprising at most  $r$  nodes). Thus,  $D_i \subseteq B_{2r+1}(x) \subseteq B_{t-1}(x)$ , by choice of  $r = \lfloor t/2 \rfloor - 1$ . This proves that node  $v$  indeed sees at least  $d$  distinct pieces of  $x$ 's proof in  $A_v(x)$ , and is able to successfully reconstruct  $\tilde{a}_v(x) = a_x$ . ◀

This suffices to prove completeness of  $\mathbf{Prv}_{minor}^t$ , similar to that of  $\mathbf{Prv}_{sep}^t$  in Appendix B.

# Local Certification of Graph Decompositions and Applications to Minor-Free Classes

Nicolas Bousquet  

Univ. Lyon, Université Lyon 1, LIRIS UMR CNRS 5205, F-69621, Lyon, France

Laurent Feuilloley  

Univ. Lyon, Université Lyon 1, LIRIS UMR CNRS 5205, F-69621, Lyon, France

Théo Pierron  

Univ. Lyon, Université Lyon 1, LIRIS UMR CNRS 5205, F-69621, Lyon, France

---

## Abstract

Local certification consists in assigning labels to the nodes of a network to certify that some given property is satisfied, in such a way that the labels can be checked locally. In the last few years, certification of graph classes received a considerable attention. The goal is to certify that a graph  $G$  belongs to a given graph class  $\mathcal{G}$ . Such certifications with labels of size  $O(\log n)$  (where  $n$  is the size of the network) exist for trees, planar graphs and graphs embedded on surfaces. Feuilloley et al. ask if this can be extended to any class of graphs defined by a finite set of forbidden minors.

In this work, we develop new decomposition tools for graph certification, and apply them to show that for every small enough minor  $H$ ,  $H$ -minor-free graphs can indeed be certified with labels of size  $O(\log n)$ . We also show matching lower bounds using a new proof technique.

**2012 ACM Subject Classification** Theory of computation  $\rightarrow$  Distributed algorithms

**Keywords and phrases** Local certification, proof-labeling schemes, locally checkable proofs, graph decompositions, minor-free graphs

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2021.22

**Related Version** *Full Version*: <https://arxiv.org/abs/2108.00059> [4]

**Funding** This work was supported by ANR project GrR (ANR-18-CE40-0032).

**Acknowledgements** The authors thank the reviewers for their comments, and Jens M. Schmidt for pointing out a mistake in a previous version.

## 1 Introduction

Local certification is an active field of research in the theory of distributed computing. On a high level, it consists in certifying global properties in such a way that the verification can be done locally. More precisely, for a given property, a local certification consists of a labeling (called a *certificate assignment*), and of a local verification algorithm. If the configuration of the network is correct, then there should exist a labeling of the nodes that is accepted by the verification algorithm, whereas if the configuration is incorrect no labeling should make the verification algorithm accept.

Local certification originates from self-stabilization, and was first concerned with certifying that a solution to an algorithmic problem is correct. However, it is also important to understand how to certify properties of the network itself, that is, to find locally checkable proofs that the network belongs to some graph class. There are several reasons for that. First, because certifying some solutions can be hard in general graphs, while they become simpler on more restricted classes. To make use of this fact, it is important to be able to certify that the network does belong to the restricted class. Second, because some distributed algorithms work only on some specific graph classes, and we need a way to ensure that the network does



© Nicolas Bousquet, Laurent Feuilloley, and Théo Pierron;  
licensed under Creative Commons License CC-BY 4.0

25th International Conference on Principles of Distributed Systems (OPODIS 2021).

Editors: Quentin Bramas, Vincent Gramoli, and Alessia Milani; Article No. 22; pp. 22:1–22:17

Leibniz International Proceedings in Informatics



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

belong to the class, before running the algorithm. Third, the distinction between certifying solutions and network properties is rather weak, in the sense that the techniques are basically the same. So we should take advantage of the fact that a lot is known about graph classes to learn more about certification.

In the domain of graph classes certification, there have been several results on various classes such as trees [26], bipartite graphs [23] or graphs of bounded diameter [8], but until two years ago little was known about essential classes, such as planar graphs. Recently, it has been shown that planar graphs and graphs of bounded genus can be certified with  $O(\log n)$ -bit labels [17, 18, 13]. This size,  $O(\log n)$ , is the gold standard of certification, in the sense that little can be achieved with  $o(\log n)$  bits, thus  $O(\log n)$  is often the best we can hope for.

Planar and bounded-genus graphs are classic examples of graphs classes defined by forbidden minors, a type of characterization that has become essential in graph theory since the Graph minor series of Robertson and Seymour [31]. Remember that a graph  $H$  is a minor of a graph  $G$ , is it possible to obtain  $H$  from  $G$  by deleting vertices, deleting edges, contracting edges. At this point, the natural research direction is to try to get the big picture of graph classes certification, by understanding all classes defined by forbidden minors. In particular, we want to answer the following concrete question.

► **Question 1** ([18, 14]). *Can any graph class defined by a finite set of forbidden minors be certified with  $O(\log n)$ -bit certificates?*

This open question is quite challenging: there are as many good reasons to believe that the answer is positive as negative.

First, the literature provides some reasons to believe that the conjecture is true. Properties that are known to be hard to certify, that is, that are known to require large certificates, are very different from minor-freeness. Specifically, all these properties (*e.g.* small diameter [8], non-3-colorability [23], having a non-trivial automorphism [23]) are non-hereditary. That is, removing a node or an edge may yield a graph that is not in the class. Intuitively, hereditary properties might be easier to certify in the sense that one does not need to encode information about every single edge or node, as the class is stable by removal of edges and nodes. Minor-freeness is a typical example of hereditary property. Moreover, this property, that has been intensively studied in the last decades, is known to carry a lot of structure, which is an argument in favor of the existence of a compact certification (that is a certification with  $O(\log n)$ -bit labels).

On the other hand, from a graph theory perspective, it might be surprising that a general compact certification existed for minor-free graphs. Indeed, for the known results, obtaining a compact certification is tightly linked to the existence of a precise constructive characterization of the class (*e.g.* a planar embedding for planar graphs [17, 13], or a canonical path to the root for trees [26]). Intuitively, this is because forbidden minor characterizations are about structures that are absent from the graphs, and local certification is often about certifying the existence of some structures. While such a characterization is known for some restricted minor-closed classes, we are far from having such a characterization for every minor-closed class. Note that there are a lot of combinatorial and algorithmic results on  $H$ -minor free graphs, but they actually follow from properties satisfied by  $H$ -minor free graphs, not from exact characterizations of such graphs. For certification, we need to rule out the graphs that do not belong to the class, hence a characterization is somehow necessary.



## 1.1 Our results

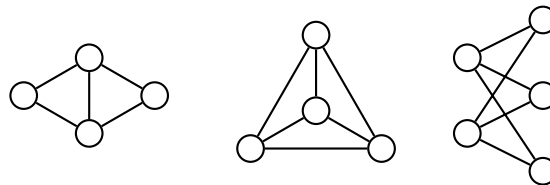
Answering Question 1 seems unfortunately out of reach, at the current state of our knowledge. We have explained above about why designing compact certification is hard for classes that do not have a constructive characterization. We will later give some intuition about why lower bounds seem equally difficult to get. In this paper, we intend to build the foundations needed to tackle Question 1. More precisely, we have four types of contributions.

First, we show how to certify some graph decompositions. Such decompositions state how to build a class based on a few elementary graphs and a few simple operations. They are essential in structural graph theory, and more specifically in the study of minor-closed classes. Amongst the most famous examples of these theorems is the proof of the 4-Color Theorem [2] or the Strong Perfect Graph Theorem [10].

Second, we show that by directly applying these tools, we can design compact certification for several  $H$ -minor free classes, for which a precise characterization is known. See Fig. 1 and 2. That is, we answer positively Question 1, for several small minors, and show that our decomposition tools can easily be used.

Class	Optimal size	Result
$K_3$ -minor free	$\Theta(\log n)$	Equivalent to acyclicity [26, 23].
Diamond-minor-free	$\Theta(\log n)$	Corollary 20
$K_4$ -minor-free	$\Theta(\log n)$	Corollary 20
$K_{2,3}$ -minor-free	$\Theta(\log n)$	Corollary 20
$(K_{2,3}, K_4)$ -minor-free ( <i>i.e.</i> outerplanar)	$\Theta(\log n)$	Corollary 20
$K_{2,4}$ -minor-free	$\Theta(\log n)$	See full version.

■ **Figure 1** Our main results for the certification of minor-closed classes.



■ **Figure 2** From left to right: the diamond, the clique on 4 vertices  $K_4$ , and the complete bipartite graph  $K_{2,3}$ .

Third, we do a systematic study of small minors to identify which is the first one that we cannot tackle. We first prove the following theorem.

► **Theorem 2.**  *$H$ -minor-free classes can be certified in  $O(\log n)$  bits when  $H$  has at most 4 vertices.*

Then, we extend this theorem to minors on five vertices with a specific shape, proving along the way new purely graph-theoretic characterizations for the associated classes. After this study, we can conclude that the next challenge is to understand  $K_5$ -minor free graphs.

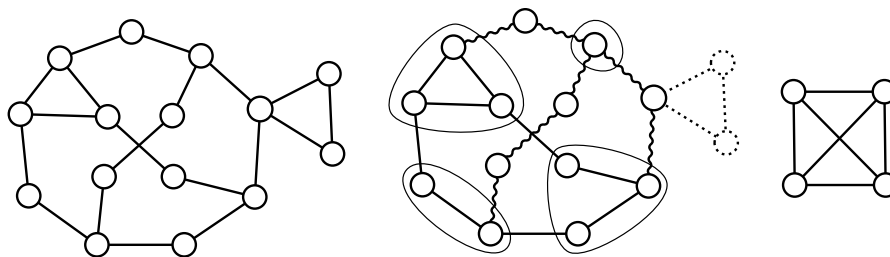
Finally, we prove a general  $\Omega(\log n)$  lower bounds for  $H$ -minor-freeness for all 2-connected graphs  $H$ . This generalizes and simplifies the lower bounds of [17] which apply only to  $K_k$  and  $K_{p,q}$ -minor-free graphs, and use ad-hoc and more complicated techniques.

At the end of the paper, we discuss why the current tools we have, both in terms of upper and lower bounds, do not allow settling Question 1. We list a few key questions that we need to answer before we can fully understand the certification of minor-closed classes, from the certification of classes with no tree minors to the certification  $k$ -connectivity, for arbitrary  $k$ .

## 1.2 Our techniques

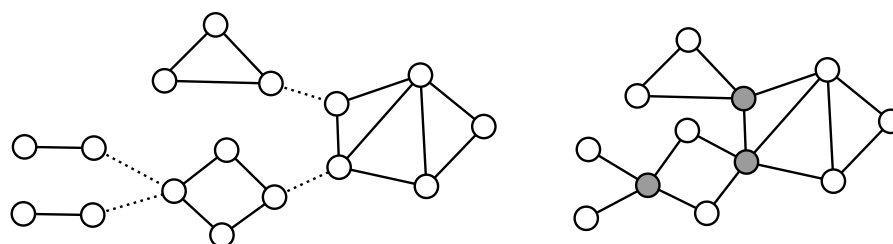
### General approach and challenges

To give some intuition about our techniques, let us focus on a concrete example:  $K_4$ -minor-free graphs. Remember that a graph has  $K_4$ -minor if we can get a  $K_4$  by deleting vertices and edges, and contracting edges. An alternative definition is that a graph has a  $K_4$ -minor, if it is possible to find four disjoint sets of vertices, called *bags*, such that: each bag is connected, there is a path between each pair of bags, these paths and bags are all vertex-disjoint (except for the endpoints of the paths that coincide with vertices of the bags). See Figure 3.



■ **Figure 3** The graph on the left has a  $K_4$  minor. Indeed, the bags of the second definition are depicted in the picture in the middle, and it is easy to find the six disjoint paths that link them. Alternatively, one can get a  $K_4$  like the one of the right-most picture by contracting all the edges inside the bags, contracting the wavy paths between bags into edges, and deleting the dotted vertices and edges.

An important observation is that, if we take a collection  $F_1, \dots, F_k$  of  $K_4$ -minor-free graphs, and organize them into a tree, by identifying pairs of vertices like in Figure 4, we get a  $K_4$ -minor-free graph.



■ **Figure 4** The five graphs with plain edges on the left picture are  $K_4$ -minor free. Organizing them into a tree by identifying the nodes linked by dotted edges makes a larger  $K_4$ -minor-free graph.

To see that, suppose that the graph we created has a  $K_4$ -minor. Then there exist bags and paths as described above. If the bags and paths are all contained in the same former  $F_i$ , then this  $F_i$  would not be  $K_4$ -minor-free, which is a contradiction. If it is not the case, then the bags and paths use vertices that belong to different subgraphs  $F_i$  and  $F_j$ . And because

of connectivity, they should use a vertex  $v$  that connects two such subgraphs (gray vertices in Figure 4). Then the bags and paths cannot be vertex-disjoint as required, because at least two of them should use the vertex  $v$ .

As a consequence of the observation above, a classic way to study  $K_4$ -minor-free graphs (as well as other classes) is to decompose the graph into maximal 2-connected components organized into a tree. This is called the *block-cut tree* of the graph, where every maximal 2-connected component is called a *block*. (Figure 4 actually show the block-cut structure of the right-most graph.) This is relevant here because 2-connected  $K_4$ -minor-free graphs have a specific structure; we will come back to this later.

Now, from the certification point of view, there is a natural strategy: first certify the structure of the block-cut tree, and then certify the special structure of each block. There are several challenges to face with this approach. First, to certify the block-cut tree, it is essential to be able to certify the connectivity of the blocks. Second, we need to avoid what we call certificate congestion, which is the issue of having too large certificates because we use too many layers of certification on some nodes. We now detail these two aspects, starting with the latter.

### Avoiding certificate congestion

In the block-cut tree of a graph, the blocks are attached to each other by shared vertices, the *cut vertices*. There is no bound on the number of blocks that are attached to a given cut vertex, and this is problematic for certification. Indeed, we cannot give to every node the list of the blocks it belongs to, as we aim for  $O(\log n)$  certificates, and such a list could contain  $\Omega(n)$  blocks. And even if we could fix the certification of the block-cut tree, the same problem would appear with the certification of the specific structure of each block: the cut vertices would have to hold a piece of certification for each block.

We basically have two tools to deal with this problem. The first one is not new, it is a degeneracy argument that already appeared in [17, 18]. A graph is  $k$ -degenerate if in every subgraph there exists a vertex that has degree at most  $k$ . Intuitively (and a bit incorrectly), this means that when we need to put a large certificate on a vertex, we can spread it on its some of its neighbors that have lower degree. A more precise statement is that, for  $k$ -degenerate graphs, we can transform a certification with  $O(f(n))$  labels *on the edges of the graphs*, into a classic certification with  $O(k \cdot f(n))$  labels on the vertices. This is relevant for our problem, as a priori there is less congestion on the edges, and minor-free classes have bounded degeneracy. Unfortunately, this is not enough for our purpose. We then build a second, more versatile tool. It consists in proving that it is possible to transform in mechanical way any certification of a graph or subgraph, into a certification that would put an empty certificate on some given vertex. Once we have this tool, we can adapt the certification of the blocks to work well in the block-cut tree: build the block-cut tree by adding blocks iteratively, making sure that the connecting node has an empty label in the certification of the newly added block.

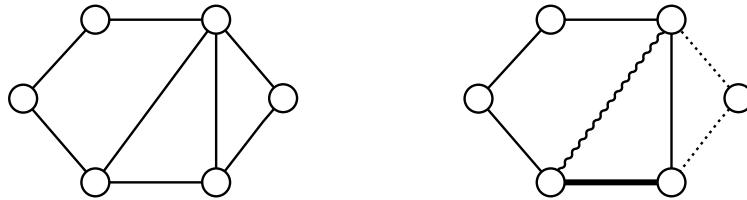
See Section 3 for the details on this topic.

### Certifying connectivity properties

Connectivity properties have been studied before in distributed certification. Specifically, certifying that for two given vertices  $s$  and  $t$ , the  $st$ -connectivity is at least  $k$  has been studied in [26] and [23]. But here we are interested in the connectivity of the graph itself, or in other words, in the  $st$ -connectivity between any pair of vertices. Clearly, proving  $st$ -connectivity for

any pair using the schemes of the literature would lead to huge certificates. Instead, we use the characterizations of  $k$ -connected graphs that are known for small values of  $k$ . There are various such characterizations, but they are all based on the same idea of *ear decomposition*.

To explain ear decompositions, consider a graph that we can build the following way (see Figure 5). Start from an edge, and iteratively apply the following process: take two different nodes of the current graph and link them by a path whose internal nodes are new nodes of the graph. It is not hard to see that such a graph is always 2-connected. Remarkably, the converse is also true: any 2-connected graph can be built (or decomposed this way). This is called an open ear decomposition, and similar constructions characterize 2-edge connected graphs and 3-vertex-connected graphs.



■ **Figure 5** Illustration of an open ear decomposition. The graph on the left can be built with the ear decomposition described on the right. First, put the bold edge. Then add the path of plain edges. Finally, add the dotted path, and the wavy path, which is just one edge.

The good thing about these constructions is that we can certify them, by describing and certifying every step. This requires some care, as when certifying a new path, we could increase the size of the certificates of the endpoints, that are already in the graph. Fortunately, the tools developed to avoid certificate congestions allow us to control the certificate size.

The details about the connectivity certification can be found in Section 4.

### Putting things together

Combining these techniques, we can prove the following theorem.

► **Theorem 3.** *For any 2-connected graph  $H$ , if the 2-connected  $H$ -minor-free graphs can be certified with  $f(n)$  bits, then the  $H$ -minor-free graphs can be certified with  $O(f(n) + \log n)$  bits.*

Going back to our example,  $K_4$ -minor-free graphs, given Theorem 3, we are left with certifying the 2-connected  $K_4$ -minor-free graphs. As said above, these have a specific shape. More precisely, 2-connected  $K_4$ -minor-free graphs have a nested ear decomposition, which is yet another type of ear decomposition, this time with additional constraints related to outerplanarity. We can certify this structure by adapting a construction from [17] for outerplanar graphs.

More generally the 2-connected graphs corresponding to most of the classes of Figure 1 have specific shapes that we can certify quite easily, which imply our compact certification schemes. We do this in Section 5. For example, the 2-connected  $C_4$ -minor-free graphs are  $K_2$  and  $K_3$ , and the 2-connected diamond-minor-free graphs are the induced cycles. A special case is  $K_{2,4}$ , that has a more complicated structure, requiring to consider 3-connected components, and some more complicated substructures. Due to space constraints, this section is deferred to the full version [4].

The full version also contains the study all the minors on at most 4 vertices, and all the minors on 5 vertices of some simple form. For these, we do not need new techniques on the

certification side, but we need to work on the graph theory side to establish new characterizations, as for these minors the literature does not help. This might be of independent interest as we study the natural notion of  $H$ -minimal graph, which are the graph that have  $H$  as a minor, but for which any vertex deletion would remove this property.

### Lower bounds

Towards the end of the paper, we show that  $\Omega(\log n)$ -bit labels are necessary to certify (2-connected) minor-free graph classes. When it comes to  $\Omega(\log n)$  lower bounds in our model, there are basically two complementary techniques (called *cut-and-plug techniques* in [14]). Both techniques basically show that paths cannot be differentiated from cycles, if the certificates use  $o(\log n)$  bits. First, in [23], the idea is to use many correct path instances, and to prove that we can plug them into an incorrect cycle instance, thanks to a combinatorial result from extremal graph theory. Second, in [19], the idea is to consider a path, to cut it into small pieces, and to show via Sterling formula, that there exists a shuffle of these pieces that can be closed into a cycle.

Previous lower bounds for minor-free graphs in [17] followed the same kind of strategies as [23] and [19], with the same type of counting arguments, more complicated constructions, and tackled only minors that were cliques or bicliques.

In this paper, we are able to do a black-box reduction between the path/cycle problem and the  $H$ -minor-freeness for any 2-connected  $H$ . This way we avoid explicit counting arguments, and get a more general result with a simpler proof.

### 1.3 Related work

Local certification first appeared under the name of *proof-labeling schemes* in [26], inspired by works on self-stabilizing algorithms (see [11] for a book on self-stabilization). It has then been generalized under the name of *locally checkable proofs* in [23], and the field has been very active since these seminal papers. In the following, we will focus on the papers about local certification of graph classes, but we refer to [14] and [16] for an introduction and a survey of local certification in general.

As said earlier, certification was first mostly about checking that the solution to an algorithmic problem was correct, a typical example being the verification of a spanning tree [26]. Some graph properties have also been studied, for example symmetry in [23], or bounded diameter in [8]. Very recently, classes that are more central in graph theory have attracted attention. It was first proved in [30], as an application of a more general method, that planar graphs can be certified with  $O(\log n)$  bits in the more general model of distributed interactive proofs. Then it was proved in [17] that these graphs can actually be certified with  $O(\log n)$  bits in the classic model, that is, without interaction. This result was extended to bounded-genus graphs in [18]. Later, [13] provided a simpler proof of both results via different techniques. It was also proved in [29] that cographs and distance-hereditary graphs have compact distributed interactive proofs.

Still in distributed computing, but outside local certification, the networks with some forbidden structures have attracted a lot of attention recently. A popular topic is the distributed detection of some subgraph  $H$ , which consists, in the CONGEST (or CONGEST-CLIQUE) model to decide whether the graph contains  $H$  as a subgraph or not (see [7] and the references therein). A related task is  $H$ -freeness testing, which is the similar but easier task consisting in deciding whether the graph is  $H$ -free or far from being  $H$ -free (in terms

of the number of edges to modify to get a  $H$ -free graph). This line of work was formalized by [6] after the seminal work of [5] (see [20] and the references therein). To our knowledge, no detection/testing algorithm or lower bounds have been designed for  $H$ -minor-freeness.

Finally, we have mentioned in the introduction that certifying that the graph belongs to some given class is important because some algorithms are specially designed to work on some specific classes. For example, there is a large and growing literature on approximation algorithms for *e.g.* planar, bounded-genus, minor-free graphs. We refer to [15] for a bibliography of this area. There are also interesting works for exact problems in the CONGEST model, *e.g.* in planar graphs [21], graphs of bounded treewidth or genus [24] and minor-free graphs [25]. In particular the authors of [25] justify the focus on minor-free graphs by the fact that this class allows for significantly better results than general graphs, while being large enough to capture many interesting networks. Very recently, [22] proved general tight results on low-congestion short-cuts (an essential tool for algorithms in the CONGEST model) for graphs excluding a dense minor.

## 2 Preliminaries

In this section, we define formally the notions we use and describe some useful known certification building blocks.

### 2.1 Graphs and minors

Let  $G = (V, E)$  be a graph. Let  $X \subseteq V$ . The *subgraph of  $G$  induced by  $X$*  is the graph with vertex set  $X$  and edge set  $E \cap X^2$ . The graph  $G \setminus X$  is the subgraph of  $G$  induced by  $V \setminus X$ . A graph  $G'$  is a *subgraph* of  $G$  if  $V' \subseteq V$  and  $E' \subseteq E$ . For every  $v \in V$ ,  $N(v)$  denotes the *neighborhood* of  $v$  that is the set of vertices adjacent to  $v$ . The graph  $G$  is  *$d$ -degenerate* if there exists an ordering  $v_1, \dots, v_n$  of the vertices such that, for every  $i$ ,  $N(v_i) \cap \{v_{i+1}, \dots, v_n\}$  has size at most  $d$ . It refines the notion of maximum degree since any graph of maximum degree  $\Delta$  are indeed  $\Delta$ -degenerate (but the gap between  $\Delta$  and the degeneracy can be arbitrarily large). Let  $u, v \in V$ , a *path* from  $u$  to  $v$  is a sequence of vertices  $v_0 = u, v_1, \dots, v_\ell = v$  such that for every  $i \leq \ell - 1$ ,  $v_i v_{i+1}$  is an edge. It is a *cycle* if  $v_\ell v_0$  also exists.

A graph  $G$  is *connected* if there exists a path from  $u$  to  $v$  for every pair  $u, v \in V$ . All along the paper, we only consider connected graphs. Indeed, in certification, the nodes can only communicate with their neighbors, so no node can communicate with nodes of another connected component.

A vertex  $v$  is a *cut-vertex* if  $G \setminus \{v\}$  is not connected. If  $G$  does not contain any cut-vertex,  $G$  is *2-(vertex)-connected*. If the removal of any edge does not disconnect the graph, we say that  $G$  is *2-edge-connected*. A graph is  *$k$ -(vertex)-connected* if there does not exist any set  $X$  of size  $k - 1$  such that  $G \setminus X$  is not connected. To avoid cumbersome notations, we will simply write  *$k$ -connected* for  *$k$ -vertex-connected*.

A graph  $H$  is a *minor* of  $G$  if  $H$  can be obtained from  $G$  by deleting vertices, deleting edges and contracting edges. Equivalently, it means that, if  $G$  is connected, there exists a partition of  $V$  into connected sets  $V_1, \dots, V_{|H|}$  such that there is (at least) an edge between  $V_i$  and  $V_j$  if  $h_i h_j$  is an edge of  $H$ . We say that  $V_1, \dots, V_{|H|}$  is a *model* of  $H$ . The graph  $G$  is  *$H$ -minor-free* if it does not contain  $H$  as a minor.

### 2.2 Local computation and certification

We assume that the graph is equipped with unique identifiers in polynomial range  $[1, n^k]$ , thus these identifiers can be encoded on  $O(\log n)$  bits.

Local certification is a mechanism for verifying properties of labeled or unlabeled graphs. In this paper we will use a local certification at distance 1, which is basically the model called *proof-labeling schemes* [26]. A convenient way to describe a local certification is with a prover and a verifier. The *prover* is an external entity that assigns to every node  $v$  a certificate  $c(v)$ . The *verifier* is a distributed algorithm, in which every node  $v$  acts as follows:  $v$  collects the identifiers and the certificates of its neighbor and itself, and outputs a decision *accept* or *reject*. A local certification certifies a graph class  $\mathcal{C}$  if the following two conditions are verified:

1. For every graph of  $\mathcal{C}$ , the prover can find a certificate assignment such that the verifier accepts, that is, all nodes output *accept*.
2. For every graph not in  $\mathcal{C}$ , there is no certificate assignment that makes the verifier accept, that is, for every assignment, there is at least one node that rejects.

The size of the certificate of  $\mathcal{C}$  is the largest size of a certificate assigned to a node of a graph of  $\mathcal{C}$ .

Note that to describe a local certification, the only essential part is the verifier algorithm, the prover is just a way to facilitate the description of a scheme.

In this paper, we are going to use a variant of the model above, called *edge certification*, where the certificates can be assigned on both the nodes and the edges. See Subsection 3.1.

### 2.3 Known building blocks for graph certification

There are few known certification schemes that we are going to use intensively as building blocks in the paper.

► **Lemma 4** ([26, 1]). *Acyclicity can be certified in  $O(\log n)$  bits.*

The classic way to certify that the graph is acyclic, is for the prover to choose a root node, and then to give to every node as its certificate its distance to the root. The nodes can simply check that the distances are consistent.

The same idea can be used to certify a *spanning tree* of the graph, encoded locally at each node by the pointer to its parent, which is simply the ID of this parent. The scheme is the same, except that the prover, in addition to the distances, gives the ID of the root, and the verification algorithm checks that all nodes have been given the same root-ID, and only takes into account the edges that correspond to pointers (also the root checks that its ID is the root-ID). A spanning tree is a very useful tool to broadcast the *existence of a vertex satisfying a locally checkable property*: simply choose a spanning tree rooted at the special vertex, encode it locally with pointers and certify it. Then the root can check that indeed it has the right property, and all the other vertices know that such a vertex exists.

Finally, with the same ideas, one can easily deduce  $O(\log n)$  certification for paths. We just add to the acyclicity scheme the verification that the degree of every node is at most 2. Note that cycles do not need certificates to be verified: every node just checks that it has degree exactly 2.

Let us now define a graph class that will appear in several decompositions.

► **Definition 5.** *A path-outerplanar graph is a graph that admits a path  $P$  that can be drawn on a horizontal line, such that all the edges that do not belong to  $P$  can be drawn above that line without crossings. The edges are said to be nested.*

We are going to use the two following classic results as black boxes.

► **Lemma 6** ([17]). *Path-outerplanar graphs can be certified with  $O(\log n)$ -bit certificates.*

► **Lemma 7** ([26]). *Every graph class can be certified with  $O(n^2)$  bits.*

The idea of the scheme is that the prover gives to every node  $v$  the map of the graph, *e.g.* as an adjacency matrix, along with the position of  $v$  in this map. Then every node can check that it has been given the same map as its neighbors, and that the map is consistent with its neighborhood in the network.

### Organization of the paper

The full version of the paper contains more material than this short version. First, we had to remove the majority of the proofs from the short version. As a consequence, some parts of the paper are mainly lists of lemmas, but the intuition provided in the introduction should be enough to follow the articulation of the reasoning. Also, as said before, several full sections of the paper appear only in the full version [4].

## 3 Avoiding certificate congestion

One can obtain many structured graph classes like minor free graphs with “gluing” operations, for instance, by identifying vertices of two graphs of the class. If we have a certification for both graphs, we would like to simply take both certificate assignments to certify the new graph. However, for the vertex on which the two graphs are glued, the size of the certificate might have doubled. While it is not a problem for bounded degree graphs, it can become problematic if many gluing operations occur around the same vertex, since this vertex would get an additional certificate from each operation. In this section, we present two ways to tackle these issues, that will be used in the forthcoming sections.

The first one consists in shifting the certification on edges instead of vertices, which helps in the sense that when gluing on vertices the edge certificate can remain unchanged. As we will see, the edge setting is equivalent to the usual vertex certification for nice enough classes. The second option uses that one can (almost) freely assume that a given vertex has an empty label in a correct certification.

### 3.1 Edge certification and degeneracy

Transforming a node certification into an edge certification can always be done without additional asymptotic costs: just copy on every edge the certificate of the two endpoints, and adapt the verification algorithm accordingly. Transforming an edge certification into a node certification is also always possible, by giving a copy of the edge label to each of its endpoint. But this transformation can drastically increase the certificate size: if an edge certification uses  $\Omega(f(n))$ -bit labels, the associated node certification might use  $\Omega(n \cdot f(n))$ -bit if the maximum degree of the graph is linear. The following theorem ensures that in degenerate graph classes there is a more efficient transformation that permits to drastically reduce the size of the certificate.

► **Theorem 8** ([18]). *Consider an edge certification of a graph class  $\mathcal{C}$  where the edges are labeled with  $f(n)$ -bit certificates. If  $\mathcal{C}$  is  $d$ -degenerate, then there exists a (node) certification with  $d \cdot f(n)$ -bit certificates.*

Note that  $H$ -minor free graphs have degeneracy  $O(h\sqrt{\log h})$  where  $h = |V(H)|$  [27, 33]. Therefore, we can freely put labels on edges when certifying classes defined by forbidden minors.



### 3.2 Certification with one empty label

In this part, our goal is to erase the certificate of a node. To this end, we first consider certification of spanning trees and strengthen both Lemma 4 and the discussion that followed in Subsection 2.3. We then extend this intermediate step to every graph class in Lemma 10.

► **Lemma 9.** *Let  $T$  be a spanning tree of  $G$ . There exists a certification of  $T$  that does not assign a label to the root, and uses the same certificate as the classic tree certification (cf. Subsection 2.3) on the other nodes.*

A *pointed graph* is a graph with one selected node. Given a class, one can build its pointed version by taking for each graph all the pointed versions of it.

► **Lemma 10.** *Consider a class  $\mathcal{C}$  that can be certified with certificates of size  $f(n)$ . One can certify the pointed class of  $\mathcal{C}$  with  $O(f(n) + \log n)$  bits, without having to put certificates on the selected node.*

The previous results can be easily iterated: one can always remove the labels of  $k$  nodes (as long as they are pairwise non-adjacent) to the cost of a factor  $k$  in the size of the certificates. Therefore, the result extends to the case of  $k$ -independent pointed classes (i.e. where an independent set of size at most  $k$  is selected instead of only one vertex).

► **Corollary 11.** *Consider a class that can be certified with certificates of size  $f(n)$ . One can certify the  $k$ -independent pointed class with  $O(kf(n) + k \log n)$  bits, without having to put certificates on the selected nodes.*

Moreover, with more constraints on the structure of the set of pointed vertices (for instance if they are all at distance at least 3), one could even obtain certificate of size  $O(f(n) + k \log n)$  (since every node receives the certificate of at most one selected node).

## 4 Connectivity and connectivity decompositions

In this section, we study the certification of connectivity properties and connectivity decompositions, in particular the block-cut tree mentioned in the introduction.

An *ear decomposition* is a way to build a graph by iteratively adding paths, the so-called *ears*. Ear decompositions are central tools for decades in structural graph theory and are used in many decomposition or algorithmic results. There exists various variants of this process, that characterize different classes and properties. For certification, these decompositions happen to be easier to manipulate than some other types of characterizations since they are based on iterative construction of the graph, and use paths, which are easy to certify. These paths are convenient since we can “propagate” some quantity of information on them as long as every vertex belongs to a bounded number of paths. In this section, we remind several such decompositions, and use them to certify various connectivity properties and decompositions.

Let us start with 2-connectivity. A graph  $G$  has an *open ear decomposition* if  $G$  can be built, by starting from a single edge, and iteratively applying the following process: take two different nodes of the current graph and link them by a path whose internal nodes are new nodes of the graph (such a path is called *an ear*). Note that this path can be a single edge, and then there is no new node. Let an *inner node* of an ear be a vertex that is created with this ear, and let a *long ear* be an ear with at least one inner node.

► **Theorem 12** ([35] reformulated). *A graph is 2-connected if and only if it has an open ear decomposition.*

As described in the introduction, we use this characterization to certify vertex and edge 2-connectivity.

► **Lemma 13.** *2-connected graphs can be certified with  $O(\log n)$  bits.*

► **Corollary 14.** *2-edge-connected graphs can be certified with  $O(\log n)$  bits.*

A more refined type of ear decomposition characterizes the 3-vertex-connected graphs, based on *Mondschein sequences*.

► **Definition 15** ([32, 28, 9]). *Let  $ru$  and  $rt$  be two edges of a graph  $G$ . A Mondschein sequence through  $rt$ , avoiding  $u$  is an open ear decomposition of  $G$  such that:*

1.  *$rt$  is in the first ear.*
2. *the ear that creates node  $u$  is the last long ear,  $u$  is its only inner vertex, and it does not contain  $ru$ .*
3. *the ear decomposition is non-separating, that is, for every long ear except the last one, every inner node has a neighbor that is created in a later ear.*

► **Theorem 16** ([9, 32]). *Let  $ru$  and  $rt$  be two edges of a graph  $G$ . The graph  $G$  is 3-vertex-connected if and only if it has a Mondschein sequence through  $rt$  avoiding  $u$ , and there are three internally vertex-disjoint path between  $t$  and  $u$ .*

We can translate this theorem into a compact certification.

► **Corollary 17.** *3-connectivity can be certified with  $O(\log n)$  bits on vertices and  $O(1)$  bits on edges.*

With the tools we previously introduced, we can now certify a well-known decomposition into parts of higher connectivity, called the *block-cut tree*. This allows to prove the following result. Due to space constraint, the needed definition and proof only appear in the full version [4].

► **Theorem 3.** *For any 2-connected graph  $H$ , if the 2-connected  $H$ -minor-free graphs can be certified with  $f(n)$  bits, then the  $H$ -minor-free graphs can be certified with  $O(f(n) + \log n)$  bits.*

## 5 Application to $C_4$ , $C_5$ , Diamond, $K_4$ and $K_{2,3}$ minor-free graphs

This section is devoted to the certification of  $C_4$ -minor-free, diamond-minor-free graphs,  $K_4$ -minor-free graphs and  $K_{2,3}$ -minor-free graphs. All the proofs can be found in the full version [4]. They will all follow the same structure: prove that the 2-connected components, which are more structured, can be certified with small labels, and then use Theorem 3 to conclude for the general case.

A *nested ear decomposition* is an open ear decomposition that starts from a path, with two properties: (1) both ends of an ear have to be connected to the same ear, and (2) for every ear, the ears that are plugged onto it are nested. Eppstein proved in [12] that, for 2-connected graphs, being  $K_4$ -minor-free is equivalent to having a nested ear decomposition. Therefore, we get the following.

► **Theorem 18.** *2-connected  $K_4$ -minor-free graphs can be certified with  $O(\log n)$ -bit labels.*

We now extend the techniques to other small graphs, but before we prove a simple statement for the case of  $C_5$ .

► **Lemma 19.** *2-connected  $C_5$ -minor free graphs are either graphs of size at most 4 or  $K_{2,p}$  or  $K'_{2,p}$  which is the complete bipartite graph  $K_{2,p}$  plus an edge between the two vertices on the set of size 2.*

► **Corollary 20.** *The following classes of graphs can be certified with  $O(\log n)$  bit certificates:  $C_4$ -minor-free graphs,  $C_5$ -minor free graphs, diamond-minor-free graphs, house-minor free graphs<sup>1</sup>, outerplanar graphs (that is  $(K_{2,3}, K_4)$ -minor-free graphs),  $K_{2,3}$ -minor-free and  $K_4$ -minor-free graphs.*

## 6 Lower bounds

In this section, we show logarithmic lower bounds for  $H$ -minor-freeness for every 2-connected graph  $H$ . These results generalize the lower bounds of [17] for  $K_k$  and  $K_{p,q}$ . Our technique is a simple reduction from the certification of paths, via a local simulation. In contrast, the proofs of [17] were ad-hoc adaptations of the constructions of [23] and [19], with explicit counting arguments. Moreover, our lower bounds apply in the stronger model of locally checkable proofs, where the verifier can look at a constant distance.

► **Theorem 21.** *For every 2-connected graph  $H$ , certifying  $H$ -minor-freeness requires  $\Omega(\log n)$  bits.*

Let us start by proving a couple of lemmas. Let  $H$  be a 2-connected graph, and let  $e = uv$  be an arbitrary edge of  $H$ . Let  $H^-$  be the graph  $H \setminus e$ . Note that  $H^-$  is connected. We are going to consider copies of  $H^-$ , that we index as  $H_i^-$ 's, and where the copies of the nodes  $u$  and  $v$  will be called  $u_i$  and  $v_i$ . Let  $\mathcal{P}$  be the class of all the graphs that can be made by taking some  $k$  copies of  $H^-$ , and by identifying for every  $i \in [1, k-1]$ ,  $v_i$  with  $u_{i+1}$ . In other words,  $\mathcal{P}$  is the set of paths, where every edge is a copy of  $H^-$ . The class  $\mathcal{C}$  is the same as  $\mathcal{P}$  except that we close the paths into cycles, that is, we identify  $v_k$  with  $u_1$ .

► **Lemma 22.** *The graphs of  $\mathcal{P}$  are all  $H$ -minor-free, and the graphs of  $\mathcal{C}$  all contain  $H$  as a minor.*

**Proof.** Let  $G$  be a graph of  $\mathcal{P}$ . Note that every vertex  $v_i$  (identified with  $u_{i+1}$ ) for  $i \in \{1, \dots, k-1\}$ , is a cut vertex of  $G$ . Therefore, since  $H$  is 2-connected, a model of  $H$  can only appear between two such nodes. By construction this cannot happen, as the graphs between the cut vertices are all  $H^-$ . Thus  $G$  is  $H$ -minor-free.

Now let  $G$  be a graph of  $\mathcal{C}$ . We claim that  $G$  contains  $H$  as a minor. Consider the following model of  $H$ . Any  $H_i^-$  is a model of  $H$  except for the edge  $uv$ . Since we have made a cycle of  $H_i^-$ 's, there is a path between  $v_i$  and  $u_i$  outside  $H_i^-$ , and this path finishes the model of  $H$ . ◀

► **Lemma 23.** *Let  $H$  be a 2-connected graph. If there is a certification with  $O(f(n))$  bits for  $H$ -minor-free graphs, then there is a  $O(f(n))$  certification for paths.*

Now Theorem 21 follows from the fact that paths cannot be certified with  $o(\log n)$  bits [26, 23]. Note that this also applies in the locally checkable proof setting, as soon as the number of copies of  $H^-$  is large enough, since the lower bound for paths also applies to locally checkable proofs.

<sup>1</sup> The house being a  $C_4$  plus a vertex connected to two consecutive vertices of the  $C_4$ .

## 7 Discussion

### Milestones to go further

In this paper, we develop several tools and use them to show that some minor closed graph classes can be certified with  $O(\log n)$  bits. While these tools probably permit certifying new classes, we simply wanted to illustrate their interest. Let us now discuss the tools that are missing in order to tackle the general question on  $H$ -minor-freeness and which steps can be interesting to tackle it.

First, as we explained in the section of the full version concerned with 5 vertices, certification of  $H$ -minor free classes seems easier when  $H$  is sparse. One first question that might be interested to look at is the following:

► **Question 24.** *Let  $T$  be a tree. Can  $T$ -minor free graphs be certified with  $O(\log n)$  bits?*

The answer to this question for small graphs  $H$  (up to 5 vertices) is not very interesting, since the number of vertices of degree at least 3 is bounded (and then the whole structure of the graph is “simple”). Even if it remains simple for any  $H$ , there is no trivial argument allowing us to certify these nodes with  $O(\log n)$  bits.

A natural approach to tackle Conjecture 1 would consist in an induction on the size of  $H$ . Indeed, knowing how to certify  $H \setminus x$  for any possible  $x$  may help to certify  $H$ . The basic idea would consist in separating two cases. 1) When  $H$  is not heavily connected where we can heavily use the fact that we can  $H \setminus x$  can be certified. And 2) when  $H$  is heavily connected, try to use a more general argument. A first step toward step 1) would consist in proving that if  $H$ -minor-freeness can be certified, then so is  $H + K_1$ -minor-freeness<sup>2</sup>. We proved it for five vertices in the full version, but the proof heavily uses the structure of the graphs on four vertices. One can then naturally ask the following general question:

► **Question 25.** *Let  $H$  be a graph. Can  $(H + K_1)$ -minor free graphs be certified with  $O(\log n)$  bits when  $H$  can be certified with  $O(\log n)$  bits?*

As in the proof of the analogue theorem for 5 vertices in the full version, we know that we can assume that  $G$  is  $H$ -minimal. Even if most of the techniques we use are specific, some (basic) general properties of  $H$ -minimal graphs which might be useful to tackle this question.

In structural graph theory, a particular class of  $H$ -minimal graphs received a considerable attention which are minimally non-planar graphs, in other words, graphs  $G$  that are minimal and that contains either a  $K_5$  or a  $K_{3,3}$  as a minor. It might be interesting to determine if minimally non-planar graphs can be certified with  $O(\log n)$  bits.

Note that if we can answer positively Question 25 positively, the second step would consist in proving the conjecture when we add to  $H$  a vertex attached to a single vertex of  $H$ . Proving this case would, in particular, imply a positive answer to Question 24.

If we want to consider dense graphs, the questions seem to become even harder. In particular, one of the first main complicated  $H$ -minor class to deal with is probably the class of  $K_5$ -free graphs. There are several reasons for that. First, it is the smallest 4-connected graph and the hardness to certify seem to be highly related to the connectivity of the graph that is forbidden as a minor. The second reason is that it is the smallest graph for which  $H$ -minor free graphs is a super class of planar graphs. In other words, we cannot take advantage of the “planarity” of the graph (formally or informally) to certify the graph class. We then ask the following question:

---

<sup>2</sup>  $H + K_1$  is the graph  $H$  plus an isolated vertex.

► **Question 26.** *Can  $K_5$ -minor free graphs be certified with  $O(\log n)$  bits?*

Wagner proved in [34] that a graph is  $K_5$ -minor-free if and only if it can be built from planar graphs and from a special graph  $V_8$  by repeated clique sums. A *clique sum* consists in taking two graphs of the class and gluing them on a clique, and then (potentially) remove edges of that clique. While it should have been easy to certify this sum if we keep the edges of the clique, the fact that they might disappear makes the work much more complicated for certification.

More generally, many decompositions are using the fact that we replace a subgraph by a smaller structure (a single vertex or an edge for instance) only connected to the initial neighbors of that structure in the graph. Certifying such structures is a challenging question whose positive answer can probably permit to break several of the current hardest cases.

### Obstacles towards lower bounds

There are also several obstacles preventing us to prove super-logarithmic lower bounds for the certificate size of  $H$ -minor-free graphs. Basically, the only techniques we know consist in (explicit or implicit) reductions to communication complexity. In particular, communication complexity.

Let us remind what these reductions look like. In such a reduction, one considers a family of graphs with two vertex sets  $A$  and  $B$ , with few edges in between. These graphs are defined in such a way that the input of Alice for the disjointness problem can be encoded in the edges of  $A$  and the input of Bob in the edges of  $B$ . Then, given a certification scheme, Alice and Bob can basically simulate the verification algorithm, and deduce an answer for the disjointness problem. If a certification with small labels existed for the property at hand, then the communication protocol would contradict known lower bounds, which proves a lower bound for certification.

The difficulty of using this proof for  $H$ -minor free graphs comes from the fact that it is difficult to control where a minor can appear, that is, to control the models of  $H$ . For example, it is difficult to control that if  $H$  appears in the graph, then the nodes  $V_i$  associated with some node  $i$  of  $H$  are on Alice's side. As a comparison, for proving properties on the diameter, [8] used a construction where all the longest paths in the graph had to start from Alice side and finish in Bob side, but such a property seems difficult to obtain for minors.

### Connectivity questions

A large part of the paper is devoted to certify connectivity and related notions that are of independent importance, for instance to certify the robustness of a network. For these, we do not have lower bounds, and leave the following question open.

► **Question 27.** *Does the certification of  $k$ -connectivity require  $\Omega(\log n)$  bits?*

For this question, it is tempting to try a construction close to the one we have used for  $H$ -minor-free graphs. For example, one could think that the nodes of the path/cycle could simulate the  $k$ -th power of the graph, which is  $k$ -connected if and only if the graph is a cycle. But this does not work: we want the *yes*-instances for the property (*e.g.* the  $k$ -connected graphs) to be mapped to *yes*-instances for acyclicity (*e.g.* paths), and not with the *no*-instances, which are the cycles.

An interesting open problem about  $k$ -connectivity also is on the positive side:

► **Question 28.** *Can  $k$ -connectivity be certified with  $O(\log n)$  bits for any  $k \geq 4$ ?*

Beyond the question of certifying the connectivity itself, we would like to be able to decompose graphs based on  $k$ -connected components, like what we did with the block-cut tree for 2-connectivity. Such decomposition are more complicated and less studied than block-cut trees, but for 3-connectivity such a tool is SPQR trees [3]. Unfortunately, similarly to the clique sum operation we mentioned earlier, some steps of the SPQR tree construction are based on edges that can be removed in later steps, making it hard to certify this structure.

---

## References

- 1 Yehuda Afek, Shay Kutten, and Moti Yung. Memory-efficient self stabilizing protocols for general networks. In *WDAG '90*, volume 486, pages 15–28, 1990. doi:10.1007/3-540-54099-7\_2.
- 2 Kenneth Appel, Wolfgang Haken, et al. Every planar map is four colorable. *Bulletin of the American mathematical Society*, 82(5):711–712, 1976.
- 3 Giuseppe Di Battista and Roberto Tamassia. Incremental planarity testing (extended abstract). In *FOCS 89*, pages 436–441, 1989. doi:10.1109/SFCS.1989.63515.
- 4 Nicolas Bousquet, Laurent Feuilloley, and Théo Pierron. Local certification of graph decompositions and applications to minor-free classes. *CoRR*, abs/2108.00059, 2021. arXiv:2108.00059.
- 5 Zvika Brakerski and Boaz Patt-Shamir. Distributed discovery of large near-cliques. *Distributed Comput.*, 24(2):79–89, 2011. doi:10.1007/s00446-011-0132-x.
- 6 Keren Censor-Hillel, Eldar Fischer, Gregory Schwartzman, and Yadu Vasudev. Fast distributed algorithms for testing graph properties. *Distributed Comput.*, 32(1):41–57, 2019. doi:10.1007/s00446-018-0324-8.
- 7 Keren Censor-Hillel, Orr Fischer, Tzlil Gonen, François Le Gall, Dean Leitersdorf, and Rotem Oshman. Fast distributed algorithms for girth, cycles and small subgraphs. In *DISC 2020*, volume 179 of *LIPICs*, pages 33:1–33:17, 2020. doi:10.4230/LIPICs.DISC.2020.33.
- 8 Keren Censor-Hillel, Ami Paz, and Mor Perry. Approximate proof-labeling schemes. *Theor. Comput. Sci.*, 811:112–124, 2020. doi:10.1016/j.tcs.2018.08.020.
- 9 Joseph Cheriyan and S. N. Maheshwari. Finding nonseparating induced cycles and independent spanning trees in 3-connected graphs. *J. Algorithms*, 9(4):507–537, 1988. doi:10.1016/0196-6774(88)90015-6.
- 10 Maria Chudnovsky, Neil Robertson, Paul Seymour, and Robin Thomas. The strong perfect graph theorem. *Annals of mathematics*, pages 51–229, 2006.
- 11 Shlomi Dolev. *Self-Stabilization*. MIT Press, 2000. URL: <http://www.cs.bgu.ac.il/~Edolev/book/book.html>.
- 12 David Eppstein. Parallel recognition of series-parallel graphs. *Inf. Comput.*, 98(1):41–55, 1992. doi:10.1016/0890-5401(92)90041-D.
- 13 Louis Esperet and Benjamin Lévêque. Local certification of graphs on surfaces. *CoRR*, abs/2102.04133, 2021.
- 14 Laurent Feuilloley. Introduction to local certification. *CoRR*, abs/1910.12747, 2019.
- 15 Laurent Feuilloley. Bibliography of distributed approximation on structurally sparse graph classes. *CoRR*, abs/2001.08510, 2020.
- 16 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>, arXiv:1606.04434.
- 17 Laurent Feuilloley, Pierre Fraigniaud, Pedro Montealegre, Ivan Rapaport, Éric Rémila, and Ioan Todinca. Compact distributed certification of planar graphs. In *PODC '20*, pages 319–328. ACM, 2020. doi:10.1145/3382734.3404505.
- 18 Laurent Feuilloley, Pierre Fraigniaud, Pedro Montealegre, Ivan Rapaport, Eric Rémila, and Ioan Todinca. Local certification of graphs with bounded genus. *CoRR*, abs/2007.08084, 2020.
- 19 Laurent Feuilloley and Juho Hirvonen. Local verification of global proofs. In *DISC 2018*, volume 121 of *LIPICs*, pages 25:1–25:17, 2018. doi:10.4230/LIPICs.DISC.2018.25.

- 20 Pierre Fraigniaud and Dennis Olivetti. Distributed detection of cycles. *ACM Trans. Parallel Comput.*, 6(3):12:1–12:20, 2019. doi:10.1145/3322811.
- 21 Mohsen Ghaffari and Bernhard Haeupler. Distributed algorithms for planar networks II: low-congestion shortcuts, mst, and min-cut. In *SODA 2016*, pages 202–219. SIAM, 2016. doi:10.1137/1.9781611974331.ch16.
- 22 Mohsen Ghaffari and Bernhard Haeupler. Low-congestion shortcuts for graphs excluding dense minors. In *PODC 2021*, page To appear., 2021.
- 23 Mika Göös and Jukka Suomela. Locally checkable proofs in distributed computing. *Theory of Computing*, 12(19):1–33, 2016. doi:10.4086/toc.2016.v012a019.
- 24 Bernhard Haeupler, Taisuke Izumi, and Goran Zuzic. Near-optimal low-congestion shortcuts on bounded parameter graphs. In *DISC 2016*, volume 9888, pages 158–172. Springer, 2016. doi:10.1007/978-3-662-53426-7\_12.
- 25 Bernhard Haeupler, Jason Li, and Goran Zuzic. Minor excluded network families admit fast distributed algorithms. In *PODC 2018*, pages 465–474, 2018.
- 26 Amos Korman, Shay Kutten, and David Peleg. Proof labeling schemes. *Distributed Computing*, 22(4):215–233, 2010. doi:10.1007/s00446-010-0095-3.
- 27 Alexandr V Kostochka. The minimum hadwiger number for graphs with a given mean degree of vertices. *Metody Diskret. Analiz.*, 38:37–58, 1982.
- 28 Lee F. Mondschein. *Combinatorial Ordering and the Geometric Embedding of Graphs*. PhD thesis, M.I.T. Lincoln Laboratory / Harvard University, 1971.
- 29 Pedro Montealegre, Diego Ramírez-Romero, and Iván Rapaport. Compact distributed interactive proofs for the recognition of cographs and distance-hereditary graphs. *CoRR*, abs/2012.03185, 2020.
- 30 Moni Naor, Merav Parter, and Eylon Yogev. The power of distributed verifiers in interactive proofs. In *SODA 2020*, pages 1096–115. SIAM, 2020. doi:10.1137/1.9781611975994.67.
- 31 Neil Robertson and Paul D Seymour. Graph minors—a survey. *Surveys in combinatorics*, 103:153–171, 1985.
- 32 Jens M. Schmidt. Mondschein sequences (a.k.a.  $(2, 1)$ -orders). *SIAM J. Comput.*, 45(6):1985–2003, 2016. doi:10.1137/15M1030030.
- 33 Andrew Thomason. An extremal function for contractions of graphs. In *Mathematical Proceedings of the Cambridge Philosophical Society*, volume 95(2), pages 261–265. Cambridge University Press, 1984.
- 34 K. Wagner. Über eine eigenschaft der ebenen komplex. In *Math. Ann.*, volume 114, pages 570–590, 1937.
- 35 Hassler Whitney. Non-separable and planar graphs. *Transactions of the American Mathematical Society*, 34:339–362, 1932. doi:10.1090/S0002-9947-1932-1501641-2.





# RandSolomon: Optimally Resilient Random Number Generator with Deterministic Termination

Luciano Freitas de Souza ✉

CEA LIST, Université de Paris-Saclay,  
Gif-sur-Yvette, France  
LTCI, Télécom Paris,  
Institut Polytechnique de Paris, France

Sara Tucci-Piergiovanni ✉

CEA LIST, Université de Paris-Saclay,  
Gif-sur-Yvette, France

Oana Stan ✉

CEA LIST, Université de Paris-Saclay,  
Gif-sur-Yvette, France

Petr Kuznetsov ✉

LTCI, Télécom Paris, Institut Polytechnique de  
Paris, France

Andrei Tonkikh ✉

LTCI, Télécom Paris,  
Institut Polytechnique de Paris, France

Renaud Sirdey ✉

CEA LIST, Université de Paris-Saclay,  
Gif-sur-Yvette, France

Nicolas Quero ✉

CEA LIST, Université de Paris-Saclay,  
Gif-sur-Yvette, France

---

## Abstract

Multi-party random number generation is a key building-block in many practical protocols. While straightforward to solve when all parties are trusted to behave correctly, the problem becomes much more difficult in the presence of faults. This paper presents RandSolomon, a partially synchronous protocol that allows a system of  $N$  processes to produce an unpredictable common random number shared by correct participants. The protocol is optimally resilient, as it allows up to  $f = \lfloor \frac{N-1}{3} \rfloor$  of the processes to behave arbitrarily, ensures deterministic termination and, contrary to prior solutions, does not, at any point, expect faulty processes to be responsive.

**2012 ACM Subject Classification** Theory of computation → Distributed algorithms

**Keywords and phrases** Byzantine Fault Tolerance, Partially Synchronous, Deterministic Termination, Randomness Beacon, Multi Party Computation, BFT-RNG

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2021.23

**Funding** Luciano Freitas de Souza was supported by Nomadic Labs, Petr Kuznetsov and Andrei Tonkikh – by TrustShare Innovation Chair.

## 1 Introduction

In a Byzantine fault-tolerant random number generator (BFT-RNG) protocol, a set of participating processes agree on a single random number that cannot be manipulated or halted, despite the presence of Byzantine failures, i.e., assuming that a faulty process may arbitrarily deviate from the prescribed algorithm. We distinguish between *commission* and *omission* failures [15]. Intuitively, a *commission* fault occurs when a process sends messages a correct process would not send, whereas an *omission* fault occurs when a process does not send messages a correct process would send.

A BFT-RNG protocol is typically divided into three phases:

1. **Generation and Commitment Phase** – each process locally generates some random value and then publicly commits to this value without revealing it.
2. **Reveal Phase** – the values previously committed are revealed.



© Luciano Freitas de Souza, Andrei Tonkikh, Sara Tucci-Piergiovanni, Renaud Sirdey, Oana Stan, Nicolas Quero, and Petr Kuznetsov;  
licensed under Creative Commons License CC-BY 4.0

25th International Conference on Principles of Distributed Systems (OPODIS 2021).

Editors: Quentin Bramas, Vincent Gramoli, and Alessia Milani; Article No. 23; pp. 23:1–23:16



Leibniz International Proceedings in Informatics

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

### 3. Computation Phase – using the values revealed, the processes decide on the resulting random number.

The idea is to make sure that at the moment the committed random values are revealed, it is already too late for the adversary to manipulate the output. Furthermore, assuming that the local random numbers are uniformly distributed, so should be the distribution of the output.

To the best of our knowledge, this paper describes the first partially synchronous BFT-RNG protocol that maintains optimal resilience (up to  $\lfloor \frac{N-1}{3} \rfloor$  Byzantine processes in a system of  $N$ ) that ensures *deterministic* termination. Unlike prior solutions [10, 31], our protocol does not expect that faulty processes remain responsive in the generation phase, i.e., it tolerates omission faults.

**State of the art.** In designing a BFT-RNG algorithm, we face two major challenges: (i) how to share random inputs despite omission failures, so that Byzantine processes cannot learn them before the reveal phase begins, and (ii) how to compute correct results despite commission faults of Byzantine processes. Existing protocols solve the first challenge by using techniques such as secret sharing [28], verifiable delay functions [4], threshold signatures [3, 5], and fully homomorphic encryption [13] and the second – by requiring a verifiable proof that a shared data was generated correctly.

*Techniques.* A  $(f, N)$ -secret sharing [28] scheme allows a process during the generation and commitment phase to share a secret  $s$  with  $N$  processes so that any subset of size  $f + 1$  among them can retrieve  $s$ , while no subset of  $f$  or less can. This way, even if a process refuses to disclose the original secret it has committed, the correct processes in the system can still reconstruct it in the reveal phase by using the shares they received earlier. Moreover, the values cannot be learned too early as the number of shares held by the Byzantine processes does not surpass  $f$ . *Threshold-signature* schemes, such as Schnorr [3] or BLS [5], are also very helpful in this context, as they allow to efficiently verify that a number of processes surpassing a given threshold agree with a certain value.

One can also make sure that the processes commit to a value without revealing it beforehand and provide a mechanism to retrieve commitments of Byzantine processes by using *verifiable delay functions* [4]. This technique guarantees that Byzantine processes cannot use the data shared by the correct processes to change their inputs and affect the result. Once a stipulated verifiable delay has expired, the correct processes can access the information presented by any process guaranteeing that the protocol is not halted.

The two homomorphic structures of most interest for BFT-RNG are Fully Homomorphic Encryption (FHE) [13] and homomorphic hashes. Given two sets  $A$  and  $B$ , a map  $f : A \rightarrow B$  is said to be  $(\circ)$ -homomorphic if it preserves an existing operation  $\circ$  on both sets:  $\forall x, y \in A, f(x \circ y) = f(x) \circ f(y)$  [6]. FHE allows processes to make operations in ciphertexts without knowing the plaintexts and can be then used instead of secret sharing for solving the same problem of preventing misbehaving parties from accessing data too early on and denying the access of correct participants to the data when it must be shared. As for homomorphic hashes, they are, as the name indicates, hash functions with homomorphic properties (i.e. by performing some operations over some data and their associated hashes, one obtains a result and a consistent associated hash). Homomorphic hashes allow to solve the second challenge of BFT-RNG design: they provide a mean to check that an operation was correctly executed by observing the hashes of the inputs and the hash of the outputs and can therefore contribute in detecting commission failures.

Other kinds of proofs of well formed data include Verifiable Random Functions (VRF) or Public Verifiable Secret Sharing (PVSS). VRF [21] are functions that once provided with an input  $x$ , output both a random number  $y$  and a proof  $\pi$  that allows any process using  $\pi$  to verify whether  $y$  was generated using  $x$  or not. Algorand’s VRF [14] uses a common coin (generated by the Algorand consensus) to correctly generate verifiable random numbers. PVSS-based proof [27] exchange together with secret shares some additional information that prove the data integrity without revealing any information of the original secret.

*Protocols.* In Table 1, which is a modified and expanded version of the table given in [26], we present a comparison including several existing BFT-RNG algorithms and the solution we present in this paper: **RandSolomon**. In some of these protocols, the networks (with  $N$  nodes) are partitioned into clusters of size  $c$ , this parameter appears in some of the complexity bounds given in the table.

■ **Table 1** Comparison of distributed RNG solutions.

RNG	Sync.	Vulnerability	Term.	Communication Complexity (Overall)	Computation Complexity (per process)	Resilience	Techniques
Cachin et al. [10]	A	Trusted key dealer	Det.	$O(N^2)$	$O(N)$	$f < \frac{N}{3}$	Unique threshold signatures (eg BLS)[5]
RandShare [31]	A	No omission in commit.	Det.	$O(N^3)$	$O(N^3)$	$f < \frac{N}{3}$	PVSS [27]
RandHound[31]	A	No omission in commit.	Prob.	$O(c^2 N)$	$O(c^2 N)$	$f < \frac{N}{3}$	PVSS [27] Multisignatures [3]
RandHerd[31]	A	No omission in commit.	Prob.	$O(c^2 \log N)$	$O(c^2 \log N)$	$f < \frac{N}{3}$	PVSS [27] Multisignatures [3]
SCRAPE[11]	S	None	Det.	$O(N^3)$	$O(N^2)$	$f < \frac{N}{2}$	PVSS [27]
DFFinity[16]	S	None	Prob.	$O(cN)$	$O(c)$	$f < \frac{N}{2}$	BLS signatures [5]
HydRand[26]	S	No omission in commit.	Det.	$O(N^2)$	$O(N)$	$f < \frac{N}{3}$	PVSS [27]
ProofOfDelay[8]	S	None	Det.	$O(N)$ + Ethereum	High	$f < \frac{N}{2}$	Delay functions [4]
No-Dealer[18]	S	None	Det.	$O(N^2)$	$O(N^2)$	$f < \frac{N}{2}$	Shamir [28] Homomorphic Hash
Nguyen et al.[22]	S	Trusted Requester	Prob.	$O(N)$	$O(1)$	$f < N$	FHE [13], VRF [21]
Ouroboros Praos[12]	P	Weaker properties	Det.	$O(N)$ + Ourob. Praos	$O(1)$ + Ourob. Praos	$f < \frac{N}{3}$	VRF [21]
Algorand[14]	P	Weaker properties	Prob.	$O(cN)$ + Algorand	$O(c)$ + Algorand	$f < \frac{N}{3}$	VRF [21]
<b>RandSolomon</b>	P	None	Det.	$O(N) \times$ Consensus	$O(N) \times$ Erasure Correcting Code	$f < \frac{N}{3}$	PK crypto ReedSolomon Retraceability

*Synchrony (Sync).* The second column of the comparison table shows which kind of synchrony the underlying system must provide in order to allow the deployment of each protocol. Here we distinguish A=Asynchronous, S=Synchronous and P=Partially Synchronous algorithms.

*Vulnerability.* It might seem impossible to have asynchronous implementations of BFT-RNG as we have already stated that this problem is impossible in the presence of at least one Byzantine participant in asynchronous systems [18]. Notice, one might introduce additional assumptions on the failure model for these solutions to exist.

This is the case with the solution by Cachin et al. [10] which assumes that there exists a special process capable of generating and distributing a key.

Other asynchronous solutions, such as RandShare, RandHound and RandHerd [31], assume that *every* entity initially publishes some information about their secret. The asynchronous protocols in [31] are therefore not fully BFT, as they do not tolerate omission failures in the generation phase. This assumption that Byzantine processes will not omit during the commitment phase of the protocol is also an exploitable vulnerability in the *synchronous* protocol HydRand [26], although it can be modified to restart once there are missing contributions. Nguyen et al.’s proposal [22], also a synchronous protocol, assumes a *Requester*, a trusted entity generating FHE keys, which can be considered as a client using the system.

Algorand [14] and Ouroboros Praos [12], maintain weak forms of RNG: common coin [14] and random beacon [12], RNG mechanisms in these protocols may not reach perfect agreement on the random value, and the coins values may be manipulated by the adversary to some extent or even be changed due to network asynchrony without affecting the correctness of their respective systems.

*Termination (Term).* A protocol ensures *deterministic termination* (Det) if it terminates in every execution, in contrast to *probabilistic termination* (Prob), when a protocol terminates with a fixed probability. RandHound, RandHerd [31] and Dfinity [16] allow a small probability, depending on the parameters of the system, of the Byzantine adversary fully corrupting a cluster, which results in prematurely halting the protocol. In the case of Algorand, a failure happens when the set (of expected cardinality  $c$ ) of nodes chosen to be proposers is empty. In the protocol by Nguyen et al. [22], this happens when all selected contributors are Byzantine.

*Complexity.* *Communication complexity* corresponds to the amount of messages exchanged and can be loosely translated into how many bits must be sent in the network for producing a result, while *Computation Complexity* measures how much time would it take to perform local computations given an input. In the table, we use term *High* to refer to the complexity of delay functions, which, though independent of the number of processes in the system (strictly speaking, their complexity is  $O(1)$ ), are very computationally heavy by design.

No-Dealer [18] specifies that the protocol must be restarted in case of certain Byzantine behavior, but does not include this fact in its complexity. As there are at most  $\frac{N}{2}$  Byzantine nodes, it might be necessary to restart this number of times, increasing their claimed complexity to the one presented in the table.

The protocol by Nguyen et al. [22] employs a summation on the secrets shared by the contributors, which results in linear computation complexity.

Finally, the two last columns **Resilience** and **Techniques** show how many Byzantine processes can be tolerated among the  $N$  participants and the main techniques employed in each solution.

**Contributions.** RandSolomon is the first BFT-RNG protocol providing deterministic termination in a partially synchronous system with  $f < \frac{N}{3}$  Byzantine processes, which is the optimal level of resilience [18]. Interestingly, the protocol relies only on standard cryptographic primitives: a public key infrastructure [25], block erasure correcting codes which can be interpreted as our version of secret-sharing [20] and standard digital signatures. The name of the protocol is inspired by the potential use of Reed-Solomon codes [24].

Our coding approach carries some similarities with SCRAPE [11] in the sense that they also recognised the potential of using codes such as Reed-Solomon to perform secret sharing. However the similarities stop there as, in RandSolomon, we not only propose a partially synchronous solution, but also introduce a new technique to cope with Byzantine commission

failures: *retraceability*, which circumvents the need for verification of the secret sharing. In a nutshell, we consider the secrets produced by Byzantine processes without checking their integrity until the last phase of the protocol, when we compute the final result. At this moment, we can retrace all the steps that should have been taken and detect a commission failure. This then results in discarding incorrectly formed data in order to ensure a correct result, based on the inputs of non-Byzantine processes.

## 2 Formal system model and properties

Before turning to the RandSolomon protocol description, let us first duly formalise the system model as well as a set of properties that a protocol must have to be considered a distributed Byzantine fault-tolerant random number generator.

Our system is made up of  $N$  nodes which run our protocol as a process which executes a prescribed sequence of steps. Among the participants, a portion  $f < \frac{N}{3}$  of them might be Byzantine who can collaborate with each other but have limited computing power.

The nodes can communicate with each other via messages that are sent through a point to point network. This network is available for all running processes and guarantees that if a message is sent through a channel, then it must be eventually delivered (in agreement with the partial-synchrony assumption). Whenever a process executes a broadcast it does so by just sending a message to every other process (we use a best effort broadcast).

Recall that in a BFT-RNG protocol, every process proceeds through clearly demarcated phases: (1) generation and commitment, (2) reveal, and (3) result computation. A phase begins with the first correct process entering it. In this setup, a BFT-RNG protocol satisfies the following properties:

- **Agreement.** Every correct process decides on the same random number;
- **Unpredictability.** Before the beginning of the reveal phase, no process can distinguish an execution that generates  $RAND$  as a random number, from an execution that generates  $RAND'$ , for any  $RAND' \neq RAND$ ;
- **Randomness.** The values decided by correct processes follow a uniform distribution;
- **Termination.** Eventually, every correct process decides on a value.

Although not an intrinsic property of *BFT-RNGs*, our protocol differs from existing protocols because it provides **retraceability**. It means that after the reveal phase, a process can verify that all the steps taken to generate the shared data used to produce the final random number were correctly followed.

## 3 The RandSolomon protocol

**Overview.** From a high-level viewpoint, the protocol aggregates enough locally generated random numbers, so that enough inputs are truly random and the final result observes all the properties desired. Numbers are produced locally, then encoded using an erasure correcting code and encrypted before sharing. All non-Byzantine processes agree on which numbers should be used by solving consensus, while the result remains secret (sealed under an encryption layer) as no process holds all the information necessary for computing it prior to the reveal phase. The protocol cannot be stopped by  $f$  (or less) Byzantine processes, as prior to the consensus the progress of correct processes depends solely on themselves and after it, thanks to our use of the erasure correcting code, the correct processes can retrieve data without using the information held by their Byzantine counterparts.

**Notation.** We shall use  $[N] = \{1, 2, \dots, N\}$ ,  $(\cdot)_i$  to indicate that the value enclosed by the parenthesis contains a signature of process  $p_i$  and  $\{\cdot\}_i$  to indicate that the value enclosed by the curly brackets was encrypted using  $p_i$ 's public key. Furthermore,  $b$  will denote the number of symbols in the encoded value to be encrypted in a given encryption key;  $z$  the size of the symbols used in a code;  $t$  is the number of erasures a code can correct;  $l$  is the length of a code;  $d$  the number of data symbols in a code.

### 3.1 Primitives

The system requires a *deterministic* encryption infrastructure where every process knows the public key of every other processes in the system, but each of them maintains its private key secret. *Deterministic* means here that at every time two processes encrypt the same number using the same key, they get the same result [2].

Although the use of deterministic encryption is crucial for the correct execution of the protocol, these primitives are used only to encrypt long-enough (at least 256 bits) sequences of uniformly random bits. As such, the source of randomness in cleartext mitigates the security issues which crop up when using deterministic encryption [23].

We use a *consensus* protocol to ensure that each correct process disposes of the same information. The consensus protocol used here must ensure that eventually every correct process outputs a value (Termination) and that not two correct processes outputs different values (Agreement). Further, the protocol must ensure *external validity* [9]: only a *valid* value can be output, i.e., the output must satisfy a predefined *valid* predicate:

► **Definition 1** (Predicate valid). *valid*( $v$ ) is true iff  $v$  contains  $N - f$  inputs signed by  $N - f$  different processes.

Any partially-synchronous algorithm that tolerates  $f$  Byzantine failures among  $3f + 1$  processes can be used [7, 32, 14].

Finally, let us consider a different perspective on secret sharing mechanisms [28]. In a classical Shamir secret-sharing protocol, when a dealer shares a secret  $s$  with  $N$  processes  $p_1, p_2, \dots, p_N$  using a threshold of  $N - t$ , it sends the shares  $s_1, s_2, \dots, s_N$  to their respective processes. Any  $N - t$  of these shares are sufficient to retrieve  $s$ , while less than  $N - t$  can reveal nothing on the secret in question. Indeed, one could consider the string  $s_1 s_2 \dots s_N$  as a code, the non-received values as erasures and hence conclude that, in fact, the secret sharing scheme can be also analysed as an Erasure Correcting Code capable of correcting  $t$  erasures [20].

In Information Theory, the number of substitutions required to change one string into another is known as *Hamming Distance* [19]. We can then conclude that we need in fact an Erasure Correcting Code with Hamming distance at least  $t + 1$ . The class of error-erasure correcting codes known as *Reed-Solomon (RS)*[24] with the required distance is capable of correcting  $t$  **erasures** (notice we do not treat it as an error correcting code, but an erasure correcting: an error correcting code is capable of correcting a string with corrupted data placed in unknown locations, while an erasure correcting code needs to know the positions of the string which were corrupted). Therefore, this class provides optimal block size known as *Singleton Bound* [29]. From a more pragmatic viewpoint, Reed-Solomon codes have free library implementations in many programming languages, they have deterministic parameters and encoding which are ideal for our requirements. Furthermore, most applications running our protocol will have relatively small block sizes and one can enhance the performances through hardware implementations [17]. It should be noted however, that any code complying with the following *Abstract Code* requirements can be used in our protocol.

**Abstract Code.**

- Have a code-word of size  $b \times N$  symbols;
- Be able to correct up to  $b \times f$  symbol erasures;
- $b \times z \geq 256$

Considering that we make use of Reed-Solomon codes we briefly present their general parameters:

**Abstract Reed-Solomon code.**

- The symbols have size  $z$  bits
- The data has length  $d$  symbols
- The code-word has length  $l$  where  $l \leq 2^z - 1$  symbols
- It can correct up to  $t$  erasures where,  $t = l - d$

Adjusting the above *Abstract RS code* to match the *Abstract code* and the system requirements, leads to the following Concrete Reed-Solomon Code which is suitable for implementing our protocol.

**Concrete Reed-Solomon code.**

- The symbols have size  $z$  bits;
- Each block to be encrypted has a size  $b$  of at least  $\frac{256}{z}$  symbols;
- The data has a length of  $b(N - f)$ -symbols;
- The code-word has a length of  $b \times N$  symbols.

It should be noted that as our protocol allows correct processes to retrace the execution followed by Byzantine processes and detect when they generate incorrect messages, we can use erasure correction instead of error correction. This drastically improves the coding performance as every error-erasure correcting code can correct two times more erasures than errors. This has two implications on our protocol: first we need fewer parity bits; second, if we were to unnecessarily use the code for errors correction, the protocol would only tolerate up to  $\lfloor \frac{N-1}{4} \rfloor$  Byzantine processes. The reason for the potential loss of resilience comes from the fact that we would need to correct  $2f$  errors:  $f$  errors introduced by the Byzantine member during the generation and  $f$  more for the missing blocks due to asynchrony. Therefore the number of parity blocks would have to be at least  $2(2f) = 4f$  blocks, while the code must have length  $N$  blocks. Because the length of a code is larger than the number of parity symbols,  $N > 4f$ . This illustrates the contribution of retraceability: it implies simpler data reception by eliminating the need to generate proofs and to check them, and guarantees better resilience whilst maintaining the correctness of the protocol.

## 3.2 Algorithm

**Generation and commitment.** Each process  $p_i$  taking part in the protocol begins by generating a random number  $r_i$  of  $b(N - f)$  symbols and encoding it using a Reed-Solomon encoder complying with the specification given in subsection 3.1 obtaining a number  $s_i$  of  $b \times N$  symbols (lines 1, 2). This encoded number  $s_i$  is then split in  $N$  blocks of  $b$  symbols and each of these blocks are encrypted using the public key of the different processes in the system in order, signing the final result and obtaining the variable  $s_i$  (line 3).

Each process  $p_i$  share their  $s_i$  (line 4) and collect  $N - f$  numbers of this type, coming from  $N - f$  distinct processes according to their signatures. With this set of  $N - f$ -numbers they can engage in consensus and learn the same set, say  $RNL$ , of  $(N - f)$  numbers generated by  $N - f$  distinct processes (line 6).

■ **Algorithm 1** RandSolomon code for process  $p_i$ .

---

*Each function is entirely executed before executing the next*

**Static Local Variables:**

$RNL := \emptyset$ : set of encoded and encrypted shared random numbers learnt in Consensus  
 $SEEN := \emptyset$ : map where the key is the index of a process and the value is the value it produced  
 $\sigma_i[1..N][1..N] := \perp$ : array of plain random number shares used in reconstruction  
 $RAND_i := 0$ : random number decided by  $p_i$

**{Generation and Commitment Phase}**

```

1   Generate random number  $r_i$  of  $b(N - f)$  symbols of  $z$ -bits
2   Encode  $r_i$  into  $s_i$  with Desired RS
3    $\underline{s}_i = (\{s_i[1]\}_1, \{s_i[2]\}_2, \dots, \{s_i[N]\}_N)_i$ 
4   Broadcast  $\langle GENERATED, \underline{s}_i \rangle$ 

```

**upon** receiving  $\langle GENERATED, \underline{s}_j \rangle$

```

5    $SEEN[j] := \underline{s}_j$ 
6   if  $|SEEN| = N - f$  then  $RNL := Consensus(SEEN)$ 

```

**{Reveal Phase}**

**upon**  $RNL \neq \emptyset$

```

7    $\forall \underline{s}_j \in RNL$  do
8       Decrypt  $\underline{s}_j[i]$  from  $\underline{s}_j$  into  $s_j[i]$ 
9        $\sigma_i[j][i] := s_j[i]$ 
10  Broadcast  $\langle REVEAL, (\sigma_i[:,i])_i \rangle$ 

```

**upon** receiving  $\langle REVEAL, (\sigma_j)_j \rangle$ ,  $j \neq i$  **execute after**  $RNL \neq \emptyset$

```

11   $\forall \underline{s}_k \in RNL$  do
12      if  $\{\sigma_j[k][j]\}_j = \underline{s}_k[j]$  from  $\underline{s}_k$  then  $\sigma_i[k][j] := \sigma_j[k][j]$ 

```

**{Result Computation Phase}**

**upon**  $RNL \neq \emptyset \wedge \forall \underline{s}_j \in RNL, \exists K \subseteq [N], |K| = N - f : \sigma_i[j][k] \neq \perp$

```

13   $step := 0$ 
14   $PRE := 0$ 
15   $\forall \underline{s}_j \in RNL$  sorted by  $j$  do
16      Decode  $\sigma_i[j]$  into  $\tilde{r}_j$  using Desired RS
17      if  $\tilde{r}_j$  encoded with Desired RS and blockwise encrypted doesn't match  $\underline{s}_j$ 
          then  $\tilde{r}_j := 0$ 
          {Circular right shift by  $step$  blocks or  $b \times step$  symbols}
18       $PRE := PRE \oplus (\tilde{r}_j \ggg step)$ 
19       $step++$ 
{XOR blocks pairwise with triple in the end if necessary}
20  for  $k := 1; 2k - 1 < N - f; k := k + 2$ 
21       $RAND_i[k] := PRE[2k - 1] \oplus PRE[2k]$ 
22  if  $2k - 1 = N - f$  then  $RAND_i[k] := RAND_i[k] \oplus PRE[N - f]$ 
23  Decide  $RAND_i$ 

```

---



**Reveal.** After obtaining the *RNL* set, each process can decrypt the blocks it is responsible for (line 8) and reveal them to the system via a broadcast (line 10).

(A best-effort broadcast in which a process simply sends the message to every other process will suffice.)

The processes gather the shares necessary for decoding the erasure correcting code, making sure that they truly are the decrypted versions of the RNL shares (line 12).

**Result computation.** Once a process has gathered at least  $N - f$  shares of each of the numbers in the RNL set, it can reconstruct all of them (line 16). If the decoded version  $\tilde{r}_j$  of a RNL number is again encoded and encrypted, leading to the same value for  $s_j$ , then this implies that any  $N - f$  shares obtained by any correct process will give the same  $\tilde{r}_j$  making it consistent to be used in the final step computations.

**Importance of verification.** Notice that if  $p_i$  is Byzantine, then it can generate a number  $r_i$  and insert  $f$  blocks with errors in  $s_i$ . By colluding with other Byzantine processes in the system, a correct process  $p_j$  might get no response from  $f$  Byzantines and get these  $f$  erroneous blocks, essentially receiving a number with  $2f$  incorrect blocks, which leads it to decode a number  $\tilde{r}'_i \neq r_i$ . Meanwhile a process  $p_k$  can get the Byzantine processes' correct shares instead of the blocks with errors, decoding  $\tilde{r}''_i = r_i$ , which would lead these two different correct processes producing two different random numbers in the end. This attack is nullified by the simple verification done in the line 17 and setting this number produced by a Byzantine process to 0, which is done by every process. It should be noted that because at least  $N - f$  numbers are used and that there are at most  $f$  Byzantines, at least  $f + 1$  numbers will not be nullified.

**Cyclic XOR.** Finally the correct processes will hold the same decoded versions of the RNL numbers which are well formed and can produce the same final random number by first cyclically shifting each number to the right by increasing steps of blocks (remember a block has  $b$  symbols) and then taking an XOR of them (line 18). Here, the reason for the shift is that for Byzantine processes might know the full contents of up to  $f$  numbers and  $f$  positions from each of the other numbers before the reveal phase. Assuming all the numbers produced by Byzantines were chosen, then the shift ensures that at least  $f + 1$  different positions from the numbers created by correct processes will be used, hence including at least one unknown value for the malicious participant before the reveal.

**Pairwise (triple) XOR and decision.** The final step is to XOR the last three blocks together and the remaining blocks pairwise when  $N - f$  is odd and XOR all the blocks pairwise when  $N - f$  is even. Suppose this last step was not taken and the shifted XOR blocks were returned. Then if the  $2f$  positions known by the Byzantine could potentially be used in the computation of a position  $pos$  in the result and these blocks XOR to a value  $x$ , they can assure that by promoting any unknown value different than  $x \oplus y$  to be the last operand used in  $pos$  assures that the value  $y$  will not appear in  $pos$ . Because of the deterministic encryption they can immediately check the candidate values for being different than  $x \oplus y$ , although it is computationally unfeasible to determine their value. In our solution, however, because we guarantee that the Byzantine do not know at least two values used, there are  $2^{b \times z}$  pairs that XOR to any given value and it is unfeasible to test the two values for being different than all of them (as  $b \times z \geq 256$  in real scale instantiations of the protocol), let alone read, which would take  $2^{2 \times b \times z}$  tests.

### 3.3 Execution example

We present now an example of a possible execution of our protocol with one Byzantine process and four processes in total illustrated in Figure 1. For pedagogical reasons we assume that the symbols have 8-bits and that each block to be encrypted contains 1 symbol ( $b = 1, z = 8$ ), relaxing the requirement that  $b \times z \geq 256$ .

The beginning of the protocol and the *Generation and Commitment Phase*, corresponding to lines from lines 1 to 3 of the algorithm is shown in Figure 1a. The correct processes  $p_1, p_2$  and  $p_3$  produce each a 3 bytes random number, correctly encoding into a 4 bytes reed-solomon codeword. The values  $s_1, s_2, s_3$  ready to be shared are obtained by encrypting each of the 4 bytes from the codewords with the public keys of the  $p_1, p_2, p_3$  and  $p_4$ , respectively. On the other hand, process  $p_4$ , who is Byzantine, maliciously produces two bad values:  $s'_4$  with an error in its third byte and  $s''_4$  with an error on its second byte.

Figure 1b then shows lines 4 and 5 where processes share their produced values and collect values coming from other processes. Notice that contrary to correct processes, Byzantine processes might send different values to different destinations.

Once each process has gathered three ( $N-f$ ) different values, they propose what they know to the consensus component (line 6 and Figure 1c). Nothing prevents the Byzantine process  $p_4$  of making more than one proposal to consensus, but any proposal which is not composed by  $N - f$  signatures is discarded. Once the consensus algorithm terminates, any valid value might be returned, but all processes will get the same result (decided value equal to  $s_1, s_2$  and  $s'_4$ ).

The *Reveal Phase* illustrated in Figure 1d then begins, comprising lines 7 to 12. At this point processes openly share the symbols that were previously encrypted in their public keys. One deviation Byzantine processes might do is to send wrong numbers that do not correspond to the agreed values counterparts. However, because of the deterministic encryption, the receiver can detect it by asserting that the encrypted version does not match the plain value received and discard it. Moreover, even if the Byzantine process does not send its share to every participant it does not matter, as  $N - f$  shares are available nonetheless.

Once processes gather three shares for each of the numbers agreed upon in consensus they can start the *Result Computation Phase* executing lines 15 to 23. Figure 1e shows how they first obtain the decoded version of the numbers and then redo both the reed-solomon encoding and the encryption of the blocks to check that they correspond to the value decided in consensus. At this point they discard the value generated by  $p_4$  nullifying its contribution and computing the final random number by XORing the other values as shown in Figure 1f.

On the right column of the same figure we can see that the processes sort the agreed numbers by their origin, in this case they take  $r_1, r_2$  and  $r''_4$  in this order. They proceed by cyclically shifting the first number by 0 blocks, the second by 1 block and the last by 2 blocks. They obtain the same number  $DD, 81, 8B$  and produce the same final random number  $D7$  by XORing all three blocks, as these are the last three blocks. Note that if the system had  $f = 3$  and  $N = 10$ , for example, the result from the cyclic XOR would have  $N - f = 7$  blocks  $B_1|B_2|B_3|B_4|B_5|B_6|B_7$  and the final random number would have three blocks:  $B_1 \oplus B_2|B_3 \oplus B_4|B_5 \oplus B_6 \oplus B_7$ .

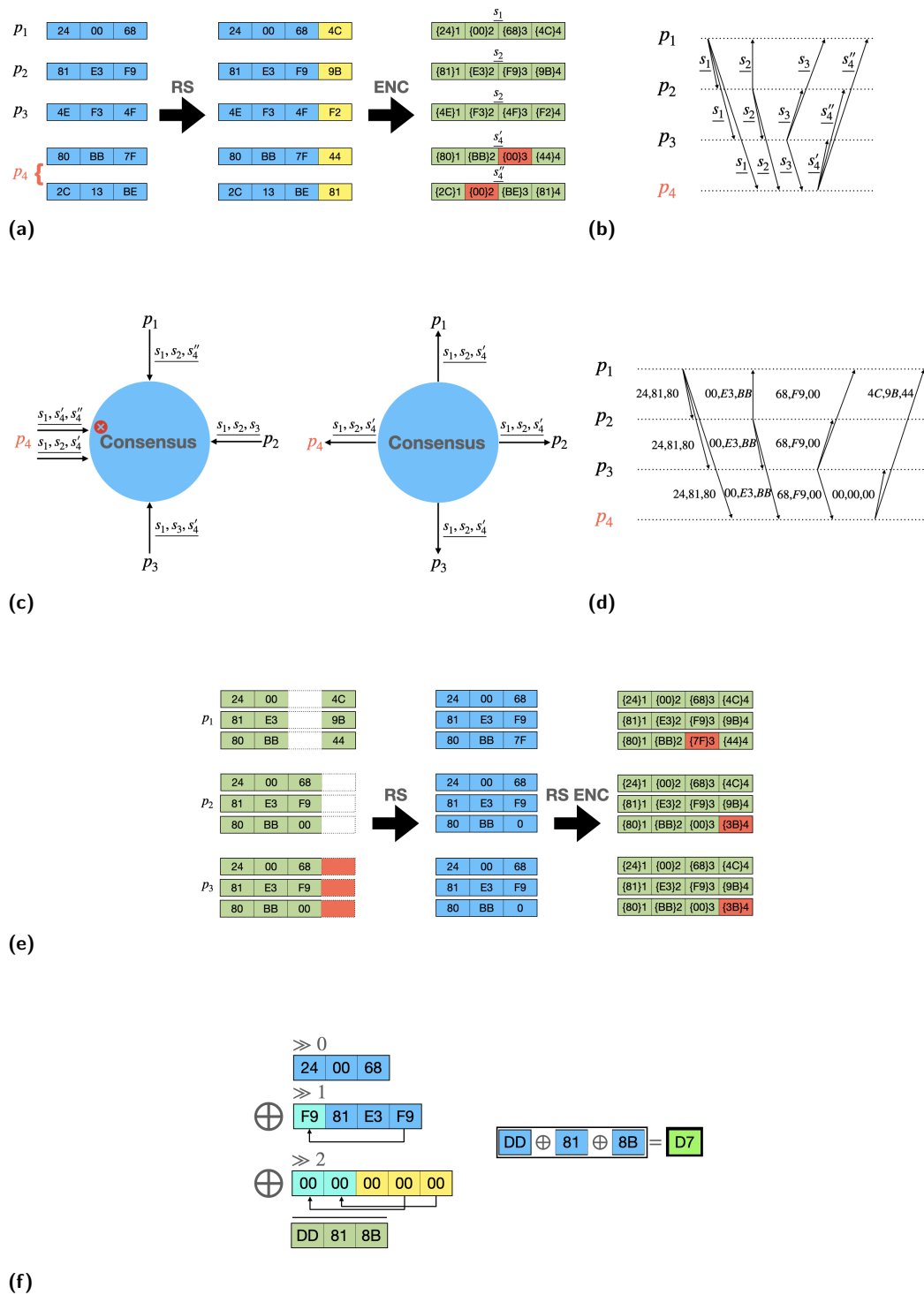


Figure 1 Example of a RandSolomon execution with one Byzantine process among a system of 4 processes.

## 4 Formal analysis of the protocol

### 4.1 Correctness

This section is devoted to the proof that `RandSolomon` is a correct partially-synchronous BFT-RNG. We do so by showing that the protocol satisfies the set of properties stated in Section 2.

► **Proposition 1.** *RandSolomon achieves Agreement.*

**Proof.** Because of the consensus using the external validity property, every correct process has the same *RNL* set. Correct processes then use shares that have been verified and match the values agreed upon (line 12), allowing them to only access the original values generated in line 2.

If a RNL number  $s_j$  passes the test in line 17, any  $N - f$  correctly decrypted shares of this number shall yield the same number, as the encoded value contains no errors. It follows that every correct process will only use correctly decrypted shares and every correct process will hold the same number  $\tilde{r}_j$  which will pass the test by our hypothesis.

If, however, this RNL number  $s_j$  does not pass the test, then there is an error in its encoding, as the test is merely checking if it was correctly done, and it will be visible to all correct processes in the system which will all proceed to ignore this number.

Therefore all  $RAND_i$  are equal, as they are formed by XORing and shifting the same RNL numbers which every correct process agrees upon. ◀

► **Proposition 2.** *RandSolomon achieves Unpredictability.*

**Proof.** A process with limited computational power has negligible probability of determining the plain value corresponding to an encrypted value it does not possess the decryption key of. It can however test that it does not correspond to a certain value.

If Byzantine processes collude and share each others values before the different processes agree on which  $N - f$  values at the end of the generation phase will compose the final result, they will know at most  $f$  full values. They will also possess  $f$  shares of each of the remaining  $f + 1$  chosen values corresponding to their positions but it is impossible for them to get any more shares prior to correct processes entering the reveal phase and sending them this information. Thus, they cannot determine the value of any given position in the decided value as the shifts makes so that at least  $2f + 1$  positions from the operands are needed in order to determine a position from the result and as established, the Byzantine can know at most  $2f$  of them. It can still determine that the result is different than some specific value though, but as each position is then determined by the XORed with at least one other position, this possibility is then nullified as it would require the Byzantine processes to test  $2^{b \times z}$  pairs of numbers in order to eliminate a value, which is computationally unfeasible with real scale protocol parameters ( $b \times z \geq 256$ ). ◀

► **Proposition 3.** *RandSolomon achieves Randomness.*

**Proof.** By hypothesis, correct processes are capable of generating uniformly random numbers. The result of XORing a uniformly distributed random variable  $X$  in  $D$  with a constant  $c$  in  $D$  is a uniformly distributed random variable in  $D$ . Also, the result of XORing two independent uniformly distributed variables  $X$  and  $Y$  over  $D$  is uniformly distributed. As we already established in the final two paragraphs of subsection 3.2, each position in the final result is independent from each other and uses at least two uniform random numbers coming

from correct processes unknown to the Byzantine before the reveal phase. This means no proposed values are preferred over others and the randomness of the operands is transferred to the output. ◀

► **Proposition 4.** *RandSolomon achieves Termination.*

**Proof.** Every correct process generates their random numbers and propose a set of  $N - f$  of them to the consensus component. This means that there will be at least  $N - f$  processes engaging in it, and because it can tolerate up to  $f$  failures, it will eventually give all correct processes their RNL sets.

Once  $N - f$  correct processes learn what the RNL set is, they will share their shards, meaning that each correct process is guaranteed to receive at least  $N - f$  correct shares of each of their RNL numbers, satisfying the conditions for entering the computation phase, where their progress becomes purely local as they do not depend on other processes anymore. ◀

## 4.2 Complexity

We shall analyse our algorithm in terms of *message complexity*: the maximum number of messages transmitted per random number generated; *bit complexity*: the maximum number of bits exchanged over the network per random bit generated; *time complexity*: the number of message round trips required per random number generated; and *computational complexity*: the number of operations to be executed per process per random number generated.

In the generation and commitment phase, each process executes one broadcast, meaning that there are  $O(N^2)$  messages being sent at this phase. After consensus is reached on the value of *RNL*, each process executes exactly one more broadcast, leaving the message complexity of this part of the protocol on  $O(N^2)$ . The result computation phase is done locally. Hence the message complexity of our protocol is  $O(N^2)$  outside consensus.

In terms of bit complexity, *RandSolomon* produces random numbers of  $O(N)$  bits, therefore we consider the number of bits exchanged divided by  $N$ . The messages of the generation phase contain random numbers whose lengths are proportional to the number of processes in the system by design. Therefore, the bit complexity of this step is  $O(N^2)$ . Afterwards in the reveal phase, each process includes one decrypted block per number in the *RNL* set. Each decrypted block has constant size and the cardinality of *RNL* is  $f + 1$ , so the bit complexity of this stage is also  $O(N^2)$ . Therefore, without taking consensus into account, the bit complexity of our protocol is  $O(N^2)$ . The inputs for consensus are comprised of  $N - f$  values of  $O(N)$  bits and therefore the bit complexity (used in the Table 1) is  $O(N) \times \text{Consensus}$ .

With respect to time complexity, our protocol requires outside consensus two message delays given the two aforementioned broadcasts, each executed by all processes in parallel. Consensus might require *view-changes* in the worst case bringing its time complexity to  $O(f)$  which corresponds to our overall time complexity. As for computational complexity we present the analysis split on the three phases of the protocol in Table 2.

■ **Table 2** Computational complexity.

Operation	Generation	Reveal	Result
Encryption	$O(N)$	$O(N^2)$	$O(N^2)$
Decryption	0	$O(N)$	0
ECC encoding	$O(1)$	0	$O(N)$
ECC decoding	0	0	$O(N)$

If the erasure correcting code used is indeed Reed-Solomon, then the encoding and decoding complexities of a single number with length  $O(N)$  is  $O(N \log N)$  [30], meaning that the per-process computational complexity is  $O(N^2 \log N)$  when this particular code is used.

When considering the complexity of the Consensus protocol, one can easily adopt last generation PBFT consensus protocols developed in the context of blockchain-type ledgers. In this context, the Tendermint (analysed in detail in [1]) or Hotstuff [32] consensus protocols can be used within RandSolomon. Doing so leads to an overall message complexity of  $O(N^2)$  and bit complexity of  $O(N^3)$  accounting view-changes with the complexity of consensus dominating that of our protocol for both protocols considered. As such, any system which already has the protocol machinery to solve consensus can implement RandSolomon without incurring a significant performance impact.

In a run where the system is synchronous (after passed GST) and the consensus leader is correct, the protocol terminates in constant number of message delays and incurs only  $O(N^2)$  bit complexity, comparable to that of synchronous protocols.

The complexity analysis of RandSolomon is summarised in Table 3.

■ **Table 3** RandSolomon Protocol complexities integrating Consensus as in [32].

Complexity	Generation	Consensus	Reveal	Result	Total
Message	$O(N^2)$	$O(N^2)$	$O(N^2)$	0	$O(N^2)$
Bit	$O(N^2)$	$O(N^3)$	$O(N^2)$	0	$O(N^3)$
Time	1 msg delay	$O(f)$	1 msg delay	0	$O(f)$
Computation	$O(N)$	$O(N)$	$O(N^2)$	$O(N^2 \log N)$	$O(N^2 \log N)$

## 5 Conclusion

We presented RandSolomon, a Byzantine fault-tolerant protocol capable of generating a common random number in a partially-synchronous system. As we have previously shown in section 1, although the problem of generating randomness in multi-party systems has already been extensively discussed, the partially-synchronous systems still lacked a BFT solution with the optimal resilience of  $f$  Byzantine participants among  $3f + 1$  with deterministic termination. Not only did we provide such a solution but we also employed very simple public key cryptography, not relying on a random oracle, by means of what we have called *retraceability*. Our approach is modular, using Consensus as a black box, which facilitates future implementations of the protocol with improved complexity metrics.

---

## References

- 1 Yackolley Amoussou-Guenou, Antonella Del Pozzo, Maria Potop-Butucaru, and Sara Tucci-Piergiovanni. Dissecting tendermint. In *International Conference on Networked Systems*, pages 166–182. Springer, 2019.
- 2 Mihir Bellare, Alexandra Boldyreva, and Adam O’Neill. Deterministic and efficiently searchable encryption. In *Annual International Cryptology Conference*, pages 535–552. Springer, 2007.
- 3 Mihir Bellare and Gregory Neven. Multi-signatures in the plain public-key model and a general forking lemma. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 390–399, 2006.
- 4 Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. Verifiable delay functions. In *Annual international cryptology conference*, pages 757–788. Springer, 2018.
- 5 Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In *International conference on the theory and application of cryptology and information security*, pages 514–532. Springer, 2001.

- 6 Ilja N Bronshtein and Konstantin A Semendyayev. *Handbook of mathematics*. Springer Science & Business Media, 2013.
- 7 Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on bft consensus. *arXiv preprint*, 2018. [arXiv:1807.04938](https://arxiv.org/abs/1807.04938).
- 8 Benedikt Bünz, Steven Goldfeder, and Joseph Bonneau. Proofs-of-delay and randomness beacons in ethereum. *IEEE Security and Privacy on the blockchain (IEEE S&B)*, 2017.
- 9 Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *Annual International Cryptology Conference*, pages 524–541. Springer, 2001.
- 10 Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in Constantinople: Practical asynchronous Byzantine agreement using cryptography. Cryptology ePrint Archive, Report 2000/034, 2000. URL: <https://eprint.iacr.org/2000/034>.
- 11 Ignacio Cascudo and Bernardo David. Scrape: Scalable randomness attested by public entities. Cryptology ePrint Archive, Report 2017/216, 2017. URL: <https://eprint.iacr.org/2017/216>.
- 12 Bernardo David, Peter Gaži, Aggelos Kiayias, and Alexander Russell. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 66–98. Springer, 2018.
- 13 Craig Gentry. *A fully homomorphic encryption scheme*. Stanford university, 2009.
- 14 Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nikolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 51–68, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3132747.3132757.
- 15 Andreas Haeberlen and Petr Kuznetsov. The fault detection problem. In *International Conference On Principles Of Distributed Systems*, pages 99–114. Springer, 2009.
- 16 Timo Hanke, Mahnush Movahedi, and Dominic Williams. DFINITY technology overview series, consensus system. *CoRR*, abs/1805.04548, 2018. [arXiv:1805.04548](https://arxiv.org/abs/1805.04548).
- 17 MA Khan, S Afzal, and R Manzoor. Hardware implementation of shortened (48, 38) reed solomon forward error correcting code. In *7th International Multi Topic Conference, 2003. INMIC 2003.*, pages 90–95. IEEE, 2003.
- 18 Mikhail Krasnoselskii, Grigorii Melnikov, and Yury Yanovich. No-dealer: Byzantine fault-tolerant random number generator. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHOPS)*, pages 568–573. IEEE, 2020.
- 19 Florence Jessie MacWilliams and Neil James Alexander Sloane. *The theory of error correcting codes*, volume 16. Elsevier, 1977.
- 20 Robert J. McEliece and Dilip V. Sarwate. On sharing secrets and Reed-Solomon codes. *Communications of the ACM*, 24(9):583–584, 1981.
- 21 Silvio Micali, Michael Rabin, and Salil Vadhan. Verifiable random functions. In *40th annual symposium on foundations of computer science (cat. No. 99CB37039)*, pages 120–130. IEEE, 1999.
- 22 Thanh Nguyen-Van, Tuan Nguyen-Anh, Tien-Dat Le, Minh-Phuoc Nguyen-Ho, Tuong Nguyen-Van, Nhat-Quang Le, and Khuong Nguyen-An. Scalable distributed random number generation based on homomorphic encryption. In *2019 IEEE International Conference on Blockchain (Blockchain)*, pages 572–579. IEEE, 2019.
- 23 Charles Rackoff and Daniel R Simon. Non-interactive zero-knowledge proof of knowledge and chosen ciphertext attack. In *Annual International Cryptology Conference*, pages 433–444. Springer, 1991.
- 24 Irving S Reed and Gustave Solomon. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics*, 8(2):300–304, 1960.
- 25 Arto Salomaa. *Public-key cryptography*. Springer Science & Business Media, 2013.

- 26 Philipp Schindler, Aljosha Judmayer, Nicholas Stifter, and Edgar Weippl. Hydrand: Practical continuous distributed randomness. Cryptology ePrint Archive, Report 2018/319, 2018. URL: <https://eprint.iacr.org/2018/319>.
- 27 Berry Schoenmakers. A simple publicly verifiable secret sharing scheme and its application to electronic voting. In *Annual International Cryptology Conference*, pages 148–164. Springer, 1999.
- 28 Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.
- 29 Richard Singleton. Maximum distance q-nary codes. *IEEE Transactions on Information Theory*, 10(2):116–118, 1964.
- 30 Alexandre Soro and Jérôme Lacan. FNT-based Reed-Solomon erasure codes. In *2010 7th IEEE Consumer Communications and Networking Conference*, pages 1–5. IEEE, 2010.
- 31 Ewa Syta, Philipp Jovanovic, Eleftherios Kokoris Kogias, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Michael J Fischer, and Bryan Ford. Scalable bias-resistant distributed randomness. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 444–460. Ieee, 2017.
- 32 Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 347–356, 2019.



# Optimal Space Lower Bound for Deterministic Self-Stabilizing Leader Election Algorithms

Lélia Blin  

Sorbonne Université, Université d'Evry-Val-d'Essonne, CNRS, LIP6 UMR 7606, 4 place Jussieu, 75005 Paris, France

Laurent Feuilloley  

Univ. Lyon, Université Lyon 1, LIRIS UMR CNRS 5205, F-69621, Lyon, France

Gabriel Le Bouder 

Sorbonne Université, CNRS, INRIA, LIP6 UMR 7606, 4 place Jussieu, 75005 Paris, France

---

## Abstract

Given a boolean predicate  $\Pi$  on labeled networks (e.g., proper coloring, leader election, etc.), a self-stabilizing algorithm for  $\Pi$  is a distributed algorithm that can start from any initial configuration of the network (i.e., every node has an arbitrary value assigned to each of its variables), and eventually converge to a configuration satisfying  $\Pi$ . It is known that leader election does not have a deterministic self-stabilizing algorithm using a constant-size register at each node, i.e., for some networks, some of their nodes must have registers whose sizes grow with the size  $n$  of the networks. On the other hand, it is also known that leader election can be solved by a deterministic self-stabilizing algorithm using registers of  $O(\log \log n)$  bits per node in any  $n$ -node bounded-degree network. We show that this latter space complexity is optimal. Specifically, we prove that every deterministic self-stabilizing algorithm solving leader election must use  $\Omega(\log \log n)$ -bit per node registers in some  $n$ -node networks. In addition, we show that our lower bounds go beyond leader election, and apply to all problems that cannot be solved by anonymous algorithms.

**2012 ACM Subject Classification** Theory of computation  $\rightarrow$  Distributed computing models

**Keywords and phrases** Space lower bound, memory tight bound, self-stabilization, leader election, anonymous, identifiers, state model, ring topology

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2021.24

**Funding** Support by ANR ESTATE (ANR-16-CE25-0009-03) and ANR GrR (ANR-18-CE40-0032).

**Acknowledgements** We thank the reviewers for their useful comments.

## 1 Introduction

### 1.1 Context

Self-stabilization is a paradigm suited to asynchronous distributed systems prone to transient failures. The occurrence of such a failure (e.g., memory corruption) may move the system to an arbitrary configuration. An algorithm is self-stabilizing if it guarantees that whenever the system is in a configuration that is illegal w.r.t. some given boolean predicate  $\Pi$ , the system returns to a legal configuration in finite time (and remains in legal configuration as long as no other failures occur). In this paper, we study self-stabilization in networks. The network is modeled as a graph, and we consider predicates defined on labeled graphs. For instance, in proper  $k$ -coloring, a configuration is legal if every node is labeled by a color  $\{1, \dots, k\}$  that is different from the colors of all its neighbors. Given a boolean predicate  $\Pi$ , a self-stabilizing algorithm for  $\Pi$  is a distributed algorithm enabling every node, given any input state, to construct a label such that the resulting labeled graph satisfies  $\Pi$ .



© Lélia Blin, Laurent Feuilloley, and Gabriel Le Bouder;  
licensed under Creative Commons License CC-BY 4.0

25th International Conference on Principles of Distributed Systems (OPODIS 2021).

Editors: Quentin Bramas, Vincent Gramoli, and Alessia Milani; Article No. 24; pp. 24:1–24:12

Leibniz International Proceedings in Informatics



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

During the execution of a self-stabilizing algorithm, the nodes exchange information along the links of the network, and this information is stored locally at every node. Specifically, processes in a distributed system have two types of memory: the *persistent* memory, and the *mutable* memory. The persistent memory is used to store the identity of the process (e.g., its globally unique MAC address), its port numbers, and the code of the algorithm executed on the process. Importantly, this section of the memory is not write enabled during the execution of the algorithm. As a consequence it is less likely to be corruptible, and most work in self-stabilization assumes that this part of the memory is not subject to failures. The mutable memory is used to store the variables used by the algorithm, and is subject to failures, that is, to the corruption of these variables. The space complexity of a self-stabilizing algorithm is the total size of all the variables used by the algorithm, including those used to encode the output label of the node. For instance, the space complexity of the algorithm for  $k$ -coloring is at least  $\Omega(\log k)$  bits per node, for encoding the colors in  $\{1, \dots, k\}$ . The question addressed in this paper is: under which circumstances is it possible to reach a space complexity as low as the size of the labels? And if not, what is the smallest space complexity that can be achieved?

Preserving small space complexity is indeed very much desirable, for several reasons. First, it is expected that self-stabilizing algorithms offer some form of universality, in the sense that they are executable on several types of networks. Networks of sensors as used in IoT, as well as networks of robots as used in swarm robotics, have the property to involve nodes with limited memory capacity, and distributed algorithms of large space complexity may not be executable on these types of networks. Second, a small space complexity is the guarantee to consume a small bandwidth when nodes exchange information, thus reducing the overhead due to link congestion [1]. In fact, a self-stabilizing algorithm is never terminating, in the sense that it keeps running in the background in case a failure occurs, for helping the system to return to a legal configuration. Therefore, nodes may be perpetually exchanging information, even after stabilization, and even when no faults occur. Limiting the amount exchanged information, and thus, in particular, the size of the variables, is therefore of the utmost importance for optimizing time, and even energy. Last but not least, increasing robustness against variable corruption can be achieved by data replication [17]. This is however doable only if the variables are reasonably small. Said otherwise, for a given memory capacity, the smaller the space complexity the larger the robustness thanks to data replication.

## 1.2 Contributions of the paper

In this paper, we focus on one of the arguably most important problems in the context of distributed computing, namely *leader election*. The objective is to maintain a unique leader in the network, and to enable the network to return to a configuration with a unique leader in case there are either zero or more than one leaders. Interestingly, encoding legal states consumes one single bit at each node. Indeed, in leader election, every node has a label with value 0 or 1, and these labels form a legal configuration if there is one and only one node with label 1. As a consequence, up to an additive constant, the space complexity is exactly the space used to encode the variables of the algorithm (which is not the case for other problems where the output itself uses some non-trivial space to be encoded).

We establish the lower bound of  $\Omega(\log \log n)$  bits per node for the space complexity of leader election. This improves the only lower bound known so far (see [3]), which states that leader election has non-constant space complexity, i.e., complexity  $\omega(1)$ , where  $f = \omega(g)$  if  $g(n)/f(n) \rightarrow 0$  when  $n \rightarrow \infty$ . More importantly, our bound matches the best known upper

bound on the space complexity of leader election, which is  $O(\log \log n)$  bits per node in bounded degree networks [6], and in particular invalidates the folklore conjecture stating that leader election is solvable using only  $O(\log^* n)$  bits of mutable memory per node.

We obtain our lower bound by establishing an interesting connection between self-stabilizing algorithms with small space complexity, and self-stabilizing algorithms performing in anonymous networks, that is, in networks in which nodes have no identifiers. (Recall that space complexity counts solely the size of the mutable memory, and does not include the immutable persistent memory where the identifiers are stored). More specifically, the technical ingredient used for establishing our results are the following. It is known that many self-stabilization problems, including vertex coloring, leader election, spanning tree construction, etc., require that the nodes are provided with identifiers, for breaking symmetry. Indeed, no algorithm can solve these problems in anonymous networks (under a standard distributed scheduler). We show that, for any self-stabilizing algorithm in a network with node identifiers, if the space complexity of the algorithm is too small, then the algorithm does not have more power than a self-stabilizing algorithm running in an anonymous network. More precisely, let  $A$  be an algorithm in a network with node identifiers, and let us assume that  $A$  has space complexity  $o(\log \log n)$  bits per node. Such a small space complexity does not prevent  $A$  from exchanging identifiers between nodes, but they must be transferred as a series of smaller pieces of information that are pipelined along a link, each of size  $o(\log \log n)$  bits. On the other hand, a node cannot store the identifier of even just one of its neighbors. We show that, with spacial complexity  $o(\log \log n)$  bits per node, there exist graphs and assignments of identifiers to the nodes of these graphs such that, in these graphs and for these assignments,  $A$  has the same behavior as an algorithm executed in these graphs but in the absence of identifiers (i.e., in the anonymous version of these graphs). We then show that no algorithms can solve leader election in these graphs in absence of identifiers, from which it follows that  $A$  cannot solve leader election in these graphs with identifiers as long as its space complexity is  $o(\log \log n)$  bits per node.

### 1.3 Related work

Space complexity of self-stabilizing algorithms has been extensively studied for *silent* algorithms, that is, algorithms that guarantee that the content of the variables of every node does not change once the algorithm has reached a legal configuration. For silent algorithms, Dolev and al. [11], proved that finding the centers of a graph, electing a leader, and constructing a spanning tree require registers of  $\Omega(\log n)$  bits per node. Silent algorithms have later been related to a concept known as *proof-labeling scheme* (PLS) [23]. Any lower bound on the size of the proofs in a PLS for a predicate  $\Pi$  on labeled graph implies a lower bound on the size of the registers for silent self-stabilizing algorithms solving  $\Pi$ . A typical example is the  $\Omega(\log^2 n)$ -bit lower bound on the size of any PLS for minimum-weight spanning trees (MST) [22], which implies the same bound for constructing an MST in a silent self-stabilizing manner [4]. Thanks to the tight connection between silent self-stabilizing algorithms and proof-labeling schemes, the space complexity of a vast collection of problems is known, for silent algorithms. (See [14] for more information on proof-labeling schemes.)

On the other hand, to our knowledge, the only lower bound on the space complexity of problems for general self-stabilizing algorithms (without the requirement of being silent) that is corresponding to our setting has been established by Beauquier et al. [3] who proved that registers of constant size are not sufficient for leader election algorithms. Interestingly, the same paper also contains several other space complexity lower bounds for models different from ours – e.g., anonymous networks, or harsher form of asynchrony.

The literature dealing with upper bounds is far richer. In particular, [5] recently presented a self-stabilizing leader election algorithm using registers of  $O(\log \log n)$  bits per node in  $n$ -node rings. This algorithm was later generalized to networks with maximum degree  $\Delta$ , using registers of  $O(\log \log n + \log \Delta)$  bits per node [6]. It is worth to notice that spanning tree construction and  $(\Delta + 1)$ -coloring have the same space complexity  $O(\log \log n + \log \Delta)$  bits per node [6]. Prior to these work, the best upper bound was a space complexity  $O(\log n)$  bits per node [5], and it has then been conjectured that, by some iteration of the technique enabling to reduce the space complexity from  $O(\log n)$  bits per node to  $O(\log \log n)$  bits per node, one could go all the way down to a space complexity of  $O(\log^* n)$  bits per node. Arguments in favor of this conjecture were that such successive exponential improvements have been observed several times in distributed computing. A prominent example is the time complexity of minimum spanning tree construction in the congested clique model [24, 16, 15, 21]. Complexities  $O(\log^* n)$  do exist in the self-stabilizing framework [2], and it seemed at first that the technique in [11] could indeed be iterated (in a similar fashion as in [7]). Our results shows that this is not the case, and that  $\Theta(\log \log n)$  is the right answer.

## 2 Model and definitions

In this paper, we are considering the *state model* for self-stabilization [9]. The asynchronous network is modeled as a simple  $n$ -nodes graph  $G = (V, E)$ , where the set of the nodes  $V$  represents the processes, and the set of edges  $E$  represents pairs of processes that can communicate directly with each other. Such pairs of processes are called *neighbors*. The set of the neighbors of node  $v$  is denoted  $N(v)$ . Each node has local variables and a local algorithm. The variables of a node are stored in its mutable memory, also called register. In the state model, each node  $v$  has read/write access to its register. Moreover, in one atomic step, every node reads its own register and the registers of its neighbors, executes its local algorithm and updates its own register if necessary. Note that the values of the variables of one node  $v \in V$  are called the *state* of  $v$ , and denoted by  $S(v)$ .

Each node  $v \in V$  has a distinct identity, denoted by  $ID(v) \in \{1, \dots, n^c\}$  for some constant  $c > 1$ . For each adjacent edge, each node has access to a locally unique port number. No assumption is made on the consistency between port numbers on each node. The mutable memory is the memory used to store the variables, while the immutable memory is used to store the identifier, the port numbers, and the code of the protocol. As a consequence, the identity and the port numbers are non corruptible constants, and only the mutable memory is considered when computing the memory complexity because it corresponds to the memory readable by the neighbors of the nodes, and thus correspond to the information transmitted during the computation. More precisely, an algorithm may refer to the identity, or to the port numbers, of the node, without the need to store them in the variables. If at least one rule of an algorithm refers to the identity of the node, we call this algorithm an *ID-based algorithm*. Otherwise, if the rules do not refer to the identity of the node we say it is an *anonymous algorithm*.

The output of the algorithm for a problem is carried through local variables of each node. The output of the problem may use all the local variables, or only a subset of them. Indeed, the algorithm must have local variables that match the output of the problem, we call these variables the specification variables. But the algorithm may also need some extra local variables, that may be necessary to compute the specification variables. For example, if we consider a silent BFS spanning tree construction, the specification variables are the variables

dedicated to pointing out the parent in the BFS. However, to respect the silent property, the algorithm needs in each node a variable dedicated to the identity of the root of the spanning tree and a variable dedicated to the distance from the root. As a consequence, we define the *specification* of problem  $P$  as a description of the correct assignments of specification variables, for this specific problem  $P$ .

A *configuration* is an assignment of values to all variables in the system, let us denote by  $\Gamma$  the set of all the configurations. A *legal configuration* is a configuration  $\gamma$  in  $\Gamma$  that respects the specification of the problem, we denote by  $\Gamma^*$  the set of legal configurations. A local algorithm is a set of rules the node can apply, each rule is of the form  $\langle label \rangle : \langle guard \rangle \rightarrow \langle command \rangle$ . A *guard* is a Boolean predicate that uses the local variables of the node and of its neighbors, and a *command* is an assignment of variables. A node is said to be *enabled* if one of its guard is true and *disabled* otherwise.

We consider an asynchronous network, the asynchrony of the system is modeled by an adversary called *scheduler* or *daemon*. The scheduler chooses, at each step, which enabled nodes will execute a rule. Several schedulers are proposed on the literature depending on their characteristics. Dubois and al. in [12] presented a complete overview of these schedulers. Since we are interested in showing a lower bound, we aim for the least challenging scheduler. Our lower bound is established under the *synchronous distributed scheduler*, a strongly fair distributed scheduler. The synchronous distributed scheduler activates, at each step, all the enabled nodes. The synchronous distributed scheduler is captured by the weakly fair and unfair distributed schedulers, but not by the central scheduler that activates only one at each step.

A configuration  $\gamma \in \Gamma$  is a legal configuration for the leader election problem if one single node is elected. More formally:

► **Definition 1** (Leader election). *Leader election in  $G = (V, E)$  is specified by a boolean variable  $\ell_v$  at each node  $v \in V$ . A configuration  $\{(v, \ell_v) : v \in V\}$  is legal if there is a node  $v \in V$  such that  $\ell_v = \text{true}$ , and for every other node  $u \in V \setminus \{v\}$ ,  $\ell_u = \text{false}$ .*

### 3 Formal Statement of the Results

#### 3.1 Lower bounds

Our first result is an  $\Omega(\log \log n)$  lower bound for leader election on the cycle.

► **Theorem 2.** *Let  $c > 1$ . Every deterministic self-stabilizing algorithm solving leader election in the state model under a strongly fair distributed scheduler requires registers on  $\Omega(\log \log n)$  bits per node in  $n$ -node graphs with unique identifiers in  $[1, n^c]$ .*

This bound improves the only lower bound known so far [3], from  $\Omega(1)$  to  $\Omega(\log \log n)$ , and it is tight, as it matches the upper bound of [5]. In particular, it invalidates the folklore conjecture stating that the aforementioned problems are solvable using only  $O(\log^* n)$  memory.

#### Optimality of the assumptions

Our lower bound is actually optimal not only in term of size, but also in terms of the assumptions we make on the setting. More precisely, our theorem has three restrictions: it works for deterministic algorithms only, with the distributed scheduler, and with identifiers in a large enough range. We will now discuss why these limitations are actually necessary.

Randomization is a common tool for symmetry breaking, and our problem is one example. Namely, [18] proved that using randomization, one can solve leader election using constant memory, which implies that our result cannot be generalized in that direction.

An important aspect of the self-stabilizing setting is the scheduler, which is the adversary that decides which nodes can take a step at each round. Different schedulers model different assumptions on the asynchrony of the setting. For example, a fully adversarial scheduler can take any decision, as long as at least one node can take a step at each round. Weaker schedulers can delay arbitrarily the step of a node but are forced to eventually activate any node, etc. Our lower bound is valid under a very weak scheduler, which means that we need a weak assumption on the asynchrony, which in turn means that our result is strong on this aspect. Precisely, what we need in our proof is that it is possible for the scheduler to activate all the nodes at every round. One type of scheduler for which this property does not hold is the so-called centralized scheduler, that activates exactly one node at every round. In this context the symmetry is broken by the scheduler itself (if two nodes are in the same situation, one will be activated first, and this breaks the symmetry). Although the proof of Theorem 2 in Section 4 does not apply to the centralized scheduler, the more general Theorem 3 will allow us to extend our result to the central daemon.

Finally, and this is probably more surprising, we need to consider identities between 1 and  $n^c$ , for  $c > 1$ . In particular, our technique does not work if the identifiers are in  $O(n)$ . This is not an artifact of our proof: it is actually necessary for the result to hold. Indeed, if the identifier range is  $[1, n]$ , then an algorithm may use the node with identifier 1 as a designated node, and have a special code for it. Algorithms using such designated nodes are called *semi-uniform algorithms* and they can achieve space complexity below our lower bound [8, 20]. Even without the possibility of having a designated node, one can take advantage of smaller identifier range: there actually exists an algorithm in constant memory if the identities are in  $[1, n + k]$  for a constant  $k$  [3].

### A general result on the power of the identifiers

Actually, our technique goes beyond the setting of Theorem 2. First, we do not need the harshest aspects of the self-stabilizing model which is that the initial configuration can be arbitrary. If we start from an empty configuration, our technique still holds. Second, the technique works for basically any problem that requires minimal symmetry breaking, not just leader election. Third, as hinted above the type of scheduler is not really important, as long as it does not break symmetry. We prove the more general following theorem.

► **Theorem 3.** *Let  $c > 1$ , and let  $\Delta(n) \in o(\log n)$  be a function. If there exists a deterministic self-stabilizing algorithm  $\mathcal{A}$  that solves a problem  $\mathcal{P}$  in the state model and uses registers of size  $o(\frac{\log \log n}{\Delta})$ , then there exists a deterministic self-stabilizing anonymous algorithm  $\mathcal{A}^a$  that solves  $\mathcal{P}$  on every large-enough graph with maximum degree  $\Delta(n)$ .*

What our paper is really about, is the power of identifiers, in a scenario where very little space/communication is used. Our core result is that below  $\Theta(\log \log n)$ , identifiers are useless, in the sense that in the worst-case the performance of an algorithm using these identifiers is the same as the performance of an anonymous algorithm. Remember that the Naor-Stockmeyer order-invariance theorem [25], that states that in the LOCAL model, for local problems, constant-time algorithms that use the exact values of the identifiers are not more powerful than the order-invariant algorithm that only use the relative ordering of the identifiers. In some sense our paper and [25] have the same take-home message, in two different contexts: if you do not have enough resources, you cannot use the (full) power of the identifiers.

Theorem 3 establishes that proving a  $\Omega(\log \log n)$  lower bounds in the ID-based setting boils down to proving that anonymous algorithms cannot solve the problem. This is useful, because indistinguishability arguments are easier to establish for anonymous algorithms than for algorithms using identifiers.

Finally, one aspect that is not explicit in the statement of the theorem but follows from the proof, is that actually this result also holds if the nodes have inputs. In particular, our result applies to the semi-uniform setting where exactly one node has a special input.

### About the port number model

Our general theorem, Theorem 3, holds in the model where a node knows its port-numbers but not the ones of its neighbors. Intuitively, this implies that a node  $u$  cannot specify that some piece of information is intended to the node of port number  $p$ , because that node does not know it has been assigned port-number  $p$ .

In some graphs, the port number assignment can be chosen in such a way that knowing the port-number assignment of the neighbors does not help. For example in the cycle of Theorem 2, we can arrange the port numbers such that every edge is assigned port number 1 by one endpoint, and port number 2 by the other. This allows to generalize our first result to the model where a node knows both port numbers on every adjacent edge.

### Central scheduler

In a celebrated paper [10], Dijkstra established, among other things, that one cannot break symmetry with anonymous algorithms in composite rings (that is, in rings whose size  $n$  is not a prime number). This results holds under a *central* strongly fair scheduler. A central scheduler is somehow the opposite of the distributed scheduler used in Theorem 2: it activates one node at every round instead of activating all of them. Yet, we can generalize Dijkstra's result. Indeed, as highlighted before, the core of our proofs, and the statement of Theorem 3, is about proving that an algorithm with too little memory cannot perform better than an anonymous algorithm, and this does not depend on the scheduler. Therefore, by combining Dijkstra's result and Theorem 3, we directly get the following corollary, that complements Theorem 2.

► **Corollary 4.** *Let  $c > 1$ . Every deterministic self-stabilizing algorithm solving leader election in the state model under a strongly fair central scheduler requires registers on  $\Omega(\log \log n)$  bits per node in  $n$ -node composite rings with unique identifiers in  $[1, n^c]$ .*

Note that assuming that the ring is composite is essential, since [19] builds a constant memory algorithm for leader election on anonymous prime rings, under a central scheduler.

## 3.2 Intuition of the proofs

### Challenge of lower bounds for non-silent algorithms

Almost all lower bounds for self-stabilization are for silent algorithms, that are required to stay in the same configuration once they have stabilized. These lower bounds are then about a static data structure, the stabilized solution. The question boils down to establishing how much memory is needed to locally certify the global correctness of the solution, and this is well-studied [13].

When we do not require that the algorithm should converge to one correct configuration, and stay there, there is no static structure on which we can reason. It is then unclear how we can establish lower bounds. One way is to think about invariants. Consider a property that

we can assume to hold in the initial configuration, and that is preserved by the computation (if it follows some memory requirement hypothesis). If no correct output configuration has this property, then we can never reach a correct output configuration.

In our proof, the property that will be preserved is that every node has the same state. This can clearly be assumed for the original configuration, and we show that basically if the memory is limited then this is preserved at each step. As the specification we use for leader election is that the leader should output 1, and the other nodes should output 0, then it is not possible that all nodes have the same state in a proper output configuration.

### Intuition on a toy problem

Let us now give some intuition about why we can replace ID-based algorithms by anonymous ones. The code of an ID-based algorithm  $\mathcal{A}$  may refer to the identifier of the node that is running it. For example, a rule of the algorithm could be:

- if the states of the current node and of its left and right neighbors are respectively  $x$ ,  $y$ , and  $z$ , then: if the identifier is odd the new state is  $a$ , otherwise it is  $b$ .

Now suppose you have fixed an identifier, and you look at the rules for this fixed identifier. In our example, if the identifier is 7, the rule becomes:

- if the states of the current node and of its left and right neighbors are respectively  $x$ ,  $y$ , and  $z$ , then: the new state is  $a$ .

This transformation can be done for any rule, thus, for an identifier  $i$ , we can get an algorithm  $\mathcal{A}_i$  specific to this identifier. When we run  $\mathcal{A}$  on every node, we can consider that every node, with some identifier  $i$  is running  $\mathcal{A}_i$ . Note that  $\mathcal{A}_i$  does not refer in its code to the identifier.

The key observation is the following. If the amount of memory an algorithm can use is very limited, then there is very limited number of different behaviors a node can have, especially if the code does not refer to the identifier. Let us illustrate this point by studying an extreme example: a ring on which states have only one bit. In this case the number of input configurations for a node, is the set of views  $(x, y, z)$  as above, with  $x, y, z \in \{0, 1\}$ . That is there are  $2^3 = 8$  different inputs, thus the algorithm can be described with 8 different rules. Since the output of the function is the new state, the output is also a single bit. Therefore, there are at most  $2^8 = 256$  different sets of rules, that is 256 different possible behaviors for a node. In other words, in this extreme case, each specific algorithm  $\mathcal{A}_i$  is equal to one of the behaviors of this list of 256 elements. This implies that, if we take a ring with 257 nodes, there exist two nodes with two distinct identifiers  $i$  and  $j$ , such that the specific algorithms  $\mathcal{A}_i$  and  $\mathcal{A}_j$  are equal.

This toy example is not strong enough for our purpose, as we want to argue about instances where all the nodes run the same code, and as we want non-constant memory. But the idea above can be strengthened to get our theorem. The key is to use the hypothesis that the identifiers are taken from a polynomially large range. As we have a pretty large palette of identifiers, we can always find, not only 2, but  $n$  distinct identifiers in  $[1, n^c]$ , such that all the specific algorithms  $\mathcal{A}_i$  correspond to the exact same behavior. In this case it is as if the algorithm were anonymous.

Note that the larger the memory is, the more different behaviors there are, and the smaller the set of identical specific algorithms we can find. This trade-off implies that for polynomial range, the construction works as long as the memory is in  $o(\log \log n)$ .



#### 4 Proof of Theorem 2

Consider a ring of size  $n$ , and an ID-based algorithm  $\mathcal{A}$  using  $f(n)$  bits of memory per node to solve leader election. An algorithm can be seen as the function that describes the behavior of the algorithm. This function takes an identifier, a state for the node, a state for its left neighbor and a state for its right neighbor, and gives the new state of the node. Formally:

$$\mathcal{A} : \begin{array}{ccccccc} [n^c] & \times & \{0,1\}^{f(n)} & \times & \{0,1\}^{f(n)} & \times & \{0,1\}^{f(n)} & \rightarrow & \{0,1\}^{f(n)} \\ (ID & , & \text{state} & , & \text{left-state} & , & \text{right-state}) & \mapsto & \text{new-state} \end{array}$$

Note that in general, we consider non-directed rings thus the nodes do not have a global consistent definition for right and left. As we are dealing with a lower bound with a worst-case on the port numbering, assuming such a consistent orientation only makes the result stronger. Now we can consider that for every identifier  $i$ , we have an algorithm of the form:

$$\mathcal{A}_i : \begin{array}{ccccccc} \{0,1\}^{f(n)} & \times & \{0,1\}^{f(n)} & \times & \{0,1\}^{f(n)} & \rightarrow & \{0,1\}^{f(n)} \\ (\text{state} & , & \text{left-state} & , & \text{right-state}) & \mapsto & \text{new-state} \end{array}$$

Thus a specific algorithm  $\mathcal{A}_i$  boils down to a function of the form:  $\{0,1\}^{3f(n)} \rightarrow \{0,1\}^{f(n)}$ . Let us call such a function a *behaviour*, and let  $\mathcal{B}_n$  be the sets of all behaviours.

► **Lemma 5.**  $|\mathcal{B}_n| = 2^{f(n) \times 2^{3f(n)}}$

**Proof.** The inputs are basically binary strings of length  $3f(n)$ , thus there are  $2^{3f(n)}$  possibilities for them. Similarly the number of possible outputs is  $2^{f(n)}$ . Thus the number of functions in  $|\mathcal{B}_n|$  is  $(2^{f(n)})^{2^{3f(n)}} = 2^{f(n) \times 2^{3f(n)}}$ . ◀

Lemma 5 implies that the smaller  $f$ , the fewer different behaviors. Let us make this more concrete with Lemma 6.

► **Lemma 6.** *If  $f(n) \in o(\log \log n)$ , then for every  $n$  large enough,  $n^{c-1} > |\mathcal{B}_n|$ .*

**Proof.** Consider the expression of  $n^{c-1}$  and  $|\mathcal{B}_n|$  after applying the logarithm twice:

$$\begin{aligned} \log \log(n^{c-1}) &= \log(c-1) + \log \log n \\ &\sim \log \log n \end{aligned}$$

$$\begin{aligned} \log \log(|\mathcal{B}_n|) &= \log \log \left( 2^{f(n) \times 2^{3f(n)}} \right) = \log \left( f(n) \times 2^{3f(n)} \right) = \log(f(n)) + 3f(n) \\ &\sim 3f(n) \end{aligned}$$

As the dominating term in the second expression is of order  $f(n) \in o(\log \log n)$ , asymptotically the first expression is larger. As  $\log \log(\cdot)$  is an increasing positive function for large values, this implies that asymptotically  $n^{c-1} > |\mathcal{B}_n|$ . ◀

The next lemma shows that if  $f(n) \in o(\log \log n)$ , we can find a large number of identifiers that have the same specific algorithm.

► **Lemma 7.** *If  $n^{c-1} > |\mathcal{B}_n|$ , then there exist a behavior function  $b$ , and a set  $S$  of  $n$  different identifiers, such that:  $\forall i \in S, \mathcal{A}_i = b$ .*

**Proof.** Let the function  $\varphi : [n^c] \rightarrow \mathcal{B}_n$  be the function that associates each identifier to its corresponding behavior function. Let  $S$  be the function that associates to each behavior  $b$ , the set of identifiers  $i$  such that  $\varphi(i) = b$ . Since  $\varphi$  is a function from  $[n^c]$  to  $\mathcal{B}_n$ , its inverse function  $S$  satisfies:  $\cup_{b \in \mathcal{B}_n} S(b) = [n^c]$ . Thereby, the average size of a set  $S(b)$  is

## 24:10 Optimal Space Lower Bound for Self-Stabilizing Algorithms

$$\frac{1}{|\mathcal{B}_n|} \sum_{b \in \mathcal{B}_n} |S(b)| = \frac{1}{|\mathcal{B}_n|} \cdot n^c.$$

Because of Lemma 6, this quantity is strictly larger than  $n$ . Thus, the average of  $|S(b)|$  among all  $b$  is larger than  $n$ , and there must exist at least one behavior  $b$  such that  $|S(b)| > n$ . We take this  $b$  and  $S = S(b)$  for the lemma. ◀

Combining the three lemmas we get that, if  $f(n) \in o(\log \log n)$ , then for any large enough  $n$  we can find  $n$  different identifiers in  $[1, n^c]$ , such that the nodes have the exact same behavior.

Now, consider a large enough graph, with identifiers taken from the set  $S$ . As we are in a self-stabilizing scenario, we can choose from which configuration we start. We actually do not need intricate configuration: we start from a configuration where all states are the same, say some string  $x$ . This cannot be a proper leader election output, because in leader election (with our specification) a leader and a non leader have different output. Therefore, at least one node is activable. Note that every node sees the same states  $x$  for itself, and its two neighbors, and that by construction they have the same behavior. Therefore if one node is activated, all nodes are activated. Now, the scheduler decides to activate all the nodes. Necessarily, all the nodes take the exact same step, and end up in a configuration where every node has the same state.

We can iterate this argument forever. In other other words, the network cannot escape the set of configurations where all nodes have the same state (as long as the scheduler activates all the nodes together, which is allowed). Therefore, the network will never reach a proper leader election configuration, and this proves Theorem 2.

### 5 Proof of Theorem 3

In this section, we prove our general result, that relates ID-based algorithms with small memory to anonymous algorithms. The proof of Theorem 3 follows from the same idea as the one of Theorem 2, but in a more general way. (The constants of the statement have not been optimized.) Consider an algorithm  $\mathcal{A}$  that solves a problem  $\mathcal{P}$  in the state model and uses registers of size  $o(\frac{\log \log n}{\Delta})$  (where the  $\Delta$  is the maximum degree of the graphs treated by  $\mathcal{A}$ , and is bounded as a function of  $n$  by  $\Delta(n) \in o(\log n)$ ).

As before an algorithm can be seen as a function that maps the view of the node to an output. But now, since the degree is not the same for every node, we would have to consider a different function for each degree  $d \in [\Delta]$ . This not very convenient for us, so let us take another point of view. We take a unique function, that takes as input: an identifier, a state, an integer  $\delta$  that represents the degree degree of the vertex at hand, and  $\Delta$  states.

$$\begin{array}{ccccccc} \mathcal{A}: & [n^c] & \times & \{0,1\}^{f(n)} & \times & [\Delta] & \times & (\{0,1\}^{f(n)})^\Delta & \rightarrow & \{0,1\}^{f(n)} \\ & (ID & , & \text{state} & , & \text{degree} & , & \Delta \text{ states}) & \mapsto & \text{new-state} \end{array}$$

To compute the output of a node with degree  $\delta < \Delta$ , the function simply ignores the last  $(\Delta - \delta)$  inputs.

Note that this corresponds to the single-port setting: in the function, each neighboring state is identified. In particular, it is not a set of neighboring states. But it is not a double-port setting: which port a neighbor assigns to a node is unknown.

Now if we fix the identifier  $i$ , we get:

$$\mathcal{A}_i : \begin{array}{l} \{0, 1\}^{f(n)} \times [\Delta] \times (\{0, 1\}^{f(n)})^\Delta \rightarrow \{0, 1\}^{f(n)} \\ \text{(state, degree, } \Delta \text{ states)} \mapsto \text{new-state} \end{array}$$

Now the equivalent of Lemma 5 is that the number of such behaviors  $\mathcal{A}_i$  is:

$$|\mathcal{B}_n| \leq \left(2^{f(n)}\right)^{2^{(\Delta+1)f(n)}\Delta}$$

We bound the double logarithm of this expression:

$$\begin{aligned} \log \log |\mathcal{B}_n| &\leq \log \log \left[ \left(2^{f(n)}\right)^{2^{(\Delta+1)f(n)}\Delta} \right] \\ &\leq \log \left[ f(n) 2^{(\Delta+1)f(n)} \Delta \right] \\ &\leq (\Delta + 1)f(n) + \log f(n) + \log \Delta(n) \\ &\in o(\log \log n) \end{aligned}$$

We get that for any large enough  $n$ ,  $n^{c-1} > |\mathcal{B}_n|$ , and by Lemma 7, we directly get that there exists an identifier assignment such that all nodes have the same behavior. In other words, on this identifier assignment, the algorithm behaves like anonymous. As a consequence any impossibility result for anonymous algorithms also holds here. This completes the proof of Theorem 3.

## 6 Conclusion

In this paper, we have established a lower bound  $\Omega(\log \log n)$  bits per node on the size of the registers for self-stabilizing algorithms solving leader election in the state model. This bound matches the upper bound  $O(\log \log n)$  bits per node for bounded-degree graphs [6]. The same upper bound on the size of the registers is known to hold in bounded-degree graphs for vertex coloring and spanning tree construction [6]. An interesting open problem is to determine the space complexity of vertex coloring and spanning tree construction.

---

### References

- 1 Jordan Adamek, Mikhail Nesterenko, and Sébastien Tixeuil. Evaluating practical tolerance properties of stabilizing programs through simulation: The case of propagation of information with feedback. In *Stabilization, Safety, and Security of Distributed Systems - 14th International Symposium, SSS 2012*, pages 126–132, 2012. doi:10.1007/978-3-642-33536-5\_13.
- 2 Baruch Awerbuch and Rafail Ostrovsky. Memory-efficient and self-stabilizing network RESET (extended abstract). In *13th Annual ACM Symposium on Principles of Distributed Computing, PODC 1994*, pages 254–263, 1994. doi:10.1145/197917.198104.
- 3 Joffroy Beauquier, Maria Gradinariu, and Colette Johnen. Memory space requirements for self-stabilizing leader election protocols. In *18th Annual ACM Symposium on Principles of Distributed Computing, PODC 1999*, pages 199–207, 1999. doi:10.1145/301308.301358.
- 4 Lélia Blin and Pierre Frgaignaud. Space-optimal time-efficient silent self-stabilizing constructions of constrained spanning trees. In *35th IEEE International Conference on Distributed Computing Systems, ICDCS 2015*, pages 589–598, 2015. doi:10.1109/ICDCS.2015.66.
- 5 Lélia Blin and Sébastien Tixeuil. Compact deterministic self-stabilizing leader election on a ring: the exponential advantage of being talkative. *Distributed Computing*, 31(2):139–166, 2018. doi:10.1007/s00446-017-0294-2.

- 6 Lélia Blin and Sébastien Tixeuil. Compact self-stabilizing leader election for general networks. In *LATIN 2018: Theoretical Informatics - 13th Latin American Symposium*, pages 161–173, 2018. doi:10.1007/978-3-319-77404-6\_13.
- 7 Lucas Boczkowski, Amos Korman, and Emanuele Natale. Minimizing message size in stochastic communication patterns: Fast self-stabilizing protocols with 3 bits. In *28th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017*, pages 2540–2559, 2017. doi:10.1137/1.9781611974782.168.
- 8 Ajoy Kumar Datta, Colette Johnen, Franck Petit, and Vincent Villain. Self-stabilizing depth-first token circulation in arbitrary rooted networks. *Distributed Computing*, 13(4):207–218, 2000. doi:10.1007/PL00008919.
- 9 Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, 1974. doi:10.1145/361179.361202.
- 10 Edsger W. Dijkstra. *Self-Stabilization in Spite of Distributed Control*, pages 41–46. Springer New York, New York, NY, 1982. doi:10.1007/978-1-4612-5695-3\_7.
- 11 Shlomi Dolev, Mohamed G. Gouda, and Marco Schneider. Memory requirements for silent stabilization. *Acta Inf.*, 36(6):447–462, 1999. doi:10.1007/s002360050180.
- 12 Swan Dubois and Sébastien Tixeuil. A taxonomy of daemons in self-stabilization, 2011. arXiv:1110.0334.
- 13 Laurent Feuilloley. Introduction to local certification. *CoRR*, abs/1910.12747, 2019. arXiv:1910.12747.
- 14 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>, arXiv:1606.04434.
- 15 Mohsen Ghaffari and Merav Parter. MST in log-star rounds of congested clique. In *2016 ACM Symposium on Principles of Distributed Computing, PODC 2016*, pages 19–28, 2016. doi:10.1145/2933057.2933103.
- 16 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 *2015 ACM Symposium on Principles of Distributed Computing, PODC 2015*, pages 91–100, 2015. doi:10.1145/2767386.2767434.
- 17 Ted Herman and Sriram V. Pemmaraju. Error-detecting codes and fault-containing self-stabilization. *Inf. Process. Lett.*, 73(1-2):41–46, 2000. doi:10.1016/S0020-0190(99)00164-7.
- 18 Gene Itkis and Leonid A. Levin. Fast and lean self-stabilizing asynchronous protocols. In *35th Annual Symposium on Foundations of Computer Science FOCS 1994*, pages 226–239, 1994. doi:10.1109/SFCS.1994.365691.
- 19 Gene Itkis, Chengdian Lin, and Janos Simon. Deterministic, constant space, self-stabilizing leader election on uniform rings. In *Distributed Algorithms, 9th International Workshop, WDAG '95*, pages 288–302, 1995. doi:10.1007/BFb0022154.
- 20 Colette Johnen. Memory efficient, self-stabilizing algorithm to construct BFS spanning trees. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing, PODC 97*, page 288, 1997. doi:10.1145/259380.259508.
- 21 Tomasz Jurdzinski and Krzysztof Nowicki. MST in  $O(1)$  rounds of congested clique. In *29th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018*, pages 2620–2632, 2018. doi:10.1137/1.9781611975031.167.
- 22 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.
- 23 Amos Korman, Shay Kutten, and David Peleg. Proof labeling schemes. *Distributed Computing*, 22(4):215–233, 2010. doi:10.1007/s00446-010-0095-3.
- 24 Zvi Lotker, Boaz Patt-Shamir, Elan Pavlov, and David Peleg. Minimum-weight spanning tree construction in  $O(\log \log n)$  communication rounds. *SIAM J. Comput.*, 35(1):120–131, 2005. doi:10.1137/S0097539704441848.
- 25 Moni Naor and Larry J. Stockmeyer. What can be computed locally? *SIAM J. Comput.*, 24(6):1259–1277, 1995. doi:10.1137/S0097539793254571.

# Accountability and Reconfiguration: Self-Healing Lattice Agreement

Luciano Freitas de Souza ✉

LTCI, Télécom Paris, Institut Polytechnique de Paris, France

Petr Kuznetsov ✉

LTCI, Télécom Paris, Institut Polytechnique de Paris, France

Thibault Rieutord ✉

CEA-List, Université Paris-Saclay, Palaiseau, France

Sara Tucci-Piergiovanni ✉ 

CEA-List, Université Paris-Saclay, Palaiseau, France

---

## Abstract

---

An *accountable* distributed system provides means to detect deviations of system components from their expected behavior. It is natural to complement fault detection with a reconfiguration mechanism, so that the system could heal itself, by replacing malfunctioning parts with new ones. In this paper, we describe a framework that can be used to implement a large class of accountable and reconfigurable replicated services. We build atop the fundamental lattice agreement abstraction lying at the core of storage systems and cryptocurrencies.

Our asynchronous implementation of accountable lattice agreement ensures that every violation of consistency is followed by an undeniable evidence of misbehavior of a faulty replica. The system can then be seamlessly reconfigured by evicting faulty replicas, adding new ones and merging inconsistent states. We believe that this paper opens a direction towards asynchronous “self-healing” systems that combine accountability and reconfiguration.

**2012 ACM Subject Classification** Theory of computation → Distributed algorithms

**Keywords and phrases** Reconfiguration, accountability, asynchronous, lattice agreement

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2021.25

**Funding** Luciano Freitas de Souza was supported by Nomadic Labs, and Petr Kuznetsov by TrustShare Innovation Chair.

## 1 Introduction

There are two major ways to deal with failures in distributed computing:

**Fault-tolerance:** we anticipate failures by investing into replication and synchronization, so that the system’s correctness is not affected by faulty components.

**Accountability:** we detect failures *a posteriori* and raise undeniable evidences against faulty components.

Accountability in computing has been proposed for generic distributed systems [18, 19] as a mechanism to detect deviations of system nodes from the algorithms they are assigned with. It has been shown that a large class of deviations of a given process from a given deterministic algorithm can be detected by maintaining a set of *witnesses* that keep track of all *observable* actions of the process and check them against the algorithm [20].

The generic approach can be, however, very expensive in practice and one may look for a more tractable, *application-specific* accountability mechanism. Indeed, instead of pursuing the ambitious goal of detecting deviations from the assigned algorithm, we might want to only care about deviations that violate the specification of the problem the algorithm is trying to solve.



© Luciano Freitas de Souza, Petr Kuznetsov, Thibault Rieutord, and Sara Tucci-Piergiovanni; licensed under Creative Commons License CC-BY 4.0

25th International Conference on Principles of Distributed Systems (OPODIS 2021).

Editors: Quentin Bramas, Vincent Gramoli, and Alessia Milani; Article No. 25; pp. 25:1–25:23

Leibniz International Proceedings in Informatics



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

The idea has been successfully employed in the context of Byzantine Consensus [11]. The accountable version of consensus guarantees correctness as long as the number of faulty processes does not exceed some fixed  $f$ . But if correctness is violated, e.g., honest processes take different decisions, then at least  $f + 1$  Byzantine processes are presented with undeniable evidences of misbehavior. This is not surprising: a decision in a typical  $f$ -resilient consensus protocol must receive *acknowledgements* from a *quorum* of processes, and any two quorums must have at least  $f + 1$  processes in common [31]. The fact that two processes took different decisions implies that at least  $f + 1$  processes in the intersection of the corresponding quorums *equivocated*, i.e., acknowledged conflicting decision values. Assuming that every decision is provided with a cryptographic *certificate* containing the set of signed acknowledgements from a quorum of processes, we can immediately construct a desired evidence. Polygraph [11], a recent accountable Byzantine Consensus protocol, naturally builds upon the classical PBFT protocol [9]. One may ask – okay, we have detected a faulty process, but what should we do next? Ideally, we would like to *reconfigure* the system by evicting the faulty process and *reinitializing* the system state.

*Reconfigurable replicated systems* [15, 16, 21, 35] allow the users to dynamically update the set of replicas. It has been recently shown that reconfiguration can be implemented in purely *asynchronous* environments [1, 2, 15, 21, 23, 35]. The idea was first applied to (read-write) storage systems [1, 2, 15], and then extended to max-registers [21, 35] and more general *lattice* data types, first in the crash-fault context [23] and then for Byzantine failures [24].

**Contribution.** In this paper, we propose a framework that can be used to implement a large class of replicated services that are both accountable and reconfigurable. Following recent work on reconfiguration [21, 23, 24], we build atop the fundamental *lattice agreement* abstraction. Lattice agreement [4, 14] (LA) takes arbitrary inputs in a *lattice* (a partially ordered set equipped with a *join* operator) and returns outputs that are (1) joins of the inputs, and (2) ordered with respect to the lattice partial order. The LA abstraction is weaker than consensus and can be implemented in an asynchronous system.

Lattice agreement appears to be a perfect match for both desired features: accountability and reconfiguration. Indeed, a quorum-based LA implementation enables detection of misbehaving parties: as soon as two correct users learn two incomparable values, they also obtain a proof of misbehavior of all replicas that *signed* both values. Furthermore, the very process of reconfiguration can be represented as agreement defined on a lattice of *configurations* [21, 23]. These two observations inspire the design of our system.

We propose an accountable *and* reconfigurable implementation that reaches agreement on a *joint* lattice: an object lattice (defining the current *state* of the replicated object) and a configuration lattice (defining the current *configuration* of the replicas). Assuming that the number of failures is less than half of the system size, our implementation is *alive*. It is also *safe* if only benign (crash) failures occur. Once safety is violated, i.e., two correct users learn two incomparable object states, some Byzantine replicas are inevitably confronted with an undeniable proof of misbehavior. The system is then seamlessly reconfigured by evicting the detected replicas, adding new ones and merging inconsistent states. Once the state is merged, the system comes back to providing safety and liveness, as long as no new replicas exhibits Byzantine behavior. Eventually all Byzantine replicas are detected and the system comes back to maintaining both liveness and safety.

**Outdated configurations are harmless.** Our system prevents users from accessing outdated configurations with the use of *forward-secure digital signature scheme* [5, 13]. A member of each new configuration is assigned a new secret key. Furthermore, honest members of an old

configuration are expected to destroy their old keys before moving to a new one. Thus, if they are later compromised, they will not be able to serve clients' requests, and the remaining Byzantine replicas will not constitute a quorum.

**On Byzantine clients.** For simplicity, our solution assumes that service replicas are subject to Byzantine failures, but clients are *benign*: they can only fail by crashing. This assumption has already been made in designs of fault-tolerant storage systems [29]. In our case, it precludes the cases when a Byzantine client brings the system into a compromised configuration or slows down the system by issuing excessive reconfiguration requests. In Appendix A we also describe a *one-shot* version of accountable lattice agreement, without reconfiguration, in which both clients and replicas can be Byzantine. Marrying reconfiguration and accountability in a *long-lived* service that can be accessed by Byzantine clients remains an important challenge. One way to address it is to assume an external *access control* mechanism [36] ensuring that only “authentic” configurations are accepted as inputs to the reconfiguration procedure. We discuss this issue in more detail in Section 6.

**Summary.** Altogether, we believe that this paper opens a new area of asynchronous “self-healing” systems that combine accountability and reconfiguration. Such a system either preserves safety and liveness or preserves liveness and compensates safety violations with eventual detection of Byzantine replicas. It also exports a reconfiguration interface that allows the clients to replace compromised replicas with new, correct ones. In this paper, we show that both mechanisms, accountability and reconfiguration, can be implemented in a purely asynchronous (in the modern parlance – *responsive*) way.

**Road map.** The rest of the paper is organized as follows. In Section 2, we introduce our system model. In Section 3, we state the problem of reconfigurable and accountable lattice agreement (RALA) and in Section 4.1, we describe our RALA implementation analysing its correctness. In Section 5, we discuss related work, and in Section 6 we present an overview of possible improvements and interesting open questions. In Appendix A, we present our one-shot accountable lattice agreement (A1LA) that assumes that both clients and replicas can be Byzantine and analyse its correctness.

## 2 System Model

We assume that the system is asynchronous and that it is composed by a set  $\Pi$  of processes that communicate over reliable message-passing channels exchanging authenticated messages. These processes are split into a set  $\Sigma$  of *replicas* that maintain a *replicated service* and a set  $\Gamma$  of *clients* that use the service. We assume the existence of a global clock with range  $\mathbb{N}$ , but the processes do not have access to it.

In each run, a process can be: (1) *correct* ( $C$ ) if it faithfully follows the algorithm it is assigned with, (2) *benign* ( $B$ ) if it can only deviate from the algorithm by prematurely stopping taking steps of its algorithm, or (3) *malicious* ( $M$ ) or Byzantine if it skips steps or takes a step not prescribed by its algorithm.

We assume a *forward-secure digital signature scheme* [5, 6, 13, 28]. In the scheme, the public key of a process  $p$  is fixed while its secret key  $sk_t^p$  evolves with its *timestamp*  $t$ , a natural number bounded by a fixed natural parameter  $T$ , usually taken sufficiently large (e.g.,  $2^{64}$ ), to accommodate any possible system lifetime. For any  $t'$ ,  $t < t' \leq T$ , the process can *update its secret key* and obtain  $sk_{t'}^p$  from  $sk_t^p$ . However, we assume that it is computationally

infeasible to “downgrade” the key to a lower timestamp, from  $sk_t^p$  to  $sk_{t'}^p$ . In particular, once a process updates its timestamp from  $t$  to  $t' > t$ , and then destroys  $sk_t^p$ , it is no longer able to sign messages with timestamp less than  $t'$ , even if it turns Byzantine later.

More formally, we model a forward-secure signature scheme as an oracle which associates every process  $p$  with a timestamp  $t_p$ . The oracle provides process  $p$  with three operations: (1)  $UpdateFSKeys(t)$  sets  $t_p$  to  $t$  if  $t$  is greater than the current value of  $t_p$  but less or equal to  $T$  (2)  $FSSign(m, t)$  returns a signature  $s$  for message  $m$  and timestamp  $t$ , assuming  $t \geq t_p$ ; (3)  $FSVerify(m, t, s, q)$  returns *true* iff the message  $m$  provides a signature  $s$  generated by a valid call  $FSSign(m, t)$  by process  $q$ .

We also make use of a weak broadcast primitive that ensures that once a correct process broadcasts a message, all correct processes eventually receive it, e.g., via a gossip mechanism. Notice that, unlike reliable broadcast [7, 8], we only require the primitive to disseminate messages broadcast by correct processes, not to make them eventually agree on the set of delivered ones.

We assume that all clients are benign. For the sake of simplicity, we assume that once a correct process learns an output, it eventually proposes a new input, and that there are only finitely many correct clients.<sup>1</sup>

### 3 Reconfigurable and Accountable Lattice Agreement: Specification

A lattice is a partially ordered set where any pair of elements has a unique *join*, or supremum, and a unique *meet*, or infimum. We denote  $\mathcal{O}$  the object lattice corresponding to the data type the user wishes to implement using the system (such as a counter, a set or commit-abort) and  $K$  the configuration lattice.

A *configuration*  $\kappa$  is a finite set of pairs  $(\sigma, inout) | \sigma \in \Sigma, inout \in \{+, -\}$ . Intuitively,  $(\sigma, +) \in \kappa$  means that  $\sigma$  has been earlier added to the configuration and  $(\sigma, -)$  means that  $\sigma$  has been removed from it. We say that a replica  $\sigma$  is a member of  $\kappa$  if  $(\sigma, +) \in \kappa$  and  $(\sigma, -) \notin \kappa$ .

$|\kappa|$  is defined as the cardinality of the set of pairs representing the configuration;  $\kappa.excluded$  returns all the replicas excluded from it;  $\kappa.included$  that were at some moment included on it;  $\kappa.members := \kappa.included \setminus \kappa.excluded$ . We only consider *well-formed* configurations  $\kappa$ :  $\kappa.excluded \subseteq \kappa.included$  (a replica can be removed only if it has been previously added).

In the *reconfigurable accountable (long-lived) lattice agreement (RALA)* abstraction, defined on a product lattice  $(\mathcal{L}, \sqsubseteq) = (\mathcal{O} \times K, \sqsubseteq^{\mathcal{O}} \times \sqsubseteq^K)$ , a client  $c_i$  periodically proposes inputs  $(\iota, \kappa) | \iota \in \mathcal{O}, \kappa \in K$  to replicas in  $\Sigma$  and obtains, as output, a value  $v \in \mathcal{L}$ .

Additionally, the client locally maintains an *accusation set*  $\alpha_i = (A, P)$  where  $A \subset \Sigma$  is a set of replicas and  $P \in \mathcal{P}$  is a *proof* (here  $\mathcal{P}$  is the set of proofs). The system provides a Boolean map  $verify-proof: (2^{\Sigma} \times \mathcal{P}) \rightarrow \{true, false\}$  that can be used by any process or third party to *verify* a proof. For example, a proof can be a set of messages that, for every replica in  $r \in A$ , contains one or more messages signed by  $r$  that cannot be sent by  $r$  in any execution of our algorithm.

When a client  $c$  receives an input  $v$  from the upper-level application we say that  $c$  *proposes*  $v$ . When  $c$  outputs a value  $v \in \mathcal{L}$ , we say that  $c$  *learns* (or *decides*)  $v$ . When  $c$  sets its accusation set to  $(A, P)$ , we say that  $c$  *accuses*  $A$  with  $P$ .

<sup>1</sup> Our specification can be easily refined to accommodate infinitely many correct clients under the assumption that the number of *concurrently* proposed values is bounded.



Given a client  $c_i$ , let  $I^i = \langle (\iota_0^i, \kappa_0^i), (\iota_1^i, \kappa_1^i), \dots \rangle$  denote the sequence of inputs and  $\Upsilon^i = \langle v_0^i, v_1^i, \dots \rangle$  denote the sequence of outputs. If for some client  $c_i$  and  $k \in \mathbb{N}$ ,  $\kappa_k^i \neq \kappa_{k+1}^i$ , i.e.,  $c_i$  proposes to change the configuration, we say that  $c_i$  *issues a reconfiguration request*.

Now a *RALA* system must ensure the following properties:

- **Validity.** Each value  $v_k^i$ ,  $k \geq 0$ , learned by a client  $c_i$  is a join of the  $k$ -prefix of its input sequence and some values from other client's inputs.
- **Completeness.** If a correct client learns a value that is incomparable with a value learnt by another correct client then it eventually accuses some replicas it had not yet accused before.

$$\forall c_i, c_j \in C \cap \Gamma, \forall k, l \in \mathbb{N}, \neg \left( v_k^i \sqsubseteq v_l^j \vee v_l^j \sqsubseteq v_k^i \right), \text{ where } c_i \text{ learns } v_k^i \text{ at time } t$$

$$\implies \exists t' > t : A^i[t] \not\sqsubseteq A^i[t']$$

- **Accusation Stability.** The accusation sets monotonically increase.

$$\forall c_i \in \Gamma, t, t' \in \mathbb{N}, t < t' : A^i[t] \subseteq A^i[t']$$

- **Accuracy.** If a client accuses a set of replicas  $A$ , then it has a valid proof against each replica in  $A$ :

$$\forall c_i \in \Gamma, \forall t \in \mathbb{N}, \text{verify-proof}(A^i[t], P^i[t])$$

- **Authenticity.** It is computationally infeasible to accuse a benign process, i.e., to construct  $P \in \mathcal{P}$  s.t.  $\text{verify-proof}(A, P) = \text{true}$  and  $A \cap B \neq \emptyset$ .
- **Agreement.** The correct clients eventually agree on the replicas they accuse.

$$\forall t \in \mathbb{N}, \forall c_i, c_j \in \Gamma, \exists t' \in \mathbb{N}, t' > t : A^i[t] \subseteq A^j[t']$$

- **Liveness.** If the system reconfigures only finitely many times, every value proposed by a correct client is eventually included in the value learned by every correct client.

$$\forall c_i \in \Gamma, \forall k \in \mathbb{N}, \forall c_j \in \Gamma, \exists \ell \in \mathbb{N} | \iota_k^i \sqsubseteq v_\ell^j$$

A configuration  $\kappa$  is said to be *active* (at a given moment of time  $t$ ) if (1) it is a join of configurations proposed and learnt by time  $t$ , (2) and no other correct process learns a configuration  $\kappa' | \kappa \sqsubset \kappa'$  by time  $t$ . Liveness guarantees of our algorithm rely upon the following condition:

**Configuration availability:** For all times  $t$ , any configuration that is active at all  $t' > t$  contains a majority of correct processes.

This is a conventional assumption in asynchronous reconfigurable systems [1, 23, 35]. The intuition behind it is the following. If an active configuration remains active forever, i.e., it is never superseded, then it should contain enough correct replicas. On the other hand, a once active but later superseded configuration may contain arbitrarily many Byzantine processes: the clients' requests will be served by the new configuration.

Notice that the properties above imply that either the values learnt by correct processes are comparable or eventually some Byzantine replicas are detected. If from some point on, no more Byzantine faults take place, we ensure that all new learnt values are comparable. Our requirement of finite number of reconfigurations is standard in the corresponding literature [2, 23, 35] and, in fact, can be shown to be necessary [34]. In practice, we ensure liveness in “sufficiently long” time intervals without reconfiguration.

Notice that the choice of new configurations to propose is left entirely to the clients, as long as the condition above is satisfied. In Section 6, we discuss possible reconfiguration strategies the clients may want to choose. However, it is important to emphasize that regardless of this strategy, **the system does not allow the accused replicas to affect the system's safety and liveness anymore.**

## 4 Reconfigurable and Accountable Lattice Agreement: Implementation

### 4.1 Algorithm

Our RALA implementation is given in Algorithm 2, Algorithm 3, and Algorithm 4. We assume that every method in the algorithms is executed by the process *sequentially*, without being interrupted by other methods of this process. Moreover, we consider that the processes *ignore* accused replicas, messages with invalid signatures and messages whose signatures do not match the configuration content.

■ **Algorithm 1** Example of Verify-proof Operation.

---

**operation** *Verify-Proof*(*accusation*(*A*,*P*))

```

1  foreach Process b ∈ A do
2      let MSG be the union of all messages by b in P
3      Check if every m in MSG has a valid signature continue if not
4      Get all ACKs in MSG and check if they are comparable, continue if not
5      Get all Proposal in MSG and check if they obey the description continue if not
6      Get all Decision in MSG and check if their ACKs hold continue if not
7      return false
8  return true

```

---

**Overview.** The clients propose values to the replicas which can either *accept* them by issuing an ACK or *reject* them by issuing a NACK. Once enough responses are gathered by the proposing client, it can accordingly either proceed to learn the value it proposed or to refine its proposal so it contains the missing information replicas raised. If no malicious replica tries to deviate, the values learnt are comparable and no accusations are raised. On the other hand, once a replica induces clients to learn incomparable values it is eventually detected and an accusation against it is produced.

The following definitions and boolean map are used in the algorithm and proofs of correctness:

► **Definition 1** (*S* satisfies configurations). *Let S be a set of replicas, κ a configuration, and D a set of configurations. We say that S satisfies D upon κ iff, for each d ⊆ D, S contains a majority of replicas in each configuration in the set κ ∪ ∪d.*

► **Definition 2** (Pending Configurations). *A configurations is called pending as long as a client has received it but has not yet included it in the most recent decided configuration (lines 38 and 44). This set is comprised of the current client proposal, as well as configurations coming from ACKs (line 13 and 24).*

The map *verify\_maj*:  $(\Sigma \times K \times 2^K) \rightarrow \{true, false\}$  where *verify\_maj*(*S*, *κ*, *D*) = *true* iff *S* satisfies *D* upon *κ*. This map is used to indicate that the client gathered all the responses it needed.

**Ledgers.** Every client maintains a local *ledger*, called *ackL*, reserved to keep track of signed ACK messages the client received and their senders. Also, clients and replicas maintain two more ledgers to register the values introduced in the system by their origin processes called *objL* and *confL*. By indexing a ledger *l* by a process *p* ( $l[p]$ ), one can recover all the values signed by *p* present in *l*.

■ **Algorithm 2** Reconfigurable Accountable Lattice Agreement: Code for client *c* part 1.

---

**Local variables:**

*status*, initially *waiting* { Boolean indicating status: *waiting* or *proposing* }  
*dest*, initially  $\emptyset$  { Set of replicas that must be contacted }  
*nackBool*, initially *false* { Flag indicating whether a NACK has already been received or not }  
*activePropNb*, initially  $-1$  { Index of the current active proposal }  
*activeOutNb*, initially  $0$  { Index of the next value to be learnt }  
*propV*, initially  $\perp$  { Value currently being proposed }  
*objL*, initially empty { Ledger matching object values in the system to their original proposer }  
*confL*, initially containing *Initial Conf* signed by *c* { Analogous to *objL* for configurations }  
*ackL*, initially empty { Ledger matching acks to the replicas that issued them }  
*pendConf*, initially  $\emptyset$  { Set of pending configurations }  
*RESPSet*, initially  $\emptyset$  { Set of replicas that responded }  
*lastDec*, initially  $(\perp, \textit{intialConfig})$  { Last decided value }

**Input:**

*inBuffer* { Values received by the client from an external source to insert in the system }

**Outputs:**

*outV*, initially  $\perp$  { Array of values learnt by the client }  
*accusation*, initially  $\emptyset$  { Set of accusations issued by the client }

**upon** *status* = *waiting* AND *inBuffer*  $\neq \perp$

9    *extract* and *sign* objects from *inBuffer* and *include* them to *objL*  
10    *extract* and *sign* configurations from *inBuffer* and *include* them to *confL*  
11    *Propose*

**operation** *Propose*

12    *propV* := *extractLedger*(*objL*, *confL*, *c*)  
13    *include propV.conf* to *pendConfSet*  
14    *status* := *proposing*  
15    *activePropNb* := *activePropNb* + 1  
16    *clear ackL[activeOutNb]*  
17    *clear RESPSet*  
18    *nackBool* := *false*  
19    *dest* := *propV.conf.included* – *lastDec.conf.excluded*  
20    *multicast*  $\langle \textit{PROPOSAL}, (\textit{objL}, \textit{confL}, \textit{lastDec}, \textit{activePropNb}) \rangle$  to replicas in *dest*

**upon** *verify\_maj*(*RESPSet*, *lastDec.conf*, *pendConf*) = *true*

21    **if** *nackBool* = *true* **then** *Propose* **else** *Decide*

**upon** *receive*  $\langle \textit{ACK}, (\textit{HASH}(\textit{propV}), \textit{lastDec}, \textit{pendConf}', \textit{activePropNb}) \rangle$   
from replica *r* AND *status* = *proposing* AND  $r \notin \textit{ackL}$  AND  $r \in \textit{dest}$

22    **if** *propV.conf*  $\in \textit{pendingConf}'$  **then**

23        *append r's ACK message* to *ackL[activeOutNb]*  
24        *include elements from pendConf'* which aren't subset of *lastDec.conf* in *pendConfSet*  
25        *append r* to *RESPSet*  
26    **else**  $\langle \textit{ACCUSATION}, (\textit{accusation}) \rangle$   
27        *include (r, ACK)* to *accusation*  
28        *broadcast*  $\langle \textit{ACCUSATION}, (\textit{accusation}) \rangle$

---

## 25:8 Accountability and Reconfiguration: Self-Healing Lattice Agreement

**Issuing a proposal.** A client starts in a *waiting* status and listens for values in its *inBuffer* to include them in a new proposal (lines 9 and 10), not taking any values from the buffer while the executing a *proposal*. Additionally, it must also listen for decisions made by other clients (lines 45 and 46) including them in its proposal, preventing malicious replicas from keeping values from it. It then proceeds to multicast its *propV* to the replicas that might satisfy the pending configurations (variable *dest*) it has seen upon the last decided configuration it came by (line 20) and waits for them to respond.

■ **Algorithm 3** Reconfigurable Accountable Lattice Agreement: Code for client *c* part 2.

---

```

upon receive  $\langle \text{NACK}, (\text{HASH}(\text{propV}), \Delta \text{objL}', \Delta \text{confL}', \text{activePropNb}) \rangle$  from replica r
  AND status = proposing AND r  $\in$  dest
29  nackV := extractLedger( $\Delta \text{objL}', \Delta \text{confL}', r$ )
30  if nackV  $\sqsubseteq$  propV return
31  objL := objL  $\cup$  objL'
32  confL := confL  $\cup$  confL'
33  nackBool := true
34  append r to RESPSet

operation Decide
35  outV[activeOutNb] := propV
36  broadcast  $\langle \text{DECISION}, (\text{objL}, \text{confL}, \text{ackL}[\text{activeOutNb}]) \rangle$ 
37  lastDec := outV[activeOutNb]
38  pendConfSet :=  $\emptyset$ 
39  activeOutNb := activeOutNb + 1
40  status := waiting

upon receive  $\langle \text{DECISION}, (\text{objL}', \text{confL}', \text{ackL}') \rangle$  from client c'
41  outV' := extractLeger(objL', confL')
42  lastDecOld := lastDec
43  lastDec := lastDec  $\cup$  outV'
44  Eliminate from pendConf subsets of lastDec.conf
45  objL := objL'  $\cup$  objL
46  confL := confL'  $\cup$  confL
47   $\forall i \mid \text{outV}' \not\sqsubseteq \text{outV}[i] \ \&\& \ \text{outV}[i] \not\sqsubseteq \text{outV}'$ 
48    let M =  $\{m \mid m \in \text{ackL}[i] \ \&\& \ m \in \text{ackL}' \ \&\& \ m \notin \text{accusation}\}$ 
49    foreach m  $\in$  M do include (m,  $\{\text{ackL}[m], \text{ackL}'[m]\}$ ) to accusation
50    if  $|M| > 0$  then broadcast  $\langle \text{ACCUSATION}, (\text{accusation}) \rangle$ 
51  if lastDec.conf  $\not\sqsubseteq$  lastDecOld.conf  $\vee$  outV'  $\not\sqsubseteq$  propV then Propose

operation extractLedger (objL', confL', sender)
52  if  $\exists$  process p  $\in$  objL' or confL' with invalid signature then
53    accusation := accusation  $\cup$   $\{(sender, \text{getMSG}(\text{objL}') \cup \text{getMSG}(\text{confL}'))\}$ 
54    broadcast  $\langle \text{ACCUSATION}, (\text{accusation}) \rangle$ 
55    return  $\emptyset$ 
56  let receivedValue =  $(\sqcup [v \mid \exists p, \text{objL}'[p] = v], \sqcup [c \mid \exists p, \text{confL}'[p] = c])$ 
57  return receivedValue

upon receive  $\langle \text{ACCUSATION}, (\text{accusation}') \rangle$  from client q
58   $\Delta \text{Proof}$  :=  $\emptyset$ 
59  foreach process b accused in accusation' with p and who isn't present in accusation do
60    include (b, p) in  $\Delta \text{Proof}$ 
61  if  $\Delta \text{Proof} \neq \emptyset$  then
62    accusation := accusation  $\cup$   $\Delta \text{Proof}$ 

```

---

**Treating Client Proposals.** The replicas that receive the proposal extract the value from the ledger (line 65). This makes use of the operation *extractLedger* which verifies that all the values came from existing clients, making these values valid. Each replica then checks whether the new proposal contains the join values it has already seen proposed (*repV*), in which case they ack it (line 73) or not, in which case they nack it (line 76), sending a complement to the ledger allowing the client to update its proposal. Benign replicas always forward their keys, destroying the old ones in the process, before responding to clients (line 72).

A replica cannot provide a client with outdated information because the timestamp used in the signature of its messages is only valid if it has been forwarded to the content it proposes and cannot be rolled back. Moreover, if a benign replica sees that a client isn't aware of a decision it has already come by, it will ignore the client proposal until it includes newer information (line 63).

The function *getMessage* (line 53) takes a set of input values and returns the set of proposals or NACK messages that originally contained them.

■ **Algorithm 4** Reconfigurable Accountable Lattice Agreement: Code for replica  $r$ .

**Local variables:**

*objL*, initially empty { Object Ledger }  
*confL*, initially empty { Configuration Ledger }  
*repV* initially  $\perp$  { Value held by replica }  
*pendConf*, initially  $\emptyset$  { Pending Configurations }  
*lastDec*, initially  $\perp$   
signature timestamp  $t_r$  initially  $|Initial\ Configuration|$

**sign** all outgoing messages  $m$  with  $FSSign(m, t_r)$

**upon** receive  $\langle PROPOSAL, (objL', confL', lastDec', activePropNb') \rangle$  from client  $c$

```

63  if  $lastDec' \sqsubseteq lastDec$  then return
64   $lastDec := lastDec \cup lastDec'$ 
65   $propV' := extractLedger(objL', confL')$ 
66   $objL := objL \cup objL'$ 
67   $confL := confL \cup confL'$ 
68  if  $repV \sqsubseteq propV'$  then
69     $repV := propV'$ 
70    Include  $propV'.conf$  to  $pendConf$ 
71     $t_r := |repV.conf|$ 
72     $UpdateFSKeysDestroyOld(t_r)$ 
73    send  $\langle ACK, (HASH(propV'), lastDec, pendConf, activePropNb') \rangle$  to  $c$ 
74  else
75     $repV := repV \sqcup propV'$ 
76    send  $\langle NACK, (HASH(propV'), objL - objL', confL - confL', activePropNb') \rangle$  to  $c$ 

```

**upon** receive  $\langle DECISION, (objL', confL', ackL') \rangle$  from client  $c$

```

77   $lastDec := lastDec \cup extractLedger(objL', confL')$ 
78  Eliminate from  $pendConf$  subsets of  $lastDec.conf$ 
79   $objL := objL' \cup objL$ 
80   $confL := confL' \cup confL$ 

```

**operation**  $extractLedger(objL', confL')$

```

81  let  $receivedValue = (\sqcup[v \exists p, objL'[p] = v], \sqcup[c \exists p, confL'[p] = c])$ 
82  return  $receivedValue$ 

```

**Treating Replica Responses.** Once the client gets an ACK from a replica, it includes the message in its *ackL* (line 23) and registers the replica in its response set (line 25). Upon reception of a NACK, a client complements its *objL* and *confL* (lines 31 and 32) and sets the NACK bool, including the replica in its response set (lines 33 and 34). When a client sees that it has gathered responses from a set of replicas that satisfies the pending configurations, it proceeds to check its NACK bool, as the presence of a NACK means that it cannot decide yet, and if it is false, then it will decide (line 21).

Each proposal gets a unique number (*activePropNb*) so clients consider only reactions to the active proposal, ignoring late messages they might receive. Clients also ignore messages coming from replicas they already accused, as well as messages signed using timestamps that do not correspond to the configuration in their contents.

A client either waits until it gets responses from a set of replicas (keeping track via *RESPSet*) that satisfies the pending configurations or until it gets a newer decided configuration from another client broadcast. It is necessary to get majorities in all those combinations of system configurations because the client doesn't know if any combination of them was learnt by another client and must be sure that it has reached all possible active configurations that can be learnt before it learns one by itself. Furthermore, the state transfer from one replica to another will be directly provided by this procedure, as once a client learns a configuration the object information is already in place, which is one of the advantages of this solution. It becomes then necessary to keep track of the state of the system by including information about which was the last combination of decisions seen (variable *lastDec*), as well as pending configurations (variable *pendConf*).

**Issuing and Treating Convictions.** We keep an array of all output values instead of just the current one, as well as their corresponding ack ledgers, indexing the currently active entry by *activeOutNb*. This is necessary in order to monitor that after long delays in the network when two correct clients re-establish their connection they can still check if in this period their decisions were comparable (line 47) and be able to accuse processes that lead them to this incomparable state. The clients broadcast their accusations as well as their decisions.

They avoid issuing redundant accusations by keeping track of the variations (line 61). If a process gets new misbehavior proofs, it includes them on its accusations (line 62). Algorithm 1 shows one possible implementation of the *verify\_proof* function, checking that every issued accusation was made after a process tried to forge some signature, issued incoherent ACKs, tried to input values in the system in discordance with the specification or decided something without gathering the necessary acknowledgements.

Once a replica is accused by a client, the client begins to ignore the replica and the underlining application can for instance issue a reconfiguration effectively replacing it by one or more new replicas. The clients will then eventually learn comparable values once they join their values and the malicious replicas that try to subvert the system have been accused.

## 4.2 Correctness

We claim that the system of processes implemented following Algorithm 2, Algorithm 3, Algorithm 4 solves RALA.

► **Definition 3.** Let's define a state  $s$  as being the value of the variable *propV*. A state  $s$  is considered as decided when the first client  $c$  in state  $s$  broadcasts its decision at line 36. Moreover, we define  $s.lastDec$  and  $s.pendConf$  as being the value of these variables on the client  $c$  at the moment of the state decision. Finally,  $s.confComb$  is defined as the set  $\{s.lastDec.conf \cup \{d\} \mid d \in 2^{s.pendConf}\}$

► **Definition 4.** We define then a graph  $G_s$  whose vertices are the different decided states of the system plus the state  $(\perp, \text{initConf})$  and whose edges exist between two vertices  $s$  and  $s'$  whenever the following is true:

$$s \rightarrow s' \Leftrightarrow s \sqsubset s' \wedge s.\text{conf} \in s'.\text{confComb}$$

► **Lemma 5.** At every benign replica, the variables  $\text{repV}$ ,  $\text{pendConf}$  and  $\text{lastDec}$  are monotonically increasing.

**Proof.** The variable  $\text{repV}$  is updated in line 69 where it is assigned a new value which has passed the test in line 68, so the new value contains the old one.

The other updates involving these variables in lines 64, 70 and 75 are joins where one of the operands is the old value, so the new values must contain the old ones. ◀

► **Lemma 6.** Given decided states  $\bar{s}$ ,  $s$  and  $s'$  in  $G_s$ , if  $\bar{s} \rightarrow s$ ,  $\bar{s} \rightarrow s'$ ,  $s'.\text{lastDec} \sqsubseteq s$ ,  $s.\text{lastDec} \sqsubseteq s'$  then either there is an edge between  $s$  and  $s'$  or there is an accusation.

**Proof.** From  $\bar{s} \rightarrow s$ ,  $\bar{s} \rightarrow s'$  we derive that:

$$\bar{s}.\text{conf} \in s.\text{confComb} \wedge \bar{s}.\text{conf} \in s'.\text{confComb} \implies \bar{s}.\text{conf} \in s.\text{confComb} \cap s'.\text{confComb}$$

Because the decision only happens after triggering the event that begins in line 21, then it must be that the clients who decided these states got responses from replicas forming majority quorums in  $\bar{s}.\text{conf}$  and they must therefore intersect in at least a replica  $r$ .

Let us assume for now that  $r$  followed the algorithm and behaved correctly. Let  $c_s$  be the client that decided  $s$  and  $c_{s'}$  be the client that decided  $s'$ . Assuming w.l.o.g that the replica  $r$  served the client  $c_s$  before, using lemma 5 and observing that  $s$  and  $s'$  correspond to the first decision of these values,  $s \sqsubset s'$ . Since  $s'$  passed the test in line 63 in replica  $r$ , it means that  $s.\text{lastDec} \sqsubseteq s'.\text{lastDec}$ , moreover because we assume that  $s'.\text{lastDec} \sqsubseteq s$  we can write:

$$\begin{aligned} s.\text{conf} &= \bigsqcup(\{s.\text{lastDec}.\text{conf}\} \cup s.\text{pendConf}) \sqsubseteq \bigsqcup(\{s'.\text{lastDec}.\text{conf}\} \cup s.\text{pendConf}) \\ &\sqsubseteq \bigsqcup(\{s.\text{conf}\} \cup s.\text{pendConf}) = s.\text{conf} \end{aligned}$$

Furthermore, we see that all pending configurations in  $s$  which weren't included by the last decided configuration in  $s'$  must also be pending in  $s'$  because this information will be carried by the ack from replica  $r$  (line 24). We can then conclude:

$$\begin{aligned} s.\text{conf} &= \bigsqcup(\{s'.\text{lastDec}.\text{conf}\} \cup s.\text{pendConf}) \\ &= \bigsqcup(\{s'.\text{lastDec}.\text{conf}\} \cup \{u \in s.\text{pendConf}, u \not\sqsubseteq s'.\text{lastDec}.\text{conf}\}) \\ &\in \bigsqcup(\{s'.\text{lastDec}.\text{conf}\} \cup C|C \sqsubseteq s'.\text{pendConf}) = s'.\text{confComb} \end{aligned}$$

Hence there is an edge from  $s$  to  $s'$  in  $G_s$  in this scenario.

If the replica  $r$  didn't follow the algorithm and issued incomparable acks, then this event shall be detected and  $r$  will be accused.  $r$ 's ack would be included in both clients  $c_s$  and  $c_{s'}$  *ackLs* being broadcasted together with the decision in line 36 and once the first client who decided and then received the other's decision go through the line 49, it would find  $r$  in both ledgers and accuse it. ◀

## 25:12 Accountability and Reconfiguration: Self-Healing Lattice Agreement

► **Lemma 7.** *All the infinite connected components of  $G_s$  have the same suffix.*

**Proof.** Because we assume that the system reconfigures finitely many times, there is a point where all the decisions regarding the different configurations the system passed, which were broadcasted in line 36, arrive at the recipient clients. They'll process the configuration included in these values in lines 43 and 46 and the use of the forward secure signatures will prevent them from processing messages of old replicas. As a result, all the clients will have the same values of  $lastDec.conf$  and  $propV.conf$ , meaning that they will contact the same replicas and need to form a majority in the active configuration  $lastDec.conf \sqcup propV.conf$ , where their majority quorums intersect. As seen in the lemma 6 if any malicious replica tries to issue incomparable ACKs the clients will accuse it and ignore it from this point onward. They will retry the proposal again until none of the replicas in the majority misbehaves.

Let  $s$  be the first state decided in this scenario, henceforth the states will be totally ordered, sharing the same configuration which is always present in  $confComb$ , meaning that these states are connected. The graph will only have one growing branch and any new state  $s'$  in it will be a descendant of  $s$ . Finally, this branch will be infinite because the clients never stop proposing. ◀

► **Theorem 8.** *The system of processes implemented following Algorithm 2, Algorithm 3, Algorithm 4 provides validity.*

**Proof.** The learnt values by a client are extracted from its  $objL$  and  $confL$ .

First of all, at the beginning of a proposal (lines 9 and 10) the value present in the input buffer is read and put into the ledgers, guaranteeing that when a decision is made it shall be present in it.

These variables are then modified in lines 31, 32, 45 and 46. Therefore, the values included into them either come from the the input buffer from the clients where they are signed, or they are informed by replicas nacking proposals after passing signature check or by the information of other clients decisions. We conclude that the values learnt always come from the client input buffers directly or indirectly. ◀

► **Theorem 9.** *The system of processes implemented following Algorithm 2, Algorithm 3, Algorithm 4 provides completeness.*

**Proof.** By Lemma 6 whenever the graph  $G_s$  forks an accusation is issued. Each fork occur when clients learn incomparable values and are caused by some Byzantine replicas which are eventually accused. Moreover, by lemma 7 the system cannot be indefinitely forked and all inconsistencies are eventually solved when no new accusations are issued as required. ◀

► **Theorem 10.** *The system of processes implemented following Algorithm 2, Algorithm 3, Algorithm 4 provides accusation stability.*

**Proof.** The set of accused processes is reflected in the algorithm via the variable *accusation* which is updated in lines 49, 53, and 62. As one can see, they either attribute this variable to a union where one of the operands is itself or a value is explicitly included into it. Therefore after each update the new value must, by the definition of these operations, include the old one, i.e. the accusation set is monotonically increasing. ◀

► **Theorem 11.** *The system of processes implemented following Algorithm 2, Algorithm 3, Algorithm 4 provides accuracy.*



**Proof.** An accusation can be issued in lines 27, 49 and 53.

The first occurrence checks that an ACK was issued but the matching configuration proposal wasn't added by the replying replica as described in line 70 meaning that this line was skipped.

On the second case the decision of incomparable values requires, as seen earlier in Theorem 9 that a replica acknowledged incomparable values, violating the behavior of benign replicas described by Lemma 5. Having two ACKs signed by the same process for incomparable values characterises an irrefutable proof and the replicas in it will be a non-empty subset of  $M$  because they deviated from the algorithm.

On the last case a replica will be caught providing fake signatures, which is by itself enough to accuse it as this is a clear deviation from the algorithm. ◀

► **Theorem 12.** *The system of processes implemented following Algorithm 2, Algorithm 3, Algorithm 4 provides authenticity.*

**Proof.** Authenticity follows from our cryptographic assumptions and specially from three properties of the underlying system:

- Every message contains a signature;
- The signatures can be verified by a public function;
- No other process can sign on behalf of a correct process. ◀

► **Theorem 13.** *The system of processes implemented following Algorithm 2, Algorithm 3, Algorithm 4 provides agreement.*

**Proof.** Agreement is a direct consequence of the dissemination of information implemented in the algorithm. Every accusation is broadcasted and every message containing an accusation is analysed and, if it holds, leads to the adoption of the information (line 60). ◀

► **Theorem 14.** *The system of processes implemented following Algorithm 2, Algorithm 3, Algorithm 4 is alive.*

**Proof.** After the system stops reconfiguring a client can eventually receive the last configuration learnt by the broadcast in line 36 and then every client will contact active configurations. If a client then starts a proposal when receiving a value  $v$  in its input buffer, a majority of replicas in all active configurations shall eventually respond to this client, following our majority of correct replicas assumption in this scenario. From this point on, all decisions shall contain  $v$  as the last learnt configuration all clients contact will provide a majority of replicas that include  $v$ . ◀

## 5 Related Work

**Accountability.** In security terms, accountability ensures that the actions of an entity can be traced solely to that entity. This supports non-repudiation, deterrence, fault detection, and after-action recovery. Distributed computing research has focused for many years on failure detection [10, 12, 22], a close relative of accountability. By identifying faulty processes, failure detection helps the distributed computation to make progress in a safe way, but does not provide evidences of misbehaviors that can be verified by a third party. To the best of our knowledge, PeerReview [20] was the first proposing a general solution to provide accountability as an add-on feature for any distributed protocol. In PeerReview each process in the system records messages in tamper-evident logs: an auditor can challenge a process,

retrieve its logs, and simulate the original protocol to ensure that the process behaved correctly. By doing so, any observable deviating action can be traced back to at least one Byzantine process that was responsible for it. The main issue is that for an auditor to prove that a process is Byzantine it must receive a response to the challenge from the process. If no response is received, the auditor cannot determine whether the process is faulty or not. As a result, some Byzantine processes might be suspected forever and never proven guilty. This limitation is common to distributed protocols that are not designed to provide accountability.

Polygraph [11] equips Byzantine Consensus with an accountability mechanism. As in our system, the very messages sent during the protocol execution carry the necessary information to construct a proof in case of Consensus agreement violation. This way, there is no need to query processes to collect evidences and construct a proof. Fairledger [25] and LLB (Long-Lived Blockchain) [32] are consensus-based state-machine replication protocols that are able to detect consistency violations in consensus instances and reconfigure themselves. In contrast, we do not rely on consensus for reconfiguration and propose a purely asynchronous accountable and reconfigurable service.

**Lattice agreement.** Attiya et al. [4] introduced the (one-shot) lattice agreement abstraction and, in the shared-memory context, described a wait-free reduction of lattice agreement to atomic snapshot. Falerio et al. [14] introduced the long-lived version of lattice agreement (adopted in this paper), called generalized lattice agreement, and described an asynchronous message-passing implementation of lattice agreement assuming a majority of correct processes. In the Byzantine failure model, Di Luna et al [27] proposed for the first time a solution for Byzantine asynchronous generalized lattice agreement, later improved by [36]. All these algorithms propose a fault-tolerant approach where safety and liveness are guaranteed with  $f < n/3$  Byzantine processes and authenticated channels. In our accountability approach, liveness and recovery from safety violations are guaranteed with  $f < n/2$  Byzantine processes and authenticated channels.

**Asynchronous reconfiguration.** Dynastore [1] was the first solution emulating a reconfigurable atomic read/write register without consensus: clients can asynchronously propose incremental additions or removals to the system configuration. Since proposals commute, concurrent proposals are collected together without the need of deciding on a total order. In [21] it has been observed that asynchronous reconfiguration can be handled using an external reliable lattice-agreement object. Reconfigurable lattice agreement [23] enables reconfigurable versions of a large class of objects and abstractions, including state-based CRDTs [33], atomic-snapshot, max-register, conflict detector and commit-adopt.

In the Byzantine fault model, Dynamic Byzantine storage [3,30] allows a trusted *administrator* to issue ordered reconfiguration calls that might also change the set of replicas. More recently, [24] describes a generic Byzantine fault-tolerant reconfigurable lattice agreement, implemented without assuming a trusted administrator.

The reconfiguration technique used in this paper takes inspiration from [23] while been enriched with the use of forward-secure signatures as proposed in [24] to protect the system from Byzantine replicas belonging to old configurations. Note that none of the cited work provide proof-of-misbehavior of Byzantine processes.

## 6 Concluding remarks

In this paper, we propose the first design of an asynchronous replicated system that not only detects misbehavior that affects its safety properties, but is also able to mitigate misbehaving replicas by reconfiguration. Compare to earlier [16, 25] and concurrent [32] work, we do not employ consensus to agree on the evolving configurations. The algorithm described in this paper can be improved and generalized in multiple ways. Below we discuss some of them.

**Garbage collection.** In the current version of our algorithm, every process locally maintains a complete history of updates, and periodic reinitialization of the system is an important issue. In particular, it appears challenging to reinitialize the set of accusations, as a slow client may never be able to be convinced that a compromised replica is not trustful anymore. One may think, e.g., of a periodic instances of a consensus protocol among the clients to agree on the new initial system state, running in parallel with our algorithm. Altogether, periodic “truncation” of the ever-growing state in an asynchronous protocol remains an interesting question for the future work.

**Complexity.** Similar to earlier solutions of (generalized) lattice agreement [14, 23], the latency of learning a value in our algorithm (in the number of asynchronous query-response rounds, assuming that the configuration does not change) is proportional to the number of concurrently proposed values. It remains unclear if there is an asymptotically faster algorithm. There are interesting solutions for *one-shot* Byzantine lattice agreement that take  $\log k$  rounds for  $k$  proposed values [37], but we do not have a comparable long-lived implementation.

For simplicity, in our algorithm, the sizes of messages grow linearly with the number of distinct values learnt by the clients. One can improve this by sending relative updates instead of complete histories in PROPOSE and DECISION messages. The size of ACK and NACK messages already grow much slower, as they use digests of corresponding proposals and only contain information about changes: in the case of ACKs, these changes consist of the pending configurations since last decision and in the case of NACKs – with respect to the proposed value the replica is responding to. An ACCUSATION message has asymptotic complexity of an ACK message. The issue of maintaining bounds on ever-growing message sizes is related to the more general question of garbage-collection and reinitialization.

**Clients: Byzantine and heavy.** Early proposals of quorum-based fault-tolerant storage systems typically assumed that clients are benign (see, e.g., [29]). While the effect of Byzantine *writers* can be mitigated using erasure coding [17] or voting [26], it appears nontrivial to handle malicious reconfiguration requests. Indeed, a Byzantine client can block progress by plunging the system in constant reconfiguration, or break safety of the replicated data by rendering the system to a compromised configuration. How to handle such attacks is an intriguing challenge.

Assuming that the clients are benign enables assigning them with a major part of the total work. This results in linear message complexity: the replicas only passively respond to clients’ queries.

Alternatively, we may follow earlier work on asynchronous Byzantine reconfiguration [24], and assume an external *access control* mechanism ensuring that inputs from the clients (including reconfiguration calls) are “acceptable”. In particular, the proposed configurations should satisfy the configuration availability condition (Section 3): every combination of

candidate configurations should contain enough correct replicas. Also, the access control mechanism should provide a verification procedure that would allow the third party to verify validity of reconfiguration requests. The clients are then only responsible for submitting valid to the set of replicas. The resulting algorithm will, however, likely to be more costly in terms of message complexity, as each reconfiguration will have to handle each of the valid requests.

Our algorithm can also be easily extended to accommodate partitions of the clients into (benign) administrators and (Byzantine-prone) users, along the lines of [25,30].

For completeness, in Appendix A, we describe a specification and a corresponding implementation of a *one-shot* lattice agreement abstraction that assumes that both clients and replicas can be Byzantine. Our system is particularly well suited for the client-administrator approach as the reconfiguration requests are issued by the proposing entities (in this case an administrator) and not the entities maintaining the system (the replicas).

**Reconfiguration strategy.** In this paper, we delegated the task of choosing new configurations to the clients. The clients are free to reconfigure the system even if no new misbehaving replicas are detected. The only requirement we impose on the configurations proposed by the clients is that resulting configurations must remain available (Section 3). But one may think of more explicit reconfiguration strategies. For example, each time a new misbehaving replica is detected, it is replaced with a new one taken from a “pool” of correct replicas (a similar approach is proposed in LLB [32] for consensus-based reconfiguration).

---

## References

- 1 M. K. Aguilera, I. Keidar, D. Malkhi, and A. Shraer. Dynamic atomic storage without consensus. *J. ACM*, 58(2):7:1–7:32, 2011.
- 2 E. Alchieri, A. Bessani, F. Greve, and J. da Silva Fraga. Efficient and modular consensus-free reconfiguration for fault-tolerant storage. In *OPODIS*, pages 26:1–26:17, 2017.
- 3 L. Alvisi, D. Malkhi, E. Pierce, M. K. Reiter, and R. N. Wright. Dynamic byzantine quorum systems. In *Proceeding International Conference on Dependable Systems and Networks. DSN 2000*, pages 283–292. IEEE, 2000.
- 4 H. Attiya, M. Herlihy, and O. Rachman. Atomic snapshots using lattice agreement. *Distributed Comput.*, 8(3):121–132, 1995.
- 5 M. Bellare and S. K. Miner. A forward-secure digital signature scheme. In *Annual International Cryptology Conference*, pages 431–448. Springer, 1999.
- 6 X. Boyen, H. Shacham, E. Shen, and B. Waters. Forward-secure signatures with untrusted update. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 191–200, 2006.
- 7 G. Bracha. Asynchronous Byzantine agreement protocols. *Information and Computation*, 75(2):130–143, Nov. 1987.
- 8 C. Cachin, R. Guerraoui, and L. Rodrigues. *Introduction to reliable and secure distributed programming*. Springer Science & Business Media, 2011.
- 9 M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, 2002.
- 10 T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, July 1996.
- 11 P. Civit, S. Gilbert, and V. Gramoli. Polygraph: Accountable byzantine agreement. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pages 403–413. IEEE, 2021.
- 12 C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, V. Hadzilacos, P. Koutnetzov, and S. Toueg. The weakest failure detectors to solve certain fundamental problems in distributed computing. In *Proceedings of the 23th ACM Symposium on Principles of Distributed Computing*, 2004.

- 13 M. Drijvers, S. Gorbunov, G. Neven, and H. Wee. Pixel: Multi-signatures for consensus. In *29th USENIX Security Symposium (USENIX Security 20)*, Boston, MA, Aug. 2020. USENIX Association.
- 14 J. Faleiro, S. Rajamani, K. Rajan, G. Ramalingam, and K. Vaswani. Generalized lattice agreement. In *PODC*, pages 125–134, 2012.
- 15 E. Gafni and D. Malkhi. Elastic configuration maintenance via a parsimonious speculating snapshot solution. In *DISC*, pages 140–153, 2015.
- 16 S. Gilbert, N. A. Lynch, and A. A. Shvartsman. Rambo: a robust, reconfigurable atomic memory service for dynamic networks. *Distributed Comput.*, 23(4):225–272, 2010.
- 17 G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter. Efficient byzantine-tolerant erasure-coded storage. In *(DSN)*, pages 135–144. IEEE Computer Society, 2004.
- 18 A. Haeberlen and P. Kuznetsov. The Fault Detection Problem. In *Proceedings of the 13th International Conference on Principles of Distributed Systems (OPODIS'09)*, Dec. 2009.
- 19 A. Haeberlen, P. Kuznetsov, and P. Druschel. The case for byzantine fault detection. In *Proceedings of the Second Workshop on Hot Topics in System Dependability (HotDep'06)*, Nov. 2006.
- 20 A. Haeberlen, P. Kuznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)*, Oct. 2007.
- 21 L. Jehl, R. Vitenberg, and H. Meling. Smartmerge: A new approach to reconfiguration for atomic storage. In *DISC*, pages 154–169, 2015.
- 22 K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. Byzantine fault detectors for solving consensus. *Comput. J.*, 46(1):16–35, 2003.
- 23 P. Kuznetsov, T. Rieutord, and S. Tucci-Piergiovanni. Reconfigurable lattice agreement and applications. In *OPODIS*, 2019.
- 24 P. Kuznetsov and A. Tonkikh. Asynchronous reconfiguration with byzantine failures. In H. Attiya, editor, *DISC*, volume 179 of *LIPICs*, pages 27:1–27:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.
- 25 K. Lev-Ari, A. Spiegelman, I. Keidar, and D. Malkhi. Fairledger: A fair blockchain protocol for financial institutions. In *OPODIS*, volume 153 of *LIPICs*, pages 4:1–4:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.
- 26 B. Liskov and R. Rodrigues. Tolerating byzantine faulty clients in a quorum system. In *ICDCS*, page 34. IEEE Computer Society, 2006.
- 27 G. A. D. Luna, E. Anceaume, and L. Querzoni. Byzantine generalized lattice agreement. In *IPDPS*, pages 674–683. IEEE, 2020.
- 28 T. Malkin, D. Micciancio, and S. Miner. Efficient generic forward-secure signatures with an unbounded number of time periods. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 400–417. Springer, 2002.
- 29 J. Martin, L. Alvisi, and M. Dahlin. Minimal byzantine storage. In D. Malkhi, editor, *DISC*, volume 2508 of *Lecture Notes in Computer Science*, pages 311–325. Springer, 2002.
- 30 J.-P. Martin and L. Alvisi. A framework for dynamic byzantine storage. In *International Conference on Dependable Systems and Networks, 2004*, pages 325–334. IEEE, 2004.
- 31 M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, Apr. 1980.
- 32 A. Ranchal-Pedrosa and V. Gramoli. Blockchain is dead, long live blockchain! accountable state machine replication for longlasting blockchain. *CoRR*, abs/2007.10541, 2020.
- 33 M. Shapiro, N. M. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *SSS*, pages 386–400, 2011.
- 34 A. Spiegelman and I. Keidar. On liveness of dynamic storage. In *Structural Information and Communication Complexity - 24th International Colloquium, SIROCCO 2017, Porquerolles, France, June 19-22, 2017, Revised Selected Papers*, pages 356–376, 2017.

- 35 A. Spiegelman, I. Keidar, and D. Malkhi. Dynamic reconfiguration: Abstraction and optimal asynchronous solution. In *DISC*, pages 40:1–40:15, 2017.
- 36 X. Zheng and V. K. Garg. Byzantine lattice agreement in asynchronous systems. In *24th International Conference on Principles of Distributed Systems, OPODIS 2020, December 14–16, 2020, Strasbourg, France (Virtual Conference)*, pages 4:1–4:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.
- 37 X. Zheng and V. K. Garg. Byzantine lattice agreement in asynchronous systems. In Q. Bramas, R. Oshman, and P. Romano, editors, *OPODIS*, volume 184 of *LIPICs*, pages 4:1–4:16, 2020.

## A Accountable Lattice Agreement with Byzantine Clients

In this section, we discuss a *one-shot* static version of accountable lattice agreement that considers that both clients and replicas might be Byzantine.

### A.1 Problem statement

The *general accountable one-shot lattice agreement (A1LA)* abstraction, defined on a lattice  $(\mathcal{L}, \sqsubseteq)$ , takes, as a single input, an element in  $\mathcal{L}$  and produces, as an output, a pair of an element in  $\mathcal{L}$  and a set of *accusations*. Again, an accusation is a pair  $(A, P)$  where  $A \subset \Pi$  and  $P$  is a *proof of misbehavior*. And we assume that the proof can be independently verified by a third party through a boolean map *verify-proof*:  $(2^\Pi \times \mathcal{P}) \rightarrow \{\text{true}, \text{false}\}$ .

We say that this version is general because both clients and replicas can be malicious. The system contains  $N$  replicas where a majority of them are correct. Let  $U \subseteq B$  be the finite set of benign clients that proposed values in that run, and let  $u_i$  denote the value proposed by a process  $p_i \in U$ . Let  $V_B$  and  $V_C$  be the sets of, respectively, benign and correct clients that learned values in that run, and let  $v_i$  denote the value learned by a process  $p_i \in V_B$  (obviously,  $V_C \subseteq V_B \subseteq U$ ). The A1LA abstraction satisfies the following properties:

- **Validity.** The value learnt by a benign client  $c_i$  with input value  $u_i$  is a join of values proposed by clients in  $U$  (including  $c_i$ ), at most  $|M|$  values coming from  $M$ , and  $u_i$ :

$$\forall c_i \in V_B : u_i \sqsubseteq v_i \wedge v_i \sqsubseteq \sqcup(\{u_j | c_j \in U\} \cup F), F \subseteq \mathcal{L}, |F| \leq |M|.$$

- **Consistency.** Either the values learned by the correct clients are totally ordered:

$$\forall c_i, c_j \in V_C : v_i \sqsubseteq v_j \vee v_i \sqsupseteq v_j.$$

or every correct process eventually accuses a set of processes.

- **Accuracy.** If a benign process  $p_i$  accuses  $A$  (with  $P$ ), then  $A$  is a subset of  $M$  and  $P$  contains a proof against each process in  $A$ .

$$\forall p_i \in B, A \subseteq M \wedge \text{verify-proof}(A, P)$$

- **Authenticity.** It is computationally infeasible to construct  $A \cap B \neq \emptyset$  and  $P \in \mathcal{P}$  such that  $\text{verify-proof}(A, P) = \text{true}$ .
- **Liveness.** If a correct client proposes a value, it eventually learns a value or accuses a set of processes.

One can see that a benign client can accuse a set of processes only if there is at least one Malicious process ( $M \neq \emptyset$ ) and in the absence of malicious processes no proofs of misbehavior are created. The Consistency property guarantees that either *correct* clients learn comparable values or some malicious process will be accused. Notice that we cannot

avoid executions in which a *Benign* but not correct client learns a value that is inconsistent with a value learnt by another benign client if there are malicious processes without issuing accusations.

## A.2 The algorithm

Our solution to A1LA is presented in Algorithm 5, Algorithm 6 and Algorithm 7. As before, each method is executed in its entirety without being interrupted. Each client might be in an *active* state where it proposes values and takes steps towards learning something new, re-proposing if necessary or in a *passive* state where it only reacts to other client proposals. On start-up clients that receive input values from the application sign them and put them to their input ledgers (line 84) and proceed to propose it to the system by multicasting (91).

When a replica receives a proposal with a ledger, it extracts the proposed value by the other process merging the new inputs to its own ledger (117). Before treating the ledger a verification is made to guarantee its integrity (113). Once the message is validated, there might be inconsistencies in the ledger introduced by malicious processes that tried to insert more than one value in the system and an accusation might be issued (118). Otherwise the ledger is consistent and an ACK can then be produced if the proposal comprises the previously received ones (126), otherwise a NACK shall be sent informing the proposer of values that it didn't include in its proposal via a complement to its ledger (130).

Received ACKs are discarded if they correspond to old proposals, or if they come from a process whose ACK has already been accounted in the current proposal, or if they don't match the proposed value they were sent for. Note that the latter is in itself a sign of byzantine behavior but doesn't constitute an irrefutable proof of misbehavior as the conflicting messages, i.e. the proposal and the ACK, are signed by different processes. When the ACK is valid it is included in the *ackL* (97) and the respective counter is incremented (98). Similarly, outdated NACKs are ignored when received, as well as empty ledgers, ledgers who don't include new values (101). Once the ledger is validated, the proposed value is set to include the information patch (87) and the counter of NACKs is incremented (103).

After a client has received responses from at least a majority of replicas (99 and 104) for its proposal it proceeds to evaluate if it can decide on a value or not, in case it has not accused any malicious processes along the way. If a NACK has been received, it tries to learn its new proposal (92) which includes the missing values in the previous attempt, otherwise it decides upon its proposal and broadcasts it alongside the ledger of ACKs it has collected (96). At this point clients who are proposing incomparable values to the one decided check the ACKs they received so far as byzantine processes can lead the system to decide incomparable values by issuing contradictory ACKs for different processes, which can be detected at this point and lead to their accusation (110).

■ **Algorithm 5** Accountable One-Shot Lattice Agreement: Code for client  $c$  part 1.

**Local variables:**

$status$ , initially passive { Boolean for the current state: passive or active }  
 $ackCnt$ , initially 0 { The number of acks received for the active proposal }  
 $nackCnt$ , initially 0 { The number of nacks received for the active proposal }  
 $activePropNb$ , initially  $-1$  { The number of nacks received for the active proposal }  
 $propV$ , initially  $\perp$  { The value being proposed }  
 $inL$ , initially empty { KV table (ledger) init. w/ proposed values signed by their originators }  
 $ackL$ , initially empty { Key value table holding received signed acks by replicas }

**Input:**

$initV$  { Value initially proposed by the process, provided by external source }

**Outputs:**

$outV$ , initially  $\perp$  { Value learnt by the client }  
 $accusation$ , initially  $\emptyset$  { Proofs of misbehavior gathered }

**upon** *startup* **if**  $initV \neq \perp$

83  $propV := initV$   
 84 include *signed*  $initV$  to  $inL$   
 85 Propose

**operation** *Propose*

86  $status := active$   
 87  $propV := extractLedger(inL, c)$   
 88  $activePropNb := activePropNb + 1$   
 89 clear  $ackL$   
 90  $ackCnt := nackCnt := 0$   
 91  $multicast \langle PROPOSAL, (inL, activePropNb) \rangle$  to Servers

**operation** *EvaluateDecision*

92 **if**  $nackCnt > 0$  **then** Propose  
 93 **else**  
 94  $outV := propV$   
 95  $status := passive$   
 96  $multicast \langle DECISION, (outV, ackL) \rangle$  to Servers

**upon** *receive*  $\langle ACK, (propV, activePropNb) \rangle$

from given replica  $r$  AND  $r \notin ackL$   $status = active$   
 97 append  $q$ 's ack to  $ackL$   
 98  $ackCnt := ackCnt + 1$   
 99 **if**  $ackCnt + nackCnt \geq \lceil \frac{N+1}{2} \rceil$  **then** EvaluateDecision

**upon** *receive*  $\langle NACK, (\Delta Ledger, activePropNb) \rangle$  from process  $q$

AND  $status = active$  AND  $\Delta Ledger \neq \emptyset$   
 100  $\Delta Value = extractLedger(\Delta Ledger, q)$   
 101 **if**  $\Delta Value \sqsubseteq propV$  **return**  
 102  $inL := inL \cup (inL')$   
 103  $nackCnt := nackCnt + 1$   
 104 **if**  $ackCnt + nackCnt \geq \lceil \frac{N+1}{2} \rceil$  **then** EvaluateDecision



---

**Algorithm 6** Accountable One-Shot Lattice Agreement: Code for client  $c$  part 2.

---

```

upon receive  $\langle DECISION, (outV', ackL') \rangle$  from process  $q$ 
105 if  $\exists (p, v) \in ackL' | v \neq outV'$ 
106      $accusation := accusation \cup \{(q, DECISION)\}$ 
107      $status = passive$ 
108     return
109 if  $outV' \not\sqsubseteq propV \ \&\& \ propV \not\sqsubseteq outV'$  then
110     let  $M = \{m | m \in ackL \ \&\& \ ackL'\}$  do
111         foreach  $m \in M$  do include  $(b, \{ackL[b], ackL'[b]\})$  to  $accusation$ 
112         if  $|M| > 0$  then  $status := passive$ 

operation  $extractLedge$  ( $inL', sender$ )
113 if  $\exists$  process  $p \in inL'$  with invalid signature then
114      $accusation := accusation \cup \{(sender, getPropNACKMSG(inL'))\}$ 
115      $status := passive$ 
116     return  $\emptyset$ 
117  $inL'' := inL \cup (inL')$ 
118 let  $M = \{m | m \in inL'' \ \&\& \ |inL''[m]| > 1\}$  do
119     foreach  $m \in M$  do include  $(m, getPropNACKMSG(inL''[m]))$  to  $accusation$ 
120      $status := passive$ 
121     return  $\emptyset$ 
122 let  $receivedValue = \sqcup [v | \exists p, inL'[p] = v]$ 
123     return  $receivedValue$ 

```

---



---

**Algorithm 7** Accountable One-Shot Lattice Agreement: Code for replica  $r$ .
**Local variables:** $inL$ , initially empty $\{$  Key value table holding initially proposed values signed by their originators  $\}$  $repV$  initially  $\perp$   $\{$  Value held by the replica  $\}$  $accusation$ , initially  $\emptyset$   $\{$  Proofs of misbehavior gathered by the replica  $\}$ **upon** receive  $\langle PROPOSAL, (inL', activePropNb') \rangle$  from process  $q$ 124  $propV' := extractLedge(inL')$ 125 **if**  $repV \sqsubseteq propV'$  **then**126 send  $\langle ACK, (propV', activePropNb') \rangle$  to  $q$ 127  $repV := propV'$ 128 **else**129  $repV := repV \sqcup propV'$ 130 send  $\langle NACK, (inL - inL', activePropNb') \rangle$  to  $q$ **operation**  $extractLedge$  ( $inL', sender$ ) $\{$  Identical to client operation with the same name without lines 115 and 120  $\}$

### A.3 Correctness

We claim that the algorithm presented in Algorithm 5, Algorithm 6 and Algorithm 7 solves the General Accountable One-Shot Lattice Agreement.

► **Lemma 15.** *At every benign process, the variable  $propV$  is monotonically increasing.*

**Proof.** The variable is updated only in line 87. As one can see, it is a join operation where one of the operands is the previous value, hence the new value by definition contains the old value. ◀

► **Lemma 16.** *If a benign process  $p$  learns a value  $v$ , then  $v$  cannot contain two or more values signed by the same process.*

**Proof.** A process adds values to its proposal  $propV$ , when it receives a *nack* response. The insertion is then subject to verification following line 118. Process  $p$  will only proceed to a deciding a value if the list comprehension yields an empty list, in which case there is at most one value introduced on its proposal per process in the system. ◀

► **Lemma 17.** *At every benign process, the variable  $repV$  is monotonically increasing.*

**Proof.** The variable is updated in lines 127 and 129. The first update assigns to this variable a new value which has passed the test in line 125, so the new value contains the old one. Similarly to  $propV$ , the second update is a join where one of the operands is the old value, so the new value must contain the old one. ◀

► **Theorem 18.** *The Algorithm presented in Algorithm 5, Algorithm 6 and Algorithm 7 provides consistency.*

**Proof.** Suppose, by contradiction, that two benign processes  $p$  and  $q$  learned two incomparable values  $v'$  and  $v''$ . The majority that acknowledged  $v'$  at  $p$  must intersect with the majority that acknowledged  $v''$  at  $q$ . Let  $r$  be any process in the intersection. If  $r$  is not Byzantine then by Lemma 17,  $v'$  and  $v''$  must be comparable and *consistency* will hold.

Otherwise,  $r$  must have acked incomparable values, which shall be detected in line 110 meaning that the processes that output incomparable values will accuse  $r$ . ◀

► **Theorem 19.** *The Algorithm presented in Algorithm 5, Algorithm 6 and Algorithm 7 provides validity.*

**Proof.** The inclusion of the process own proposal follows from Lemma 15 with the initialisation of  $propV$  to  $initV$ , remarking that  $outV$  is but one of the values taken by  $propV$ .

As for the cap on the number of values coming from byzantine processes, suppose that there are at least  $|M| + c$ , where  $c \in \mathbb{N}^*$ , values coming from byzantine processes. It means that at least one byzantine process  $b$  signed two or more initial values that are output by a benign process. Because of Lemma 16, this is impossible. ◀

► **Theorem 20.** *The Algorithm presented in Algorithm 5, Algorithm 6 and Algorithm 7 provides accuracy.*

**Proof.** An accusation can be issued in lines 114, 119, 111.

On the first case it will have a proposal signed by a process which doesn't hold valid signed origins for its values. One of the values can come from the process itself, in which case a benign process would have signed it and put in its ledger on the initialisation. The other values must come through NACKs that also provide signed origins obtained in line 117

which benign processes include in its ledger. Having a proposal signed by a process which provided fake signatures to the origin of its values consist as an irrefutable proof and  $M'$  will be a non-empty subset of  $M$ .

The second scenario tracks processes that have inserted more than one value in the system. A benign process would only do it once during initialisation and having two inclusions signed by the same process consists as irrefutable proof and  $M'$  will be a non-empty subset of  $M$ .

Finally, the decision of incomparable values requires, as seen earlier in Theorem 18 that a process acknowledged incomparable values, violating the behavior of benign processes described by Lemma 17. Having two ACKs signed by the same process for incomparable values characterises as irrefutable proof and  $M'$  will be a non-empty subset of  $M$ . ◀

► **Theorem 21.** *The Algorithm presented in Algorithm 5, Algorithm 6 and Algorithm 7 provides authenticity.*

**Proof.** This property is exactly the same as Theorem 12. ◀

► **Theorem 22.** *The Algorithm presented in Algorithm 5, Algorithm 6 and Algorithm 7 provides liveness.*

**Proof.** Following Theorem 19 combined with Lemma 15, each run can have at most  $|U|$  different values being proposed. Since by the end of this many proposals a client shall propose the join of all these values, it will get ACKs from a majority of processes proceeding to learn a value or gather enough information for accusing at least one byzantine process, as at this byzantine clients must have introduced more than one value in the system for the proposal not to go through and Theorem 20 holds. ◀



# Design and Analysis of a Logless Dynamic Reconfiguration Protocol

William Schultz ✉

Northeastern University, Boston, MA, USA

Siyuan Zhou ✉

MongoDB, New York, NY, USA

Ian Dardik ✉

Northeastern University, Boston, MA, USA

Stavros Tripakis ✉

Northeastern University, Boston, MA, USA

---

## Abstract

Distributed replication systems based on the replicated state machine model have become ubiquitous as the foundation of modern database systems. To ensure availability in the presence of faults, these systems must be able to dynamically replace failed nodes with healthy ones via *dynamic reconfiguration*. MongoDB is a document oriented database with a distributed replication mechanism derived from the Raft protocol. In this paper, we present *MongoRaftReconfig*, a novel dynamic reconfiguration protocol for the MongoDB replication system. *MongoRaftReconfig* utilizes a logless approach to managing configuration state and decouples the processing of configuration changes from the main database operation log. The protocol's design was influenced by engineering constraints faced when attempting to redesign an unsafe, legacy reconfiguration mechanism that existed previously in MongoDB. We provide a safety proof of *MongoRaftReconfig*, along with a formal specification in TLA+. To our knowledge, this is the first published safety proof and formal specification of a reconfiguration protocol for a Raft-based system. We also present results from model checking the safety properties of *MongoRaftReconfig* on finite protocol instances. Finally, we discuss the conceptual novelties of *MongoRaftReconfig*, how it can be understood as an optimized and generalized version of the single server reconfiguration algorithm of Raft, and present an experimental evaluation of how its optimizations can provide performance benefits for reconfigurations.

**2012 ACM Subject Classification** Information systems → Parallel and distributed DBMSs; Software and its engineering → Software verification

**Keywords and phrases** Fault Tolerance, Dynamic Reconfiguration, State Machine Replication

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2021.26

**Related Version** *Full Version*: <https://arxiv.org/abs/2102.11960> [28]

**Supplementary Material** *Software (TLA+ specifications [26])*: <https://doi.org/10.5281/zenodo.5715510>

**Funding** This work has been partially supported by NSF award CNS-1801546.

**Acknowledgements** We would like to thank Tess Avitabile for her critical insights during the development of the reconfiguration protocol and discovery of subtle bugs in early design proposals. We would like to thank Judah Schvimer, A. Jesse Jiryu Davis, Pavi Vetriselvan, and Ali Mir for offering helpful insights during the protocol design and implementation process. We would also like to thank Shuai Mu for providing helpful comments on initial drafts of this paper.



© William Schultz, Siyuan Zhou, Ian Dardik, and Stavros Tripakis;  
licensed under Creative Commons License CC-BY 4.0

25th International Conference on Principles of Distributed Systems (OPODIS 2021).

Editors: Quentin Bramas, Vincent Gramoli, and Alessia Milani; Article No. 26; pp. 26:1–26:16

Leibniz International Proceedings in Informatics



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

## 1 Introduction

Distributed replication systems based on the replicated state machine model [24] have become ubiquitous as the foundation of modern, fault-tolerant data storage systems. In order for these systems to ensure availability in the presence of faults, they must be able to dynamically replace failed nodes with healthy ones, a process known as *dynamic reconfiguration*. The protocols for building distributed replication systems have been well studied and implemented in a variety of systems [4, 7, 9, 30]. Paxos [12] and, more recently, Raft [22], have served as the logical basis for building provably correct distributed replication systems. Dynamic reconfiguration, however, is an additionally challenging and subtle problem [1] that has not been explored as extensively as the foundational consensus protocols underlying these systems. Variants of Paxos have examined the problem of dynamic reconfiguration but these reconfiguration techniques may require changes to a running system that impact availability [14] or require the use of an external configuration master [15]. The Raft consensus protocol, originally published in 2014, provided a dynamic reconfiguration algorithm in its initial publication, but did not include a precise discussion of its correctness or include a formal specification or proof. A critical safety bug [20] in one of its reconfiguration protocols was found after initial publication, demonstrating that the design and verification of reconfiguration protocols for these systems is a challenging task.

MongoDB [17] is a general purpose, document oriented database which implements a distributed replication system [27] for providing high availability and fault tolerance. MongoDB's replication system uses a novel consensus protocol that derives from Raft [34]. Since its inception, the MongoDB replication system has provided a custom, legacy protocol for dynamic reconfiguration of replica members that was not based on a published algorithm. This legacy protocol managed configurations in a *logless* fashion i.e. each server only stored its latest configuration. In addition, it decoupled reconfiguration processing from the main database operation log. These features made for a simple and appealing protocol design, and it was sufficient to provide basic reconfiguration functionality to clients. The legacy protocol, however, was known to be unsafe in certain cases. In recent versions of MongoDB, reconfiguration has become a more common operation, necessitating the need for a redesigned, safe reconfiguration protocol with rigorous safety guarantees. From a system engineering perspective, a primary goal was to keep design and implementation complexity low. Thus, it was desirable that the new reconfiguration protocol minimize changes to the legacy protocol to the extent possible. In this paper, we present *MongoRaftReconfig*, a novel dynamic reconfiguration protocol that achieves the above design goals.

*MongoRaftReconfig* provides safe, dynamic reconfiguration, utilizes a logless approach to managing configuration state, and decouples reconfiguration processing from the main database operation log. Thus, it bears a high degree of architectural and conceptual similarity to the legacy MongoDB protocol, satisfying our original design goal of minimizing changes to the legacy protocol. We provide rigorous safety guarantees of *MongoRaftReconfig*, including a proof of the protocol's main safety properties along with a formal specification in TLA+ [16], a specification language for describing distributed and concurrent systems. To our knowledge, this is the first published safety proof and formal specification of a reconfiguration protocol for a Raft-based system. We also verified the safety properties of finite instances of *MongoRaftReconfig* using the TLC model checker [33], which provides additional confidence in its correctness. Finally, we discuss the conceptual novelties of *MongoRaftReconfig*, related to its logless design and decoupling of reconfiguration processing. In particular, we discuss how it can be understood as an optimized and generalized variant of

the single server Raft reconfiguration protocol. We also include a preliminary experimental evaluation of how these optimizations can provide performance benefits over standard Raft, by allowing reconfigurations to bypass the main operation log.

To summarize, in this paper we make the following contributions:

- We present *MongoRaftReconfig*, a novel, logless dynamic reconfiguration protocol for the MongoDB replication system.
- We present a proof of *MongoRaftReconfig*'s key safety properties. To our knowledge, this is the first published safety proof of a reconfiguration protocol for a Raft-based system.
- We present a formal specification of *MongoRaftReconfig* in TLA+. To our knowledge, this is the first published formal specification of a reconfiguration protocol for a Raft-based system.
- We present results of model checking the safety properties of *MongoRaftReconfig* on finite protocol instances using the TLC model checker.
- We discuss the conceptual novelties of *MongoRaftReconfig*, and how it can be understood as an optimized and generalized variant of the single server Raft reconfiguration protocol.
- We provide a preliminary experimental evaluation of *MongoRaftReconfig*'s performance benefits, demonstrating how it improves upon reconfiguration in standard Raft.

## 2 Background

### 2.1 System Model

Throughout this paper, we consider a set of *server* processes  $Server = \{s_1, s_2, \dots, s_n\}$  that communicate by sending messages. We assume an asynchronous network model in which messages can be arbitrarily dropped or delayed. We assume servers can fail by stopping but do not act maliciously i.e. we assume a “fail-stop” model with no Byzantine failures. We define both a *member set* and a *quorum* as elements of  $2^{Server}$ . Member sets and quorums have the same type but refer to different conceptual entities. For any member set  $m$ , and any two non-empty member sets  $m_i, m_j$ , we define the following:

$$Quorums(m) \triangleq \{s \in 2^m : |s| \cdot 2 > |m|\} \quad (1)$$

$$QuorumsOverlap(m_i, m_j) \triangleq \forall q_i \in Quorums(m_i), q_j \in Quorums(m_j) : q_i \cap q_j \neq \emptyset \quad (2)$$

where  $|S|$  denotes the cardinality of a set  $S$ . We refer to Definition 2 as the *quorum overlap* condition.

### 2.2 Raft

Raft [19] is a consensus protocol for implementing a replicated log in a system of distributed servers. It has been implemented in a variety of systems across the industry [21]. Throughout this paper, we refer to the original Raft protocol as described and specified in [19] as *standard Raft*.

The core Raft protocol implements a replicated state machine using a static set of servers. In the protocol, time is divided into *terms* of arbitrary length, where terms are numbered with consecutive integers. Each term has at most one leader, which is selected via an *election* that occurs at the beginning of a term. To dynamically change the set of servers operating the protocol, Raft includes two, alternate algorithms: *single server membership change* and *joint consensus*. In this paper we are only concerned with *single server membership change*. The single server change approach aims to simplify reconfiguration by allowing only reconfigurations that add or remove a single server. Reconfiguration is accomplished by

writing a special reconfiguration entry into the main Raft operation log that alters the local configuration of a server. In this paper, when referring to reconfiguration in standard Raft, we assume it to mean the single server change protocol.

## 2.3 Replication in MongoDB

MongoDB is a general purpose, document oriented database that stores data in JSON-like objects. A MongoDB database consists of a set of collections, where a collection is a set of unique documents. To provide high availability, MongoDB provides the ability to run a database as a *replica set*, which is a set of MongoDB servers that act as a consensus group, where each server maintains a logical copy of the database state.

MongoDB replica sets utilize a replication protocol that is derived from Raft, with some extensions. We refer to MongoDB's abstract replication protocol, without dynamic reconfiguration, as *MongoStaticRaft*. This protocol can be viewed as a modified version of standard Raft that satisfies the same underlying correctness properties. A more in depth description of *MongoStaticRaft* is given in [34, 27], but we provide a high level overview here, since the *MongoRaftReconfig* reconfiguration protocol is built on top of *MongoStaticRaft*. In a replica set running *MongoStaticRaft* there exists a single *primary* server and a set of *secondary* servers. As in standard Raft, there is a single primary elected per term. The primary server accepts client writes and inserts them into an ordered operation log known as the *oplog*. The oplog is a logical log where each entry contains information about how to apply a single database operation. Each entry is assigned a monotonically increasing timestamp, and these timestamps are unique and totally ordered within a server log. These log entries are then replicated to secondaries which apply them in order leading to a consistent database state on all servers. When the primary learns that enough servers have replicated a log entry in its term, the primary will mark it as *committed*, guaranteeing that the entry is permanently durable in the replica set. Clients of the replica set can issue writes with a specified *write concern* level, which indicates the durability guarantee that must be satisfied before the write can be acknowledged to the client. Providing a write concern level of *majority* ensures that a write will not be acknowledged until it has been marked as committed in the replica set. A key, high level safety requirement of the replication protocol is that if a write is acknowledged as committed to a client, it should be durable in the replica set.

## 3 **MongoRaftReconfig: A Logless Dynamic Reconfiguration Protocol**

In this section we present the *MongoRaftReconfig* dynamic reconfiguration protocol. First, we provide an overview and some intuition on the protocol design in Section 3.1. Section 3.2 provides a high level, informal description of the protocol along with a condensed pseudocode description in Algorithm 1. Sections 3.3 and 3.4 provide additional detail on the mechanisms required for the protocol to operate safely, and the TLA+ formal specification of *MongoRaftReconfig* is discussed briefly in Section 3.5.

The complete description of *MongoRaftReconfig* is left to the full version of the paper [28]. The pseudocode presented in Algorithm 1 describes the reconfiguration specific behaviors of *MongoRaftReconfig*, which are the novel aspects of the protocol and the contributions of this paper.



### 3.1 Overview and Intuition

Dynamic reconfiguration allows the set of servers operating as part of a replica set to be modified while maintaining the core safety guarantees of the replication protocol. Many consensus based replication protocols [29, 14, 22] utilize the main operation log (the *oplog*, in MongoDB) to manage configuration changes by writing special reconfiguration log entries. The *MongoRaftReconfig* protocol instead decouples configuration updates from the main operation log by managing the configuration state of a replica set in a separate, logless replicated state machine, which we refer to as the *config state machine*. The config state machine is maintained alongside the oplog, and manages the configuration state used by the overall protocol.

In order to ensure safe reconfiguration, *MongoRaftReconfig* imposes specific restrictions on how reconfiguration operations are allowed to update the configuration state of the replica set. First, it imposes a *quorum overlap* condition on any reconfiguration from  $C$  to  $C'$ , which is an approach adopted from the Raft single server reconfiguration algorithm. This ensures that all quorums of two adjacent configurations overlap with each other, and so can safely operate concurrently. In order to allow the system to pass through many configurations over time, though, *MongoRaftReconfig* imposes additional restrictions which address two essential aspects required for safe dynamic reconfiguration: (1) *deactivation* of old configurations and (2) *state transfer* from old configurations to new configurations. Essentially, it must ensure that old configurations, which may not overlap with newer configurations, are appropriately prevented from executing disruptive operations (e.g. electing a primary or committing a write), and it must also ensure that relevant protocol state from old configurations is properly transferred to newer configurations before they become active. The details of these restrictions and their safety implications are discussed further in Section 3.3.

In the remainder of this section we give an overview of the behaviors of *MongoRaftReconfig*, along with a pseudocode description of the protocol. We discuss its correctness in more depth in Section 4.

### 3.2 High Level Protocol Behavior

At a high level, dynamic reconfiguration in *MongoRaftReconfig* consists of two main aspects: (1) updating the current configuration and (2) propagating new configurations between servers. Configurations also have an impact on election behavior which we discuss below, in Section 3.4. Formally, a *configuration* is defined as a tuple  $(m, v, t)$ , where  $m \in 2^{Server}$  is a member set,  $v \in \mathbb{N}$  is a numeric configuration *version*, and  $t \in \mathbb{N}$  is the numeric *term* of the configuration. For convenience, we refer to the elements of a configuration tuple  $C = (m, v, t)$  as, respectively,  $C.m$ ,  $C.v$  and  $C.t$ . Each server of a replica set maintains a single, durable configuration, and it is assumed that, initially, all nodes begin with a common configuration,  $(m_{init}, 1, 0)$ , where  $m_{init} \in (2^{Server} \setminus \emptyset)$ .

To update the current configuration of a replica set, a client issues a *reconfiguration* command to a primary server with a new, desired configuration,  $C'$ . Reconfigurations can only be executed on primary servers, and they update the primary's current local configuration  $C$  to the specified configuration  $C'$ . The version of the new configuration,  $C'.v$ , must be greater than the version of the primary's current configuration,  $C.v$ , and the term of  $C'$  is set equal to the current term of the primary processing the reconfiguration. After a reconfiguration has occurred on a primary, the updated configuration needs to be communicated to other servers in the replica set. This is achieved in a simple, gossip like manner. Secondaries receive information about the configurations of other servers via periodic heartbeats. They need to

■ **Algorithm 1** Pseudocode description of *MongoRaftReconfig* reconfiguration specific behavior.

### Definitions

$$C_{(i)} \triangleq (\text{config}[i], \text{configVersion}[i], \text{configTerm}[i])$$

$$C_i > C_j \triangleq (C_i.t > C_j.t) \vee (C_i.t = C_j.t \wedge C_i.v > C_j.v)$$

$$C_i \geq C_j \triangleq (C_i > C_j) \vee ((C_i.v, C_i.t) = (C_j.v, C_j.t))$$

$$Q1(i) \triangleq \exists Q \in \text{Quorums}(\text{config}[i]) : \forall j \in Q : (C_{(j)}.v, C_{(j)}.t) = (C_{(i)}.v, C_{(i)}.t) \triangleright \text{Config Quorum Check}$$

$$Q2(i) \triangleq \exists Q \in \text{Quorums}(\text{config}[i]) : \forall j \in Q : \text{term}[j] = \text{term}[i] \triangleright \text{Term Quorum Check}$$

$$P1(i) \triangleq \exists Q \in \text{Quorums}(\text{config}[i]) : \text{all entries committed in terms } \leq \text{term}[i] \text{ are committed in } Q$$

<pre> 1: <b>State and Initialization</b> 2: Let <math>m_{init} \in 2^{Server} \setminus \emptyset</math> 3: <math>\forall i \in Server :</math> 4:   <math>\text{term}[i] \in \mathbb{N}</math>, initially 0 5:   <math>\text{state}[i] \in \{Pri., Sec.\}</math>, initially <i>Secondary</i> 6:   <math>\text{config}[i] \in 2^{Server}</math>, initially <math>m_{init}</math> 7:   <math>\text{configVersion}[i] \in \mathbb{N}</math>, initially 1 8:   <math>\text{configTerm}[i] \in \mathbb{N}</math>, initially 0 9: 10: <b>Actions</b> 11: <b>action:</b> RECONFIG(<math>i, m_{new}</math>) 12:   <b>require</b> <math>\text{state}[i] = Primary</math> 13:   <b>require</b> <math>Q1(i) \wedge Q2(i) \wedge P1(i)</math> 14:   <b>require</b> <math>QuorumsOverlap(\text{config}[i], m_{new})</math> 15:   <math>\text{config}[i] \leftarrow m_{new}</math> 16:   <math>\text{configVersion}[i] \leftarrow \text{configVersion}[i] + 1</math> 17: 18: <b>action:</b> SENDCONFIG(<math>i, j</math>) </pre>	<pre> 19:   <b>require</b> <math>\text{state}[j] = Secondary</math> 20:   <b>require</b> <math>C_{(i)} &gt; C_{(j)}</math> 21:   <math>C_{(j)} \leftarrow C_{(i)}</math> 22: 23: <b>action:</b> BECOMELEADER(<math>i, Q</math>) 24:   <b>require</b> <math>Q \in \text{Quorums}(\text{config}[i])</math> 25:   <b>require</b> <math>i \in Q</math> 26:   <b>require</b> <math>\forall v \in Q : C_{(i)} \geq C_{(v)}</math> 27:   <b>require</b> <math>\forall v \in Q : \text{term}[i] + 1 &gt; \text{term}[v]</math> 28:   <math>\text{state}[i] \leftarrow Primary</math> 29:   <math>\text{state}[j] \leftarrow Secondary, \forall j \in (Q \setminus \{i\})</math> 30:   <math>\text{term}[j] \leftarrow \text{term}[i] + 1, \forall j \in Q</math> 31:   <math>\text{configTerm}[i] \leftarrow \text{term}[i] + 1</math> 32: 33: <b>action:</b> UPDATETERMS(<math>i, j</math>) 34:   <b>require</b> <math>\text{term}[i] &gt; \text{term}[j]</math> 35:   <math>\text{state}[j] \leftarrow Secondary</math> 36:   <math>\text{term}[j] \leftarrow \text{term}[i]</math> </pre>
---	--

have some mechanism, however, for determining whether one configuration is newer than another. This is achieved by totally ordering configurations by their  $(version, term)$  pair, where term is compared first, followed by version. If configuration  $C_j$  compares as greater than configuration  $C_i$  based on this ordering, we say that  $C_j$  is *newer* than  $C_i$ . A secondary can update its configuration to any that is newer than its current configuration. If it learns that another server has a newer configuration, it will fetch that server's configuration, verify that it is still newer than its own upon receipt, and install it locally.

The above provides a basic outline of how reconfigurations occur and how configurations are propagated between servers in *MongoRaftReconfig*. The pseudocode given in Algorithm 1 gives a more abstract and precise description of these behaviors. Note that, in order for the protocol to operate safely, there are several additional restrictions that are imposed on both reconfigurations and elections, which we discuss in more detail below, in Sections 3.3 and 3.4.

### 3.3 Safety Restrictions on Reconfigurations

In *MongoStaticRaft*, which does not allow reconfiguration, the safety of the protocol depends on the fact that the *quorum overlap* condition is satisfied for the member sets of any two configurations. This holds since there is a single, uniform configuration that is never modified. For any pair of arbitrary configurations, however, their member sets may not satisfy this property. So, in order for *MongoRaftReconfig* to operate safely, extra restrictions are needed on how nodes are allowed to move between configurations. First, any reconfiguration that moves from  $C$  to  $C'$  is required to satisfy the quorum overlap condition i.e.  $QuorumsOverlap(C.m, C'.m)$ . This restriction is discussed in Raft's approach to reconfiguration [19], and is adopted by *MongoRaftReconfig*. Even if quorum overlap is ensured

between any two adjacent configurations, it may not be ensured between all configurations that the system passes through over time. So, there are additional preconditions that must be satisfied before a primary server in term  $T$  can execute a reconfiguration out of its current configuration  $C$ :

- Q1. Config Quorum Check:* There must be a quorum of servers in  $C.m$  that are currently in configuration  $C$ .
- Q2. Term Quorum Check:* There must be a quorum of servers in  $C.m$  that are currently in term  $T$ .
- P1. Oplog Commitment:* All oplog entries committed in terms  $\leq T$  must be committed on some quorum of servers in  $C.m$ .

The above preconditions are stated in Algorithm 1 as  $Q1(i)$ ,  $Q2(i)$ , and  $P1(i)$ , and they collectively enforce two fundamental requirements needed for safe reconfiguration: *deactivation* of old configurations and *state transfer* from old configurations to new configurations. Q1, when coupled with the election restrictions discussed in Section 3.4, achieves deactivation by ensuring that configurations earlier than  $C$  can no longer elect a primary. Q2 ensures that term information from older configurations is correctly propagated to newer configurations, while P1 ensures that previously committed oplog entries are properly transferred to the current configuration, ensuring that any primary in a current or later configuration will contain these entries.

### 3.4 Configurations and Elections

When a node runs for election in *MongoStaticRaft*, it must ensure its log is appropriately up to date and that it can garner a quorum of votes in its term. In *MongoRaftReconfig*, there is an additional restriction on voting behavior that depends on configuration ordering. If a replica set server is a candidate for election in configuration  $C_i$ , then a prospective voter in configuration  $C_j$  may only cast a vote for the candidate if  $C_i$  is newer than or equal to  $C_j$ . Furthermore, when a node wins an election, it must update its current configuration with its new term before it is allowed to execute subsequent reconfigurations. That is, if a node with current configuration  $(m, v, t)$  wins election in term  $t'$ , it will update its configuration to  $(m, v, t')$  before allowing any reconfigurations to be processed. This behavior is necessary to appropriately deactivate concurrent reconfigurations that may occur on primaries in a different term. This configuration re-writing behavior is analogous to the write in Raft's corrected membership change protocol proposed in [20].

### 3.5 Formal Specification

The complete, formal description of *MongoRaftReconfig* is given in the TLA+ specification in the supplementary materials [26]. Note that TLA+ does not impose an underlying system or communication model (e.g. message passing, shared memory), which allows one to write specifications at a wide range of abstraction levels. Our specifications are written at a deliberately high level of abstraction, ignoring some lower level details of the protocol and system model. In practice, we have found the abstraction level of our specifications most useful for understanding and communicating the essential behaviors and safety characteristics of the protocol, while also serving to make automated verification via model checking more feasible, which is discussed further in Section 4.4.

## 4 Correctness

In this section we present a brief outline of our safety proof for *MongoRaftReconfig*. We do not address liveness properties in this work. The full proof is left to [28].

The key, high level safety property of *MongoRaftReconfig* that we establish in this paper is *LeaderCompleteness*, which is a fundamental safety property of both standard Raft and *MongoStaticRaft*, and is stated below as Theorem 2. This property states that if a log entry has been committed in term  $T$ , then it must be present in the logs of all primary servers in terms  $> T$ . Essentially, it ensures that writes committed by some primary will be permanently durable in the replica set. Below we give a high level, intuitive outline of the proof.

### 4.1 Overview

Conceptually, *MongoRaftReconfig* can be viewed as an extension of the *MongoStaticRaft* replication protocol that allows for dynamic reconfiguration. *MongoRaftReconfig*, however, violates the property that all quorums of any two configurations overlap, which *MongoStaticRaft* relies on for safety. It is therefore necessary to examine how *MongoRaftReconfig* operates safely even though it cannot rely on the quorum overlap property. In *MongoStaticRaft*, there are two key aspects of protocol behavior that depend on quorum overlap: (1) *elections* of primary servers and (2) *commitment* of log entries. Elections must ensure that there is at most one unique primary per term, referred to as the *ElectionSafety* property. Additionally, if a log entry is committed in a given term, it must be present in the logs of all primary servers in higher terms, referred to as the *LeaderCompleteness* property. Both of these safety properties must be upheld in *MongoRaftReconfig*.

*LeaderCompleteness* is the essential, high level safety property that we must establish for *MongoRaftReconfig*. *ElectionSafety* is a key, auxiliary lemma that is required in order to show *LeaderCompleteness*. So, this guides the general structure of our proof. Section 4.2 presents an intuitive outline of the *ElectionSafety* proof, followed by a similar discussion of *LeaderCompleteness* in Section 4.3. The full proofs are left to [28].

### 4.2 Election Safety

In *MongoStaticRaft*, if an election has occurred in term  $T$  it ensures that some quorum of servers have terms  $\geq T$ . This prevents any future candidate from being elected in term  $T$ , since the quorum required for any future election will contain at least one of these servers, preventing a successful election in term  $T$ . This property, referred to as *ElectionSafety*, is stated below as Lemma 1.

► **Lemma 1** (Election Safety). *For all  $s, t \in Server$  such that  $s \neq t$ , it is not the case that both  $s$  and  $t$  are primary and have the same term.*

$\forall s, t \in Server :$

$$(state[s] = Primary \wedge state[t] = Primary \wedge term[s] = term[t]) \Rightarrow (s = t)$$

In *MongoRaftReconfig*, ensuring that a quorum of nodes have terms  $\geq T$  after an election in term  $T$  is not sufficient to ensure that *ElectionSafety* holds, since there is no guarantee that all quorums of future configurations will overlap with those of past configurations. To address this, *MongoRaftReconfig* must appropriately *deactivate* past configurations before creating new configurations. Conceptually, configurations in the protocol can be considered as either or

*active* or *deactivated*, the former being any configuration that is not deactivated. Deactivated configurations cannot elect a new leader or execute a reconfiguration. *MongoRaftReconfig* ensures proper deactivation of configurations by upholding an invariant that the quorums of all active configurations overlap with each other. In addition to deactivation of configurations, *MongoRaftReconfig* must also ensure that term information from one configuration is properly transferred to subsequent configurations, so that later configurations know about elections that occurred in earlier configurations. For example, if an election occurred in term  $T$  in configuration  $C$ , even if  $C$  is deactivated by the time  $C'$  is created, the protocol must also ensure that  $C'$  is “aware” of the fact that an election in  $T$  occurred in  $C$ . *MongoRaftReconfig* ensures this by upholding an additional invariant stating that the quorums of all active configurations overlap with some server in term  $\geq T$ , for any past election that occurred in term  $T$ .

Collectively, the two above invariants are the essential properties for understanding how the *ElectionSafety* property is upheld in *MongoRaftReconfig*. The formal statement of these invariants and the complete proof is left to [28]. In the following section, we briefly discuss the *LeaderCompleteness* property and its proof, which relies on the *ElectionSafety* property.

### 4.3 Leader Completeness

*LeaderCompleteness* is the key high level safety property of *MongoRaftReconfig*. It ensures that if a log entry is committed in term  $T$ , then it is present in the logs of all leaders in terms  $> T$ . Essentially, it ensures that committed log entries are durable in a replica set. It is stated below as Theorem 2, where *committed*  $\in \mathbb{N} \times \mathbb{N}$  refers to the set of committed log entries as  $(index, term)$  pairs, and  $InLog(i, t, s)$  is a predicate determining whether a log entry  $(i, t)$  is contained in the log of server  $s$ .

► **Theorem 2 (Leader Completeness).** *If a log entry is committed in term  $T$ , then it is present in the log of any leader in term  $T' > T$ .*

$$\forall s \in Server : \forall (cindex, cterm) \in committed : \\ (state[s] = Primary \wedge cterm < term[s]) \Rightarrow InLog(cindex, cterm, s) \quad (3)$$

In *MongoStaticRaft*, *LeaderCompleteness* is ensured due to the overlap between quorums used for commitment of a write and quorums used for the election of a primary. In *MongoRaftReconfig*, this does not hold, so the protocol instead upholds a more general invariant, stating that, for all committed entries  $E$ , the quorums of all active configurations overlap with some server that contains  $E$  in its log. *MongoRaftReconfig* also ensures that newer configurations appropriately disable commitment of log entries in older terms. We defer the statement of these invariants and the complete proof of Theorem 2 and its supporting lemmas to [28].

### 4.4 Model Checking

In addition to the safety proof outlined above, we used TLC [33], an explicit state model checker for TLA+ specifications, to gain additional confidence in the safety of the protocol. We consider it important to augment the human reasoning process for protocols like this with some type of machine based verification, even if the verification is incomplete, since it is easy for humans to make subtle errors in reasoning when considering distributed protocols of this nature.

We verified fixed, finite instances of *MongoRaftReconfig* to provide a sound guarantee of protocol correctness for given parameters. *MongoRaftReconfig* is an infinite state protocol, so verification via explicit state model checking is, necessarily, incomplete. That is, it does not establish correctness of the protocol for an unbounded number of servers or system parameters. It does, however, provide a strong initial level of confidence that the protocol is safe. A goal for future work is to develop a complete, machine checked safety proof using the TLA+ proof system [5].

#### 4.4.1 Methodology and Results

Formally, *MongoRaftReconfig* behaves as an extension of *MongoStaticRaft* that allows for dynamic reconfiguration. Thus, it can be viewed as a composition of two distinct subprotocols: one for managing the oplog, and one for managing configuration state. The oplog is maintained by *MongoStaticRaft*, and configurations are maintained by a protocol we refer to below as *MongoLoglessDynamicRaft*, which implements the logless replicated state machine that manages the configuration state of the replica set. Algorithm 1 summarizes the behaviors of *MongoLoglessDynamicRaft*. This compositional approach to describing *MongoRaftReconfig* is formalized in our TLA+ specification which can be found in the supplementary materials [26]. Our verification efforts centered on checking the two key safety properties discussed in the above sections, *ElectionSafety* and *LeaderCompleteness*. We summarize the results below, leaving the full details to [28].

**Checking Leader Completeness.** We were able to successfully verify the *LeaderCompleteness* property on a finite instance of *MongoRaftReconfig* with 4 servers, logs of maximum length 2, maximum configuration versions of 3, and maximum server terms of 3. That is, we manually imposed a constraint preventing the model checker from exploring any states exceeding these finite bounds. Model checking this instance generated approximately 345 million distinct protocol states and took approximately 8 hours to complete with 20 TLC worker threads on a 48-core, 2.30GHz Intel Xeon Gold 5118 CPU.

**Checking Election Safety.** As evidenced by the above metrics, it was difficult to scale verification of the *LeaderCompleteness* property to much larger system parameters. So, to provide additional confidence, we checked the *ElectionSafety* property on the *MongoLoglessDynamicRaft* protocol in isolation, which allowed us to verify instances with significantly larger parameters. Due to the compositional structure of *MongoRaftReconfig*, verifying that the *ElectionSafety* property holds on *MongoLoglessDynamicRaft* is sufficient to ensure that it holds in *MongoRaftReconfig*. Intuitively, the additional preconditions imposed by *MongoRaftReconfig* only *restrict* the behaviors of *MongoLoglessDynamicRaft*, but do not augment them. We formalize and prove this fact via a refinement based argument, whose details are left to [28]. This allows us to assume our verification efforts for *MongoLoglessDynamicRaft* hold in *MongoRaftReconfig*, providing stronger confidence in the correctness of the overall protocol.

We successfully verified the *ElectionSafety* property on a finite instance of *MongoLoglessDynamicRaft* with 5 servers, maximum configuration versions of 4, and maximum terms of 4. Model checking this instance generated approximately 812 million distinct states and took around 19.5 hours to complete with 20 TLC worker threads on a 48-core, 2.30GHz Intel Xeon Gold 5118 CPU. The ability to check these considerably larger parameter values in only several extra hours of wall clock time demonstrates the effectiveness of this compositional model checking approach, helping us mitigate state space explosion [6].

## 5 Conceptual Insights

*MongoRaftReconfig* can be viewed as a generalization and optimization of the standard Raft reconfiguration protocol. To explain the conceptual novelties of our protocol and how it relates to standard Raft, we discuss below the two primary aspects of the protocol which set it apart from Raft: (1) decoupling of the oplog and config state machine and (2) logless optimization of the config state machine. These are covered in Sections 5.1 and 5.2, respectively. Section 6 provides an experimental evaluation of how these novel aspects can provide performance benefits for reconfiguration, by allowing reconfigurations to bypass the main operation log in cases where it has become slow or stalled.

### 5.1 Decoupling Reconfigurations

In standard Raft, the main operation log is used for both normal operations and reconfiguration operations. This coupling between logs has the benefit of providing a single, unified data structure to manage system state, but it also imposes fundamental restrictions on the operation of the two logs. Most importantly, in order for a write to commit in one log, it must commit all previous writes in the other. For example, if a reconfiguration log entry  $C_j$  has been written at log index  $j$  on primary  $s$ , and there is a sequence of uncommitted log entries  $U = \langle i, i + 1, \dots, j - 1 \rangle$  in the log of  $s$ , in order for a reconfiguration from  $C_j$  to  $C_k$  to occur, all entries of  $U$  must become committed. This behavior, however, is stronger than necessary for safety i.e. it is not strictly necessary to commit these log entries before executing a reconfiguration. The only fundamental requirements are that previously committed log entries are committed by the rules of the current configuration, and that the current configuration has satisfied the necessary safety preconditions. Raft achieves this goal implicitly, but more conservatively than necessary, by committing the entry  $C_j$  and all entries behind it. This ensures that all previously committed log entries, in addition to the uncommitted operations  $U$ , are now committed in  $C_j$ , but it is not strictly necessary to pipeline a reconfiguration behind commitment of  $U$ . *MongoRaftReconfig* avoids this by separating the oplog and config state machine and their rules for commitment and reconfiguration, allowing reconfigurations to bypass the oplog if necessary. Section 6 examines this aspect of the protocol experimentally.

### 5.2 Logless Optimization

Decoupling the config state machine from the main operation log allows for an optimization that is enabled by the fact that reconfigurations are “update-only” operations on the replicated state machine. This means that it is sufficient to store only the latest version of the replicated state, since the latest version can be viewed as a “rolled-up” version of the entire (infinite) log. This logless optimization allows the configuration state machine to avoid complexities related to garbage collection of old log entries and it simplifies the mechanism for state propagation between servers. Normally, log entries are replicated incrementally, either one at a time, or in batches from one server to another. Additionally, servers may need to have an explicit procedure for deleting (i.e. rolling back) log entries that will never become committed. In the logless replicated state machine, all of these mechanisms can be combined into a single conceptual action, that atomically transfers the entire log of server  $s$  to another server  $t$ , if the log of  $s$  is newer, based on the index and term of its last entry. In *MongoRaftReconfig*, this is implemented by the *SendConfig* action, which transfers configuration state from one server to another.

## 6 Experimental Evaluation

In a healthy replica set, it is possible that a failure event causes some subset of replica set servers to degrade in performance, causing the main oplog replication channel to become lagged or stall entirely. If this occurs on a majority of nodes, then the replica set will be prevented from committing new writes until the performance degradation is resolved. For example, consider a 3 node replica set consisting of nodes  $\{n0, n1, n2\}$ , where nodes  $n1$  and  $n2$  suddenly become slow or stall replication. An operator or failure detection module may want to reconfigure these nodes out of the set and add in two new, healthy nodes,  $n3$  and  $n4$ , so that the system can return to a healthy operational state. This requires a series of two reconfigurations, one to add  $n3$  and one to add  $n4$ . In standard Raft, this would require the ability to commit at least one reconfiguration oplog entry with one of the degraded nodes ( $n1$  or  $n2$ ). This prevents such a reconfiguration until the degradation is resolved. In *MongoRaftReconfig*, reconfigurations bypass the oplog replication channel, committing without the need to commit writes in the oplog. This allows *MongoRaftReconfig* to successfully reconfigure the system in such a degraded state, restoring oplog write availability by removing the failed nodes and adding in new, healthy nodes.

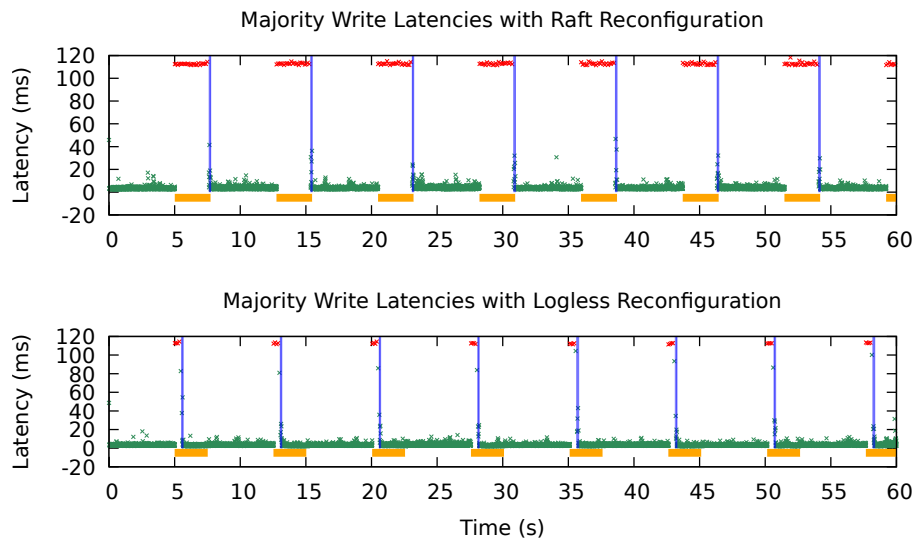
Note that if a replica set server experiences a period of degradation (e.g. a slow disk), both the oplog and reconfiguration channels will be affected, which would seem to nullify the benefits of decoupling the reconfiguration and oplog replication channels. In practice, however, the operations handled by the oplog are likely orders of magnitude more resource intensive than reconfigurations, which typically involve writing a negligible amount of data. So, even on a degraded server, reconfigurations should be able to complete successfully when more intensive oplog operations become prohibitively slow, since the resource requirements of reconfigurations are extremely lightweight.

### 6.1 Experiment Setup and Operation

To demonstrate the benefits of *MongoRaftReconfig* in this type of scenario, we designed an experiment to measure how quickly a replica set can reconfigure in new nodes to restore majority write availability when it faces periodic phases of degradation. For comparison, we implemented a simulated version of the Raft reconfiguration algorithm in MongoDB by having reconfigurations write a no-op oplog entry and requiring it to become committed before the reconfiguration can complete [25]. Our experiment initiates a 5 node replica set with servers we refer to as  $\{n0, n1, n2, n3, n4\}$ . We run the server processes co-located on a single Amazon EC2 t2.xlarge instance with 4 vCPU cores, 16GB memory, and a 100GB EBS disk volume, running Ubuntu 20.04. Co-location of the server processes is acceptable since the workload of the experiment does not saturate any resource (e.g. CPU, disk) of the machine. The servers run MongoDB version v4.4-39f10d with a patch to fix a minor bug [18] that prevents optimal configuration propagation speed in some cases.

Initially,  $\{n0, n1, n2\}$  are voting servers and  $\{n3, n4\}$  are non voting. In a MongoDB replica set, a server can be assigned either 0 or 1 votes. A non-voting server has zero votes and it does not contribute to a commit majority i.e. it is not considered as a member of the consensus group. Our experiment has a single writer thread that continuously inserts small documents into a collection with write concern *majority*, with a write concern timeout of 100 milliseconds. There is a concurrent fault injector thread that periodically simulates a degradation of performance on two secondary nodes by temporarily pausing oplog replication on those nodes. This thread alternates between *steady* periods and *degraded* periods of time, starting out in *steady* mode, where all nodes are operating normally. It runs for 5 seconds in





■ **Figure 1** Latency of majority writes in the face of node degradation and reconfiguration to recover. Red points indicate writes that timed out i.e. failed to commit. Orange horizontal bars indicate intervals of time where system entered a *degraded* mode. Thin, vertical blue bars indicate successful completion of reconfiguration events.

*steady* mode, then transitions to *degraded* mode for 2.5 seconds, before transitioning back to *steady* mode and repeating this cycle. When the fault injector enters *degraded* mode, the main test thread simulates a “fault detection” scenario (assuming some external module detected the performance degradation) by sleeping for 500 milliseconds, and then starting a series of reconfigurations to add two new, healthy secondaries and remove the two degraded secondaries. Over the course of the experiment, which has a 1 minute duration, we measure the latency of each operation executed by the writer thread. These latencies are depicted in the graphs of Figure 1. Red points indicate writes that failed to commit i.e. that timed out at 100 milliseconds. The successful completion of reconfigurations are depicted with vertical blue bars. It can be seen how, when a period of degradation begins, the logless reconfiguration protocol is able to complete a series of reconfigurations quickly to get the system back to a healthy state, where writes are able to commit again and latencies drop back to their normal levels. In the case of Raft reconfiguration, writes continue failing until the period of degradation ends, since the reconfigurations to add in new healthy nodes cannot complete.

## 7 Related Work

Dynamic reconfiguration in consensus based systems has been explored from a variety of perspectives for Paxos based systems. In Lamport’s presentation of Paxos [13], he suggests using a fixed parameter  $\alpha$  such that the configuration for a consensus instance  $i$  is governed by the configuration at instance  $i - \alpha$ . This restricts the number of commands that can be executed until the new configuration becomes committed, since the system cannot execute instance  $i$  until it knows what configuration to use, potentially causing availability issues if reconfigurations are slow to commit. Stoppable Paxos [14] was an alternative method later proposed where a Paxos system can be reconfigured by stopping the current state machine

and starting up a new instance of the state machine with a potentially different configuration. This “stop-the-world” approach can hurt availability of the system while a reconfiguration is being processed. Vertical Paxos allows a Paxos state machine to be reconfigured in the middle of reaching agreement, but it assumes the existence of an external configuration master [15]. In [4], the authors describe the Paxos implementation underlying Google’s Chubby lock service, but do not include details of their approach to dynamic reconfiguration, stating that “While group membership with the core Paxos algorithm is straightforward, the exact details are non-trivial when we introduce Multi-Paxos...”. They remark that the details, though minor, are “...subtle and beyond the scope of this paper”.

The Raft consensus protocol, published in 2014 by Ongaro and Ousterhout [22], presented two methods for dynamic membership changes: single server membership change and joint consensus. A correctness proof of the core Raft protocol, excluding dynamic reconfiguration, was included in Ongaro’s PhD dissertation [19]. Formal verification of Raft’s linearizability guarantees was later completed in Verdi [32], a framework for verifying distributed systems in the Coq proof assistant [3], but formalization of dynamic reconfiguration was not included. In 2015, after Raft’s initial publication, a safety bug in the single server reconfiguration approach was found by Amos and Zhang [2], at the time PhD students working on a project to formalize parts of Raft’s original reconfiguration algorithm. A fix was proposed shortly after by Ongaro [20], but the project was never extended to include the fixed version of the protocol. The Zab replication protocol, implemented in Apache Zookeeper [29], also includes a dynamic reconfiguration approach for primary-backup clusters that is similar in nature to Raft’s joint consensus approach.

The concept of decoupling reconfiguration from the main data replication channel has previously appeared in other replication systems, but none that integrate with a Raft-based system. RAMBO [8], an algorithm for implementing a distributed shared memory service, implements a dynamic reconfiguration module that is loosely coupled with the main read-write functionality. Additionally, Matchmaker Paxos [31] is a more recent approach for reconfiguration in Paxos based protocols that adds dedicated nodes for managing reconfigurations, which decouples reconfiguration from the main processing path, preventing performance degradation during configuration changes. There has also been prior work on reconfiguration using weaker models than consensus [10], and approaches to logless implementations of Paxos based replicated state machine protocols [23], which bear conceptual similarities to our logless protocol for managing configuration state. Similarly, [11] presents an approach to asynchronous reconfiguration under a Byzantine fault model that avoids reaching consensus on configurations by utilizing a lattice agreement abstraction.

## 8 Conclusions and Future Work

In this paper we presented *MongoRaftReconfig*, a novel, logless dynamic reconfiguration protocol that improves upon and generalizes the single server reconfiguration protocol of standard Raft by decoupling the main operation and reconfiguration logs. Although *MongoRaftReconfig* was developed for and presented in the context of the MongoDB system, the ideas and underlying protocol generalize to other Raft-based replication protocols that require dynamic reconfiguration. Goals for future work include development of a machine checked safety proof of the protocol’s correctness with help of the TLA+ proof system [5], in addition to running more in depth experiments to evaluate how *MongoRaftReconfig* behaves under more varied workloads.

---

**References**

---

- 1 Marcos Aguilera, Idit Keidar, Dahlia Malkhi, Jean-Philippe Martin, and Alexander Shraer. Reconfiguring Replicated Atomic Storage: A Tutorial. *Bulletin of the European Association for Theoretical Computer Science EATCS*, 2010.
- 2 Brandon Amos and Huanchen Zhang. Specifying and proving cluster membership for the Raft distributed consensus algorithm, 2015. URL: <https://www.cs.cmu.edu/~aplatzer/course/pls15/projects/bamos.pdf>.
- 3 Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.
- 4 Tushar D Chandra, Robert Griesemer, and Joshua Redstone. Paxos Made Live: An Engineering Perspective. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, pages 398–407, New York, NY, USA, 2007. Association for Computing Machinery. doi:10.1145/1281100.1281103.
- 5 Kaustuv Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. Verifying safety properties with the tla+ proof system. In *International Joint Conference on Automated Reasoning*, pages 142–148. Springer, 2010.
- 6 Edmund M Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. Model checking and the state explosion problem. In *LASER Summer School on Software Engineering*, pages 1–30. Springer, 2011.
- 7 James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012*, 2012. doi:10.1145/2518037.2491245.
- 8 Seth Gilbert, Nancy A. Lynch, and Alexander A. Shvartsman. Rambo: A robust, reconfigurable atomic memory service for dynamic networks. *Distributed Computing*, 2010. doi:10.1007/s00446-010-0117-1.
- 9 Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liqian Pei, and Xin Tang. TiDB: a Raft-based HTAP database. *Proceedings of the VLDB Endowment*, 2020. doi:10.14778/3415478.3415535.
- 10 Leander Jehl and Hein Meling. Asynchronous reconfiguration for Paxos state machines. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2014. doi:10.1007/978-3-642-45249-9\_8.
- 11 Petr Kuznetsov and Andrei Tonkikh. Asynchronous Reconfiguration with Byzantine Failures. In Hagit Attiya, editor, *34th International Symposium on Distributed Computing (DISC 2020)*, volume 179 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 27:1–27:17, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.DISC.2020.27.
- 12 Leslie Lamport. The Part-Time Parliament. *ACM Transactions on Computer Systems*, 1998. doi:10.1145/279227.279229.
- 13 Leslie Lamport. Paxos Made Simple. *ACM SIGACT News*, 2001. doi:10.1145/568425.568433.
- 14 Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Stoppable paxos. *TechReport, Microsoft Research*, 2008.
- 15 Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical Paxos and Primary-Backup Replication. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*, PODC '09, pages 312–313, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1582716.1582783.

## 26:16 Logless Dynamic Reconfiguration

- 16 Stephan Merz. *The Specification Language TLA+*, pages 401–451. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. doi:10.1007/978-3-540-74107-7\_8.
- 17 MongoDB Github Project, 2021. URL: <https://github.com/mongodb/mongo>.
- 18 MongoDB JIRA Ticket SERVER-46907, 2020. URL: <https://jira.mongodb.org/browse/SERVER-46907>.
- 19 Diego Ongaro. Consensus: Bridging Theory and Practice. *Doctoral thesis*, 2014.
- 20 Diego Ongaro. Bug in single-server membership changes, July 2015. URL: <https://groups.google.com/g/raft-dev/c/t4xj6dJTP6E/m/d2D9LrWRza8J>.
- 21 Diego Ongaro. The Raft Consensus Algorithm, 2021. URL: <https://raft.github.io/>.
- 22 Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference, USENIX ATC'14*, pages 305–320, USA, 2014. USENIX Association.
- 23 Denis Rystsov. CASPaxos: Replicated State Machines without logs, 2018. arXiv:1802.07000.
- 24 Fred B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys (CSUR)*, 1990. doi:10.1145/98163.98167.
- 25 William Schultz. MongoDB Experiment Source Code, September 2021. URL: <https://github.com/will162794/mongo/tree/2bb9f30da>.
- 26 William Schultz. MongoRaftReconfig TLA+ Specifications, November 2021. doi:10.5281/zenodo.5715511.
- 27 William Schultz, Tess Avitabile, and Alyson Cabral. Tunable Consistency in MongoDB. *Proc. VLDB Endow.*, 12(12):2071–2081, August 2019. doi:10.14778/3352063.3352125.
- 28 William Schultz, Siyuan Zhou, Ian Dardik, and Stavros Tripakis. Design and Analysis of a Logless Dynamic Reconfiguration Protocol. *arXiv preprint*, 2021. arXiv:2102.11960.
- 29 Alexander Shraer, Benjamin Reed, Dahlia Malkhi, and Flavio Junqueira. Dynamic reconfiguration of primary/backup clusters. In *Proceedings of the 2012 USENIX Annual Technical Conference, USENIX ATC 2012*, 2019.
- 30 Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. CockroachDB: The Resilient Geo-Distributed SQL Database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, SIGMOD '20*, pages 1493–1509, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3318464.3386134.
- 31 Michael Whittaker, Neil Giridharan, Adriana Szekeres, Joseph M Hellerstein, Heidi Howard, Faisal Nawab, and Ion Stoica. Matchmaker Paxos: A Reconfigurable Consensus Protocol [Technical Report], 2020. arXiv:2007.09468.
- 32 Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas Anderson. Planning for change in a formal verification of the raft consensus protocol. In *CPP 2016 - Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, co-located with POPL 2016*, 2016. doi:10.1145/2854065.2854081.
- 33 Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking TLA+ specifications. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 54–66. Springer, 1999.
- 34 Siyuan Zhou and Shuai Mu. Fault-Tolerant Replication with Pull-Based Consensus in MongoDB. In *NSDI*, pages 687–703, 2021.

# Optimal Good-Case Latency for Rotating Leader Synchronous BFT

Ittai Abraham ✉

VMware Research, Herzliya, Israel

Kartik Nayak ✉

Duke University, Durham, NC, USA

Nibesh Shrestha ✉

Rochester Institute of Technology, NY, USA

---

## Abstract

This paper explores the good-case latency of synchronous Byzantine Fault Tolerant (BFT) consensus protocols in the rotating leader setting. We first present a lower bound that relates the latency of a broadcast when the sender is honest and the latency of switching to the next sender. We then present a matching upper bound with a latency of  $2\Delta$  ( $\Delta$  is the pessimistic synchronous delay) with an optimistically responsive change to the next sender. The results imply that both our lower and upper bounds are tight. We implement and evaluate our protocol and show that our protocol obtains similar latency compared to state-of-the-art stable-leader protocol Sync HotStuff while allowing optimistically responsive leader rotation.

**2012 ACM Subject Classification** Security and privacy → Distributed systems security

**Keywords and phrases** Distributed Computing, Byzantine Fault Tolerance, Synchrony

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2021.27

**Related Version** *Previous Version*: <https://eprint.iacr.org/2021/1138.pdf>

## 1 Introduction

Byzantine fault tolerant (BFT) consensus protocols provide a consistent service despite some malicious and arbitrary process failures. These protocols have been particularly useful in developing infrastructures that span across multiple entities some of which may be incentivized to act maliciously. Thus BFT protocols have been widely adopted to build decentralized blockchain technologies that promise tamper-proof ledger services without a central authority. In accordance, this problem has been studied for over 40 years under various system models and assumptions [26, 2, 7, 22]. Among these, authenticated BFT protocols under *synchrony* assumption are particularly interesting as these protocols can tolerate one-half Byzantine failures [14, 15, 18] compared to partially synchronous or asynchronous protocols which tolerate only upto one-third Byzantine failures [13].

When consensus is to be achieved for a sequence of values, a particularly well-studied approach among BFT protocols is the *stable-leader* approach where a single replica takes lead in coordinating all participating replicas into reaching consensus [7]. Stable-leader BFT protocols usually provide better performance both in terms of throughput and latency as they avoid a *view-change* process to switch leaders which incurs additional cost. From a practical perspective, an important metric in these protocols is the good-case latency to achieve consensus. Informally, good-case latency of a BFT protocol is the time for all honest replicas to commit (over all executions and adversarial strategies), given an honest leader. A recent work [3] provides a complete categorization of the good-case latency of BFT protocols under different network settings and number of faults. In particular, assuming synchrony, they provide matching upper and lower bounds of  $\Delta + O(\delta)$  good-case latency where  $\Delta$  is the known pessimistic network delay and  $\delta$  is the unknown actual network delay.



© Ittai Abraham, Kartik Nayak, and Nibesh Shrestha;  
licensed under Creative Commons License CC-BY 4.0

25th International Conference on Principles of Distributed Systems (OPODIS 2021).

Editors: Quentin Bramas, Vincent Gramoli, and Alessia Milani; Article No. 27; pp. 27:1–27:19

Leibniz International Proceedings in Informatics



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

While the stable-leader approach is favored for their better performance, many applications require a democracy favoring policy where the participating replicas take turn in being leader to coordinate consensus decisions. We call this a *rotating-leader* approach. The rationale is to obtain better *fairness*, *censorship resistance*, and uniform distribution of work. Indeed, several recent protocols have been designed with this goal in mind [9, 10, 16, 17, 24, 8, 20]. In general, rotating-leader BFT protocols can broadly be classified into two categories. In the first kind, multiple leaders are selected to propose values concurrently in different slots wherein each leader proposes a value in their own instance in the *common log*. While the value proposed by an honest leader can be decided with a small constant latency in the good-case, the value proposed by a Byzantine leader can take a linear number of rounds before it gets finalized in the worst case. In many state machine replication applications, a value proposed in an instance is not useful until all prior instances in the log have been finalized. Thus, such protocols always have worst-case latency which is linear.

The second kind utilizes the “block chaining” paradigm where a leader proposes a value in the form of a *block* which explicitly extends a block  $B$  proposed by an earlier leader by including hash of previous block  $B$  called its *parent*. In this paradigm, when a block  $B$  is committed, all its ancestors are also committed and all blocks in the log up to block  $B$  can be safely used. In addition, when the leader of block  $B$  is honest, block  $B$  and all its ancestors can be committed with a small constant latency in the good-case. This is a natural approach adopted by many existing protocols [26, 2, 8, 17, 25]. In this work, we focus on such rotating-leader BFT protocols that utilize block chaining paradigm. Under this paradigm, existing rotating-leader BFT protocols such as PiLi [10], Dfinity [17, 1] and Streamlet [8] incur a latency of  $65\Delta$ ,  $17\Delta$  and  $12\Delta$  respectively to commit a single decision in the good-case. While the good-case latency captures the latency of a commit given an honest leader, a rotating-leader protocol keeps changing leaders through a view-change process each time. Thus, the overall latency of a rotating-leader protocol depends on the combination of good-case latency and the latency of the view-change. In this work, we initiate a study to optimize the combined latency for rotating-leader protocols. A natural approach to optimize the combined latency is to spend as little time during the view-change as possible. Ideally, we want to change leaders and have the new leader propose responsively in  $O(\delta)$  time compared to waiting  $\Omega(\Delta)$  delay during the view-change process. We refer to such property as *optimistically responsive* rotating leader chained protocol defined as follows:

► **Definition 1** (Optimistically Responsive Rotating Leader Chained Protocol, Informal). *We say that a rotating-leader chained protocol has optimistically responsive leader change if for any two consecutive honest leaders  $p_i$  and  $p_{i+1}$ ,  $p_{i+1}$  sends its input within  $O(\delta)$  time after receiving  $p_i$ 's input.*

Allowing consecutive leaders to propose responsively allows a protocol to progress at network speed when the leaders are honest. In the context of stable-leader approach, a recent work Sync HotStuff [2] allows a fixed leader to propose responsively. For the rotating-leader approach, few recent works [25, 21] offer solutions that allow consecutive leaders to propose responsively. However, these protocols require a special *optimistic condition* (where  $> 3n/4$  replicas are expected to be honest) to provide responsiveness. Moreover, none of the prior works formalize this notion to provide theoretical feasibility results w.r.t. the good-case latency for the rotating-leader approach.

To close this gap, our paper explores the optimal good-case latency for rotating-leader synchronous BFT protocols with responsive proposals. Specifically we ask,

*What is the optimal good-case latency of optimistically responsive rotating-leader chained consensus protocol?*

We answer this question by providing a lower bound relating the good-case latency for a single slot with that of the view-change. Interestingly, we observe that if the protocol performs view-change in  $O(\delta)$ , then the commit latency for a single slot cannot be  $< 2\Delta$ . We also provide a matching upper bound protocol with  $2\Delta + O(\delta)$  latency. In essence, our results are tight (ignoring  $O(\delta)$  terms). We also evaluate our upper bound protocol against state-of-the-art synchronous protocol and observe that our protocol provides similar latency metric compared to Sync HotStuff which is the state-of-the-art synchronous consensus protocol that follows stable-leader approach and significantly better compared to its rotating-leader counterpart.

**A lower bound on the good-case latency of rotating-leader consensus protocols with responsive proposals.** Our first result presents a lower bound on the good-case latency of a rotating-leader consensus protocols with responsive proposals. Specifically, we show the following:

► **Theorem 2** (Lower bound on the good-case latency of an optimistically responsive rotating-leader chained consensus protocol, Informal). *There exists an execution in an optimistically responsive rotating-leader chained consensus protocol in an unsynchronized start model tolerating  $n/3 \leq f < n/2$  faults where the following two conditions do not hold simultaneously even in executions where messages between honest parties arrive instantaneously and all parties start at time 0: (i) the good-case commit latency is less than  $2\Delta$ , and (ii) honest senders propose responsively in  $O(\delta)$  time after receiving a proposal from the previous honest sender.*

Intuitively, our lower bound states that if a protocol performs view-change in  $O(\delta)$  time, then the commit latency for a single slot has to be at least  $2\Delta$  time; otherwise the safety of a commit cannot be guaranteed. This lower bound applies to protocols in an *unsynchronized start model* where parties do not all start the protocol at the same time (explained later).

**Optimal responsively proposing rotating-leader protocol with  $2\Delta$ -synchronous latency.**

Our second result presents a matching upper bound protocol that achieves optimal commit latency of  $2\Delta$  with responsive proposals. Our upper bound result is presented in the form of state machine replication (SMR) protocol that decides on a sequence of values. In our SMR protocol, the leaders are rotated after each proposal. A sequence of honest leaders can propose within  $2\delta$  of previous proposal thus supporting responsive proposals. The good-case commit latency for a single slot is optimal, i.e.,  $2\Delta$ . Due to responsive leader rotation, our protocol changes leaders before prior proposed values have been committed. For a sequence of  $k$  honest leaders, our protocols can commit  $k$  values in  $2\Delta + O(k\delta)$  time.

**Implementation and evaluation.** We implement and evaluate the performance of our protocol and compare it with Sync HotStuff [2] which is the state-of-art synchronous protocol with stable-leader approach. In terms of latency, our protocol has similar latency profile compared to Sync HotStuff, i.e., like Sync HotStuff, our protocol commits  $k$  values in  $2\Delta + O(k\delta)$  time. We validate this in our evaluation by showing a latency comparable to Sync HotStuff. However, Sync HotStuff being a stable-leader protocol does not provide fairness and censorship resistance. To ensure fair comparison, we evaluate our protocol against Sync HotStuff with leader rotation where leaders are rotated after each proposal. Compared to this variation, we obtain *four times better latency* in all configurations.

## 2 Model and Definitions

We consider a system consisting of  $n$  replicas in a reliable, authenticated all-to-all network, where up to  $f < n/2$  replicas can be Byzantine faulty. The Byzantine replicas may behave arbitrarily. A replica that is not corrupted is considered to be honest and executes the protocol as specified.

Communication between honest replicas are synchronous. Thus, if a replica  $r$  sends a message  $x$  to another replica  $r'$  at time  $t$ ,  $r'$  receives the message by time  $t + \delta$  if  $r$  is honest. The delay parameter  $\delta$  is upper bounded by  $\Delta$ . The upper bound  $\Delta$  is known, but  $\delta$  is unknown to the system.  $\delta$  can be regarded as an actual delay in the real-world network. We assume all honest replicas have clocks moving at the same speed.

We make use of digital signatures and a public-key infrastructure (PKI) to validate messages and detect equivocation. Message  $x$  sent by a node  $p$  is digitally signed by  $p$ 's private key and is denoted by  $\langle x \rangle_p$ . In addition, we use  $H(x)$  to denote the invocation of the random oracle  $H$  on input  $x$ .

► **Definition 3** (Byzantine Fault-tolerant State Machine Replication [23]). *A Byzantine fault-tolerant state machine replication protocol commits client requests as a linearizable log to provide a consistent view of the log akin to a single non-faulty server, providing the following two guarantees. (i) **Safety**. Honest replicas do not commit different values at the same log position. (ii) **Liveness**. Each client request is eventually committed by all honest replicas.*

► **Definition 4** (Good-case Latency [3]). *A Byzantine broadcast (or Byzantine reliable broadcast) protocol has good-case latency of  $T$ , if all honest parties commit within time  $T$  since the broadcaster starts the protocol (over all executions and adversarial strategies), given the designated broadcaster is honest.*

**Chained BFT SMR.** As mentioned before, our work focuses on BFT protocols that utilize the block chaining paradigm where a leader proposes a value in the form of a block by explicitly extending a block  $B_k$  proposed by an earlier leader by including hash of previous block  $B_k$  called its *parent*. A block determines a unique hash chain for all previous blocks in the log. The first block in the chain is called the *genesis* block, and the distance from the genesis block to a block  $B$  in the chain is called the *height* of block  $B$ . A block  $B_k$  at height  $k$  has the format,  $B_k := (b_k, H(B_{k-1}))$  where  $b_k$  denotes the proposed payload at height  $k$ ,  $B_{k-1}$  is the block at height  $k - 1$  and  $H(B_{k-1})$  is the hash digest of  $B_{k-1}$ . A block  $B_k$  is said to extend a block  $B_h$  if  $B_k = B_h$  or  $B_k$  is a descendant of  $B_h$ . All the blocks in the chain from genesis block up to block  $B_{k-1}$  are the *ancestors* of block  $B_k$ . In this paradigm, when a block  $B_k$  is committed, all its ancestors are also committed.

With leaders proposing responsively in  $O(\delta)$  time after receiving proposal from a prior leader, an honest leader may propose a new block by extending on a block proposed by a prior Byzantine leader without detecting conflicting block proposals made by the Byzantine leader. When such inconsistencies are detected, a natural solution is to execute some fallback mechanism to collect blocks possibly committed by some honest replicas as the new leader could have proposed blocks extending conflicting proposals. However, such fallback mechanisms worsen the good-case latency of a protocol as it involves notifying of misbehavior by earlier Byzantine leaders and collecting possibly committed blocks to decide on a safe value to extend. In this work, we focus on responsively proposing rotating leader chained BFT SMR protocols that do not involve any fallback mechanism and always decide on blocks proposed by honest leaders irrespective of inconsistencies introduced by an earlier Byzantine leader. This allows our protocol to always commit with a small constant latency in the good-case when the leader is honest.



**Rotating sender chained reliable broadcast.** We formalize the rotating-leader chained BFT SMR protocols in the form of rotating sender chained reliable broadcast as follows. Our lower bound is presented in the form of rotating sender chained reliable broadcast.

► **Definition 5** (Rotating Sender Chained Reliable Broadcast). *In a rotating sender chained reliable broadcast, there is a sequence of designated senders  $p_1, \dots, p_k$  for  $k \geq 1$  sending messages such that when a sender  $p_i$  broadcasts  $B_x$  at some height  $x$  in the log, then the following conditions hold:*

1. (Correctness) *If some honest party  $q$  delivers message  $B_x$  at height  $x$  in the log, then eventually every honest party delivers  $B_x$  at height  $x$  in the log.*
2. (Validity) *If sender  $p_i$  is honest, then every honest party eventually delivers the input  $B_x$  that  $p_i$  broadcasts at height  $x$ .*
3. (Sequentiality) *If an honest sender  $p_j$  with  $j < i$  sends message  $B_{x'}$ , then the input  $B_x$  sent by an honest sender  $p_i$  must extend  $B_{x'}$ .*
4. (Extension) *When an honest party  $q$  delivers a message  $B_x$  at height  $x$  in the log, it delivers all the ancestors of message  $B_x$ .*

Next, we formally define *optimistically responsive rotating leader chained reliable broadcast* as follows.

► **Definition 6** (Optimistically Responsive Rotating Sender Chained Reliable Broadcast). *We say that a rotating sender chained reliable broadcast is optimistically responsive if for any two consecutive honest senders  $p_i$  and  $p_{i+1}$ ,  $p_{i+1}$  starts the broadcast in the sequence within  $O(\delta)$  time after receiving  $p_i$ 's input.*

### 3 A Lower Bound on the Good-Case Latency of Optimistically Responsive Rotating Sender Chained Reliable Broadcast

In this section, we provide a lower bound on the good-case latency of rotating leader protocols where the next leaders propose responsively without waiting  $\Omega(\Delta)$  time after receiving proposals from previous leaders. In particular, the lower bound captures the relationship between latency of the view-change and the good-case latency of a commit. Essentially, it says that if a consensus protocol tolerating  $n/3 \leq f < n/2$  Byzantine faults allows a new leader to propose responsively in  $\alpha$  time, the commit latency for a single slot cannot be less than  $2\Delta - \alpha$ . Thus, the sum of latencies has to be at least  $2\Delta$ .

**Unsynchronized starts.** Our lower bound assumes an unsynchronized start model [3, 25] where honest parties may start the protocol execution at different times decided by an adversary such that the following conditions hold: (i) each honest party starts the protocol at time  $\leq \Delta$  and (ii) an honest party starts the protocol before receiving a message from any other party. Such a model captures state machine replication protocols where replicas move to the next view/slot at different times. Byzantine parties, on the other hand, are assumed to start the protocol execution at time 0. The parties start the protocol with a fixed state independent of when the protocol execution started; in particular, they do not have access to the execution start time.

**Intuition.** Any Byzantine fault tolerant consensus protocol tolerating  $f$  Byzantine faults cannot wait to receive messages from more than  $n - f$  parties. For a synchronous consensus protocol tolerating  $f < n/2$ , an honest party can wait for messages from at most  $f + 1$  parties, assuming  $n = 2f + 1$  for simplicity sake. Consider a set  $P$  of  $f$  honest parties and a set  $Q$  of

$f$  honest parties. Suppose the current sender  $s_1$  is Byzantine and sends some value  $B_e$  which arrives at parties in  $P$  at some time 0. The Byzantine sender  $s_1$  also sends conflicting values  $B'_e$  to honest parties in  $Q$  which arrives only at time  $\Delta - \alpha$ . Observe that messages from a single Byzantine party  $s_1$  is sufficient for parties in  $P$  to form a quorum of  $f + 1$  messages. In addition, messages between parties in  $P$  and  $Q$  can be delayed by up to  $\Delta$  time. Thus, parties in  $P$  may not learn about conflicting value  $B'_e$  before  $2\Delta - \alpha$  time and will commit to value  $B_e$  before  $2\Delta - \alpha$  time where  $2\Delta - \alpha$  is the protocol commit latency.

The next sender  $s_2 \in Q$  receives value  $B'_e$  at time  $\Delta - \alpha$  and proposes value  $B_{e+1}$  by time  $\Delta$  with  $\alpha$  being the view-change latency. Note that sender  $s_2$  may not learn about value  $B_e$  until time  $\Delta$ . For sender  $s_2$ , the first sender  $s_1$  appears to be honest until time  $\Delta$ . Thus, by sequentiality property (refer Definition 5), honest sender  $s_2$  extends  $B'_e$  and proposes value  $B_{e+1}$  by time  $\Delta$ . Observe that due to  $\Delta$  delay between messages exchanged between parties in  $P$  and  $Q$ , parties in  $P$  does not receive either  $B'_e$  or  $B_{e+1}$  before time  $2\Delta - \alpha$  and hence cannot prevent parties in  $P$  to commit. Similarly, parties in  $Q$  does not receive value  $B_e$  before  $\Delta$  and does not prevent the next sender  $s_2$  from extending  $B'_e$  and proposing  $B_{e+1}$ . Since, our protocol always commits the value proposed by an honest sender, parties in  $Q$  eventually commit  $B_{e+1}$  and by extension property commit  $B_e$ . An important observation here is that if the commit latency were at least  $2\Delta - \alpha$ , parties in  $P$  would learn about value  $B'_e$  by time  $2\Delta - \alpha$  and would not commit. This implies the sum of responsive view-change latency and good-case latency has to be at least  $2\Delta$ .

► **Theorem 7** (Lower bound on the good-case latency of optimistically responsive rotating sender sequenced reliable broadcast). *For any  $\alpha < \Delta$ , there exists an execution in an optimistically responsive rotating sender chained reliable broadcast protocol in an unsynchronized start model tolerating  $n/3 \leq f < n/2$  faults where the following two conditions do not hold simultaneously even in executions where messages between honest parties arrive instantaneously and all parties start at time 0: (i) the good-case commit latency is less than  $2\Delta - \alpha$ , and (ii) honest senders broadcast responsively in at most  $\alpha$  time after receiving an input from the previous honest sender.*

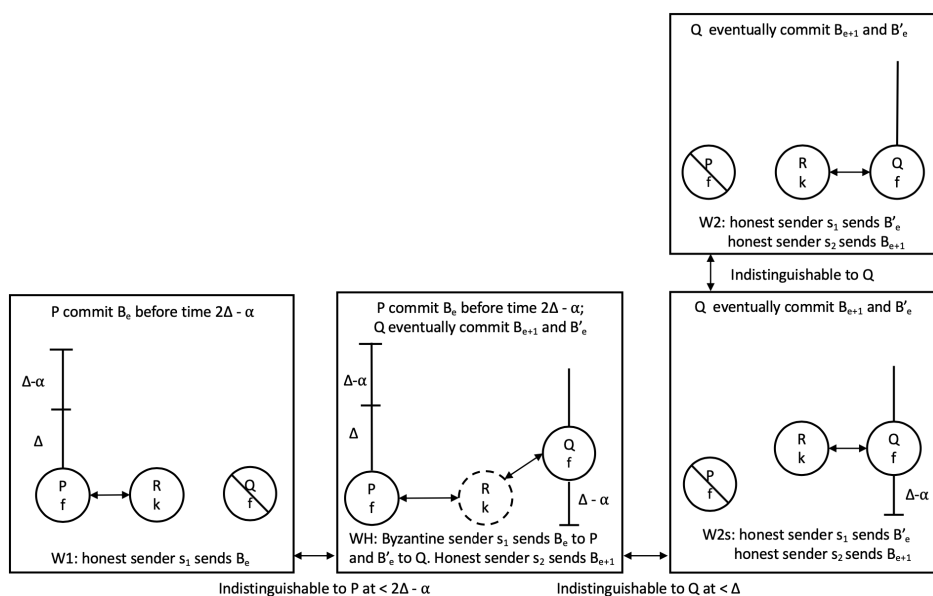
**Proof.** Suppose for the sake of contradiction, there exists such a protocol. We will show a sequence of worlds, and through an indistinguishability argument show a violation in the last execution. Consider the parties being partitioned into the following 3 sets: (i)  $P$ : a set of  $f$  parties, (ii)  $Q$ : a set of  $f$  parties which includes the second sender  $s_2$ , and (iii)  $R$ : a set of  $\max(1, n - 2f)$  parties which includes the first sender  $s_1$ .

**World 1 (W1).** *Setup.* Parties in  $P \cup R$  are honest while parties in  $Q$  are crashed. An honest sender  $s_1 \in R$  sends value  $B_e$  at some height  $e$  to all parties. Parties in  $P$  and  $R$  start at time 0.

*Message schedule.* Messages exchanged between parties in  $P$  and  $R$  arrive instantaneously. *Execution and views of honest parties.* This execution satisfies (i), so parties in  $P$  commit  $B_e$  before  $2\Delta - \alpha$  time.

**World 2 (W2).** *Setup.* Parties in  $Q \cup R$  are honest while parties in  $P$  have crashed. An honest sender  $s_1 \in R$  sends value  $B'_e$  at height  $e$  to all parties. Parties in  $Q$  and  $R$  start at time 0.

*Message schedule.* Messages exchanged between parties in  $Q$  and  $R$  arrive instantaneously. *Execution and views of honest parties.* The next sender  $s_2 \in Q$  receives  $B'_e$  at time 0. By sequentiality, the next sender  $s_2$  extends  $B'_e$  and sends value  $B_{e+1}$  by time  $\alpha$ . Since, sender  $s_2$  is honest, by validity, parties in  $Q \cup R$  eventually commit  $B_{e+1}$  at height  $e + 1$ . By extension, parties in  $Q \cup R$  commit  $B'_e$  at height  $e$ .



■ **Figure 1** Latency of Optimistically responsive rotating sender chained reliable broadcast. Dotted circles represent Byzantine parties. Crossed circles represent crashed parties. Value of  $k = n - 2f$ .

**World 2-shifted (W2s).** *Setup.* Parties in  $Q \cup R$  are honest while parties in  $P$  have crashed. An honest sender  $s_1 \in R$  sends value  $B'_e$  to all parties. All parties start at time  $\Delta - \alpha$ . *Message schedule.* Messages exchanged between parties in  $Q$  and  $R$  arrive instantaneously.

▷ **Claim 8.** Parties in  $Q \cup R$  cannot distinguish between World 2 and World 2-shifted.

*Proof.* Observe that in an unsynchronized start model, parties start with a fixed state independent of when the protocol execution started and parties do not have access to the starting time. Moreover, all parties in  $Q \cup R$  start at time  $\Delta - \alpha$  in World 2-shifted. Thus, for parties in  $Q \cup R$ , this execution is indistinguishable to World 2. ◁

*Execution and views of honest parties.* By Claim 8, parties in  $Q \cup R$  cannot distinguish between World 2 and World 2-shifted. Thus, the next sender  $s_2$  sends value  $B_{e+1}$  by time  $\Delta - \alpha + \alpha = \Delta$  by extending on  $B'_e$ . Since, this execution is identical to World 2, parties in  $Q \cup R$  eventually commit  $B_{e+1}$  at height  $e + 1$ .

**World Hybrid (WH).** *Setup.* Parties in  $R$  are Byzantine. All other parties are honest. The Byzantine sender  $s_1 \in R$  sends value  $B_e$  to parties in  $P$  and value  $B'_e$  to parties in  $Q$ . Parties in  $P$  start at time 0 while parties in  $Q$  start at time  $\Delta - \alpha$ .

*Message schedule.* Parties in  $P$  receive sender's value at time 0 while parties in  $Q$  receive sender's value at time  $\Delta - \alpha$ .

Parties in  $R$  perform a split-brain attack where they behave like World 1 towards parties in  $P$ , and behave like World 2-shifted towards parties in  $Q$ . We denote each brain of  $R$  as  $R1$  and  $R2$  such that  $R1$  only communicate with  $P$  and  $R2$  only communicate with  $Q$ .

Messages between parties in  $P$  and  $R1$  arrive instantaneously (like in World 1). Parties in  $R2$  sends messages to parties in  $Q$  only after time  $\Delta - \alpha$  and messages between  $Q$  and  $R2$  arrive instantaneously (like in World 2-shifted). Messages exchanged between parties in  $P$  and  $Q$  is delayed by  $\Delta$ .

▷ **Claim 9.** The parties in  $P$  cannot distinguish between World 1 and World Hybrid until  $2\Delta - \alpha$  time.

*Proof.* In World 1, parties in  $P$  start at time 0 and messages exchanged with parties in  $R$  arrive instantaneously. Since, parties in  $Q$  are crashed, parties in  $P$  do not receive any messages from parties in  $Q$ . Similarly, in World Hybrid, parties in  $P$  start at time 0 and messages exchanged with parties in  $R1$  arrive instantaneously where parties in  $R1$  behave identical to World 1 for parties in  $P$ . Moreover, since parties in  $Q$  start at time  $\Delta - \alpha$  and messages between parties in  $P$  and  $Q$  are delayed by  $\Delta$ , parties in  $P$  do not receive any messages from parties in  $Q$  until  $2\Delta - \alpha$  time. Thus, parties in  $P$  cannot distinguish between World 1 and World Hybrid until  $2\Delta - \alpha$  time.  $\triangleleft$

$\triangleright$  **Claim 10.** The parties in  $Q$  cannot distinguish between World 2-shifted and World Hybrid until time  $\Delta$ .

*Proof.* In World 2-shifted, parties in  $Q$  start at time  $\Delta - \alpha$  and messages exchanged with parties in  $R$  arrive instantaneously. Since, parties in  $P$  are crashed in World 2-shifted, parties in  $Q$  receive no messages from parties in  $P$ . Similarly, in World Hybrid, parties in  $Q$  start at time  $\Delta - \alpha$ . Parties in  $R2$  send messages only at time  $\Delta - \alpha$  and messages between  $R2$  and  $Q$  arrive instantaneously. Observe that parties in  $P$  start at time 0 and messages between  $P$  and  $R$  are delayed by  $\Delta$ . Thus, parties in  $Q$  receive no messages from parties in  $P$  until time  $\Delta$ . Thus, parties in  $Q$  cannot distinguish between World 2-shifted and World Hybrid until time  $\Delta$ .  $\triangleleft$

*Execution and views of honest parties.* By Claim 9, parties in  $P$  cannot distinguish between World 1 and World Hybrid until  $2\Delta - \alpha$  time. Thus, parties in  $P$  commit  $B_e$  by time  $2\Delta - \alpha$  at height  $e$ .

By Claim 10, parties in  $Q$  cannot distinguish between World 2-shifted and World Hybrid until  $\Delta$  time. Thus, the next sender  $s_2 \in Q$  sends input  $B_{e+1}$  that extends  $B'_e$  by time  $\Delta$ . By validity, parties in  $Q$  eventually commit  $B'_{e+1}$  at height  $e + 1$ . By extension, parties in  $Q$  also commit  $B'_e$  at height  $e$ . This violates correctness property between parties in  $P$  and  $Q$  at height  $e$ . A contradiction.  $\blacktriangleleft$

## 4 Optimal Rotating-Leader BFT SMR with $2\Delta$ -synchronous Latency

In this section, we present a rotating-leader chained BFT SMR protocol with optimal  $2\Delta$ -synchronous latency that allows *responsive* leader rotation while tolerating  $f < n/2$  Byzantine faults. Prior synchronous protocols such as Sync HotStuff follow stable-leader paradigm and can make progress at network speed in the steady-state i.e., it can commit  $k$  proposals in  $2\Delta + O(k\delta)$  where  $2\Delta$  is the commit latency for a single proposal. However, when the protocol changes leaders, they require a synchronous wait of  $\Omega(\Delta)$  time during view-change process. Protocols such as OptSync [25] can perform responsive leader rotation. However, they require optimistic conditions where  $> 3n/4$  replicas behave honestly. In the absence of optimistic conditions, they incur a synchronous delay of  $\Omega(\Delta)$  time. In contrast, our protocol support responsive leader rotation in the absence of optimistic conditions while still tolerating  $f < n/2$  Byzantine faults.

In addition, the commit step in our protocol is non-blocking i.e., we do not require replicas to commit before moving into higher epoch. Thus, our protocol can make progress at network speed while supporting responsive leader rotation and can commit  $k$  proposals in  $2\Delta + O(k\delta)$  time with different leaders when there is a sequence of honest leaders. In this regard, our protocol can be viewed as rotating-leader version of Sync HotStuff.

**Epochs.** Our protocol progresses through a series of numbered *epochs* with each epoch  $e$  coordinated by a distinct leader  $L_e$ . Epochs are numbered by non-negative integers starting with 1. The leaders for each epoch are rotated irrespective of the progress made in each epoch. For simplicity, we use round-robin leader election and the leader of epoch  $e$ , represented as  $L_e$ , is determined by  $e \bmod n$ . The leaders can also be selected at random by using public known function such as random beacons [12] which allows leader election in  $O(\delta)$  time. When the leader of an epoch is honest, the protocol progresses through epoch responsively, i.e., in  $O(\delta)$  time; otherwise each epoch lasts for  $7\Delta$  time.

**Blocks and block format.** We represent a proposal in an epoch in the form of a *block*. Each block references its predecessor with the exception of the genesis block which has no predecessor. A block  $B_k$  is said to be *valid* if (i) its predecessor block is valid, or if  $k = 1$ , predecessor is  $\perp$ , and (ii) the payload in the block meets the application-level validity conditions. Note that a leader  $L_e$  proposes a single block in an epoch.

**Certified blocks, and locked blocks.** A block certificate on a block  $B_k$  consists of  $t + 1$  distinct signatures in an epoch  $e$  and is represented by  $C_e(B_k)$ . We define a simple ranking rule as leaders propose a single block in each epoch. Block certificates are ranked by epochs, i.e., blocks certified in a higher epoch has a higher rank. During the protocol execution, each replica keeps track of all certified blocks and keeps updating the highest certified block to its knowledge. Replicas will lock on highest ranked certified blocks and do not vote for blocks that do not extend highest ranked block certificates to ensure safety of a commit.

**Block extension and equivocation.** A block  $B_k$  *extends* a block  $B_l$  ( $k \geq l$ ) if  $B_l$  is an ancestor of  $B_k$ . Note that a block  $B_k$  extends itself. Two blocks  $B_k$  and  $B_{k'}$  proposed in the same epoch *equivocate* one another if they are not equal.

## 4.1 Protocol Details

### ■ Protocol 1 Rotating-leader BFT SMR.

For each epoch  $e = 1, 2, \dots$ , replica  $r$  performs following operations:

1. **Epoch Advancement.** When `epoch-timere` reaches 0, broadcast  $\langle \text{clock}, e + 1 \rangle_r$ . Replica  $r$  advances to epoch  $e + 1$  using following rules:
  - On receiving  $f + 1$  votes for epoch  $e$  block.
  - On receiving  $f + 1$  epoch  $e$  clock messages.
 Upon entering epoch  $e + 1$ , broadcast an epoch  $e$  certificate and send highest ranked block certificate to  $L_{e+1}$ , set `epoch-timere+1` to  $7\Delta$  and start counting down. Abort `epoch-timere`.
2. **Propose.** If leader  $L_e$  of epoch  $e$  has  $C_{e-1}(B_l)$  propose immediately; otherwise wait for  $2\Delta$  time. Broadcast  $\langle \text{propose}, B_k, e, C_{e'}(B_l) \rangle_L$  where  $B_k$  extends highest certified block  $B_l$  known to  $L_e$ .
3. **Vote.** Upon receiving the first proposal  $\langle \text{propose}, B_k, e, C_{e'}(B_l) \rangle_L$ , if  $B_k$  extends the highest ranked certificate known to replica  $r$ , broadcast a vote in the form of  $\langle \text{vote}, B_k, e \rangle_r$ .
4. **(Non-blocking) Commit.** If `epoch-timere`  $> 2\Delta$  and replica  $r$  receives  $C_e(B_k)$ , set `commit-timere` to  $2\Delta$  and start counting down. When `commit-timere` reaches 0, if no epoch- $e$  equivocation has been detected, commit  $B_k$  and all its ancestors.
5. **Equivocation.** Forward the equivocating hashes signed by  $L_e$  and abort `commit-timere`. While still in epoch  $e$ , broadcast  $\langle \text{clock}, e + 1 \rangle_r$ .

## 27:10 Optimal Good-Case Latency Rotating Leader Synchronous BFT

Our protocol (refer Protocol 1) is simple and includes following five steps for an epoch  $e$ .

**Epoch advancement.** Each replica  $r$  keeps track of epoch duration  $\text{epoch-timer}_e$  for epoch  $e$ . When its  $\text{epoch-timer}_{e-1}$  expires, replica  $r$  broadcasts an epoch  $e$  clock message, i.e.,  $\langle \text{clock}, e \rangle_r$  to all replicas. Replica  $r$  enters epoch  $e$  when it receives either  $f + 1$  distinct  $\langle \text{clock}, e \rangle$  messages (i.e., an epoch  $e$  clock certificate) or when it receives an epoch  $e$  block certificate  $\mathcal{C}_e(B_l)$  for some block  $B_l$ . Upon entering epoch  $e$ , replica  $r$  broadcasts an epoch  $e$  certificate (either clock certificate or block certificate) to perform epoch synchronization among all honest replicas. Replica  $r$  also sends its highest ranked certificate  $\mathcal{C}_e(B_l)$  to the leader  $L_e$  if it sent a clock certificate while entering epoch  $e$ . In addition, it aborts all timers below epoch  $e$  and sets  $\text{epoch-timer}_e$  to  $7\Delta$  and starts counting down.

**Propose.** Upon entering epoch  $e$ , if Leader  $L_e$  has an epoch  $e-1$  block certificate, it proposes immediately; otherwise, it waits for  $2\Delta$  time to ensure it can receive the highest ranked certificate from all honest replicas. The leader  $L_e$  proposes a block  $B_k := (b_k, H(B_{k-1}))$  by broadcasting  $\langle \text{propose}, B_k, v, \mathcal{C}_{e'}(B_l) \rangle_{L_e}$  where  $\mathcal{C}_{e'}(B_l)$  is the highest ranked certificate known to  $L_e$ .

**Vote.** When a replica  $r$  receives the first proposal for  $B_k$  either from  $L_e$  or through some other replica, if  $r$  hasn't received a proposal for an equivocating block, i.e., it has not detected a leader equivocation in epoch  $e$ , it forwards the proposal to all replicas. If block  $B_k$  extends the highest ranked certificate known to replica  $r$ , it broadcasts a vote for  $B_k$  in the form of  $\langle \text{vote}, B_k, e \rangle_r$ ; otherwise, replica  $r$  does not vote for the proposed block. Voting for blocks that extends the highest ranked certificate ensures safety of committed blocks.

**Commit.** When replica  $r$  receives  $f + 1$  distinct vote messages for an epoch  $e$  block  $B_k$  (denoted by  $\mathcal{C}_e(B_k)$ ) and when  $\text{epoch-timer}_e$  is large enough ( $2\Delta$ ), it sets its  $\text{commit-timer}_e$  to  $2\Delta$  and starts counting down. Note that replica  $r$  moves to epoch  $e + 1$  as soon as it receives  $\mathcal{C}_e(B_k)$  while its  $\text{commit-timer}_e$  for block  $B_k$  is still running. When  $\text{commit-timer}_e$  reaches 0, if no equivocation for epoch- $e$  has been detected, replica  $r$  commits  $B_k$  and all its ancestors. Waiting for  $2\Delta$  wait before commit ensures no honest replica has voted for an equivocating block in epoch  $e$ . In addition, honest replicas start their  $\text{commit-timer}_e$  only when their  $\text{epoch-timer}_e \geq 2\Delta$ . This ensures no honest replica entered an higher epoch without receiving  $\mathcal{C}_e(B_k)$ .

**Equivocation.** At any time in epoch  $e$ , if a replica  $r$  detects an equivocation, it broadcasts equivocating hashes signed by leader  $L_e$  along with an epoch  $e$  clock message. Replica  $r$  also stops performing epoch  $e$  operations.

**How does our protocol support responsive leader rotation?** In general, consensus protocols require that all honest replicas receive and lock on a unique certificate for a block to be committed in an epoch before moving to higher epoch and not vote for blocks that do not extend this unique certificate to ensure safety of a commit. Prior synchronous protocols such as Sync HotStuff [2] achieve this property by adding a synchronous wait of at least  $1\Delta$  while moving to higher epoch. In Sync HotStuff [2], a replica starts its  $\text{commit-timer}$  of  $2\Delta$  as soon as it votes for the proposed block and commits when it does not detect any equivocation in the epoch before its  $\text{commit-timer}$  expires. While the replica that committed may not have detected an equivocation before its commit, other honest replicas might have detected

an equivocation. Waiting for  $\Delta$  time before moving to higher epoch ensures all honest replicas receive at least  $f + 1$  votes for the committed block. However, a synchronous wait of full  $\Delta$  during epoch-change makes protocols non-responsive. To achieve responsive epoch-change, our protocol instead waits for a certificate for the proposed block before starting the `commit-timer` and forwards the received certificate. Thus, replicas can responsively receive the certificate. In addition, a single block is proposed in an epoch. Thus, if the next leader receives the certificate for the current proposed block, it can propose immediately as it is already the highest ranked block certificate and all honest replicas will vote for the block extending this certificate. Initiating the `commit-timer` only upon receiving a certificate for the proposed block has also been explored in prior protocols such as Flexible BFT [19] which works in hybrid model (with both synchronous and partial synchronous assumptions) and follows stable-leader approach. In contrast, our protocol follows rotating-leader approach under synchrony assumption.

**How does  $2\Delta$  wait ensure safety?** Consider an honest replica  $r$  that commits a block  $B_k$  in epoch  $e$  at time  $t$ . Replica  $r$  must have received  $\mathcal{C}_e(B_k)$  at time  $t - 2\Delta$  and did not detect any block  $e$  equivocation by time  $t$ . This implies no honest replica either received or voted for an equivocating block in epoch  $e$  by time  $t - \Delta$ ; otherwise, replica  $r$  would have detected an epoch  $e$  equivocation by time  $t$ . In addition, since all honest replicas receive proposal for  $B_k$  by time  $t - \Delta$ , no honest replica will vote for an equivocating block in epoch  $e$ . This ensures the certificate for block  $B_k$  is unique. In addition, since replica  $r$  committed in epoch  $e$ , its `epoch-timere` must have been `epoch-timere`  $> 2\Delta$  at time  $t - 2\Delta$ . Since, honest replicas are synchronized within  $\Delta$  time, honest replicas that are still in epoch  $e$  must have `epoch-timere`  $> \Delta$  by time  $t - 2\Delta$ . Replica  $r$  broadcasts  $\mathcal{C}_e(B_k)$  at time  $t - 2\Delta$  when it starts its `commit-timer` at  $t - 2\Delta$ . Thus, these replicas will receive  $\mathcal{C}_e(B_k)$  before entering epoch  $e + 1$ . This ensures all honest replicas receive  $\mathcal{C}_e(B_k)$  before entering epoch  $e + 1$  and hence, no honest replica will vote for blocks that do not extend  $\mathcal{C}_e(B_k)$ . This ensures safety of committed block  $B_k$ .

Note that it is not required for all honest replicas to commit block  $B_k$  in the same epoch  $e$ . A Byzantine leader may send equivocating blocks to other honest replicas after time  $t - \Delta$  and replica  $r$  may not receive such equivocation before its `commit-timer` expires. However, if an honest replica  $r$  commits a block  $B_k$ , our protocol ensures that all honest replicas receive a unique certificate for  $B_k$  before entering epoch  $e + 1$  and no honest replica vote for blocks that do not extend  $B_k$  in higher epoch. Eventually, due to leader rotation, there will be an honest leader and this honest leader will propose block  $B_h$  extending  $B_k$ . All honest replicas will commit block  $B_h$  proposed by an honest leader and all honest replicas will commit  $B_k$  via ancestor rule (since  $B_h$  extends  $B_k$ ).

► **Remark.** Our protocol can be extended with an additional commit rule that commits a block  $B_\ell$  proposed in epoch  $e - f$  at the end of epoch  $e$  if the highest ranked chain in epoch  $e$  extends  $B_\ell$ , assuming there is a set of unique  $f + 1$  leaders in the last  $f + 1$  epochs. Such a commit rule can be used to commit faster when there is a set of  $f + 1$  consecutive honest leaders and progressing through  $f + 1$  epochs is faster than waiting for  $2\Delta$  time and obtain better commit latency.

The rationale behind the commit rule is the following. Out of last  $f + 1$  honest leaders, there will be at least one honest leader between epochs  $e - f$  and  $e$ . Consider an epoch  $e'$  (with  $e - f < e' < e$ ) and its honest leader  $L_{e'}$ . The leader  $L_{e'}$  will extend on the highest ranked chain certificate and propose some block  $B_k$  in a timely manner. Assume  $B_k$  extends  $B_\ell$ . Thus, all other honest replicas will vote for the proposed block  $B_k$  and all honest replicas will receive and lock on  $\mathcal{C}_{e'}(B_k)$  before entering epoch  $e' + 1$ . Hereafter, no honest replica will vote for a block that does not extend  $B_k$ , and all certified blocks after epoch  $e'$  must extend

$B_k$ , i.e., a conflicting chain of rank higher than  $\mathcal{C}_{e'}(B_k)$  cannot form. Thus, the highest ranked chain in epoch  $e$  must extend  $B_k$  and since  $B_k$  extends  $B_\ell$ ,  $B_\ell$  is safe to commit. A recent work, Apollo [4] uses such a commit rule to obtain better latency when there are a sequence of consecutive honest leaders. In their work, the protocol proposes without waiting for a certificate and can obtain  $O(f\delta)$  latency with a sequence of honest leaders. In contrast, our protocol requires waiting for a certificate to obtain a good-case latency of  $2\Delta$ .

Due to space constraints, we present security analysis in Appendix A.

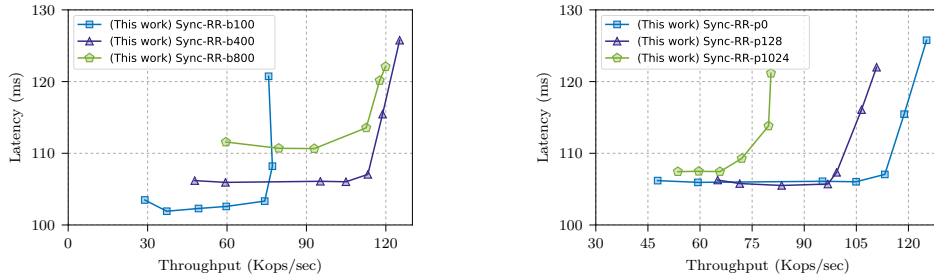
## 5 Evaluation

In this section, we compare the performance of our protocol with state-of-art synchronous protocol Sync HotStuff in both stable leader mode and rotating-leader mode.

### 5.1 Implementation Details and Methodology

Our implementation is an adaption of the open-source implementation of Sync HotStuff [11]. We modify the core consensus logic to our protocol and extend it to support leader rotation. We also extend Sync HotStuff protocol to support leader rotation after each block proposal. In this modified version, the leader is rotated every  $5\Delta$  time ( $2\Delta$  wait for the next leader before proposing in the new epoch,  $2\Delta$  time to commit a block and  $1\Delta$  wait during epoch-change.)

Each block consists of a batch of client commands. Each command contains a unique command identifier and an associated payload. The number of commands in a block determines its batch size. The throughput and latency results were measured from the perspective of external clients that run on separate machines from that of the replicas. The clients broadcast a configurable number of commands to every replica at certain configurable time interval. In all of our experiments, we ensure that the performance of replicas are not limited by lack of client commands.



(a) Varying batch sizes.

(b) Varying payload.

■ **Figure 2** Throughput vs. latency at varying batch sizes and payload at  $\Delta = 50\text{ms}$  and  $f = 1$ .

**Experimental Setup.** All our replicas and clients were installed on Amazon EC2 `c5.2xlarge` instances. Each instance has 8 vCPUs supported by Intel Xeon Platinum 8000 processors with maximum network bandwidth of upto 10Gbps. The network latency between two machines is measured to be less than 1ms. We used `secp256k1` for digital signatures in votes and a quorum certificate consists of an array of  $f + 1$  signatures.

**Baselines.** We make comparisons with the state-of-the-art synchronous protocol (Sync HotStuff) in two modes: (i) Sync HotStuff with a stable leader which is the default protocol, and (ii) Sync HotStuff with rotating-leaders with each leader making one block proposal per epoch.



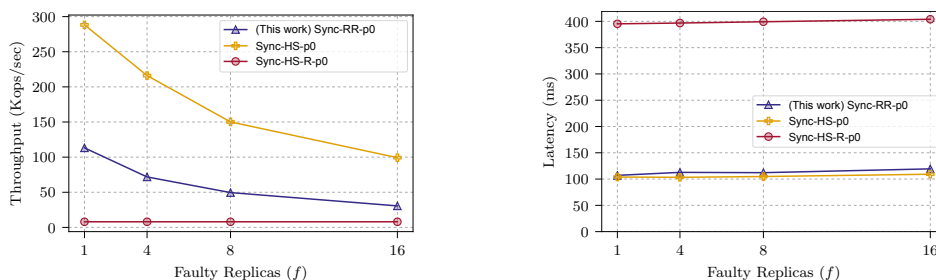
In general, stable leader protocols have better performance in terms of throughput. For fair comparison with our rotating-leader protocol, we also compare against a rotating-leader version of Sync HotStuff protocol where a new leader proposes a block every  $5\Delta$  time.

## 5.2 Basic Performance

We first evaluate the basic performance of our protocol when tolerating  $f = 1$  fault at  $\Delta = 50\text{ms}$ . We measure the observed throughput (i.e., number of committed commands per second) and the end-to-end latency for clients. In our first experiment (Figure 2a), each command has a zero-byte payload and we vary batch size at different values: 100, 400, and 800 as represented by the three lines in the graph.

Each point in the graph represents the measured throughput and latency for a run with a given load sent by clients. The load is increased by sending more number of commands in a given time interval. As seen in the graph, the throughput increases with increasing load without increasing latency upto a certain point before reaching saturation. After saturation, the latency increases while the throughput either remains consistent or slightly degrades. We observe that the throughput is maximum at around 113 Kops/sec when the batch size is 400 with a latency of around 107ms. We set the batch size to be 400 for our following experiments.

In our second experiment (Figure 2b), we vary the command request/response payload at different values in bytes 0/0, 128/128 and 1024/1024 with a fixed batch size of 400. We observe that as the payload size increases, the throughput, measured in number of commands, decreases. We also observe a marginal drop in latency with increasing payload.



(a)  $f$  vs. throughput.

(b)  $f$  vs. latency.

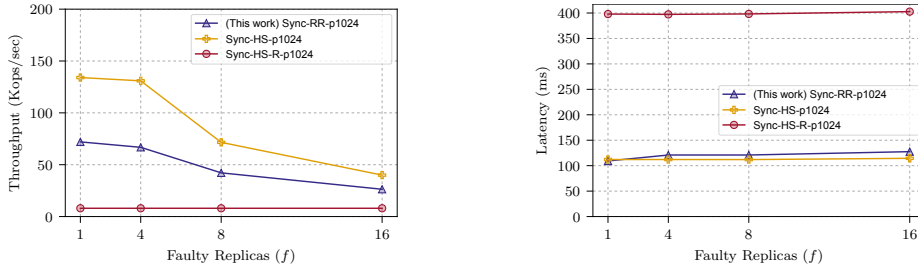
■ **Figure 3** Performance as function of faults at  $\Delta = 50\text{ms}$ , 400 batch size, and 0/0 payload.

## 5.3 Scalability and Comparison with Prior Work

Next, we evaluate our protocol scales as the number of replicas increase. We compare our protocol against standard Sync HotStuff (Sync-HS) and rotating-leader version of Sync HotStuff (Sync-HS-R). First, we evaluate the protocol performance with zero-payload commands to understand the raw overhead incurred by the underlying consensus mechanism at different values of  $f$  (Figure 3). Then, we study how the protocols perform at a higher payload of 1024/2024 (Figure 4). We use a batch size of 400 and  $\Delta$  of 50ms for both these experiments. For Sync-HS-R, we use a batch size of 2000. Each data point in the graphs represent the throughput and latency at the saturation point without overloading the replicas.

**Comparison with Sync HotStuff.** Figures 3 and 4 compares the throughput and latency of Sync Hotstuff with our protocol at two different payloads: 0/0 and 1024/1024. We observe the latency of the our protocol is similar to Sync HotStuff as both protocols have

## 27:14 Optimal Good-Case Latency Rotating Leader Synchronous BFT

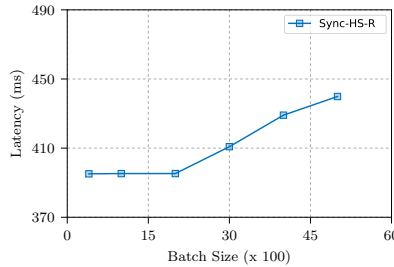


(a)  $f$  vs throughput.

(b)  $f$  vs latency.

■ **Figure 4** Performance as function of faults at  $\Delta = 50$ ms, optimal batch size, and 1024/1024 payload.

$2\Delta$  commit latency. However, Sync HotStuff has much better throughput compared to our protocol. This is due to following reasons: (i) the performance of Sync HotStuff depends on the  $f + 1$  fastest replicas in the system, (ii) being a stable-leader protocol, the leader can schedule block proposals as soon as sufficient commands to form a block are available. In contrast, the performance of our protocol is bottlenecked due to following reasons: (i) the performance depends on the slowest replica in the system due to round-robin leader rotation (ii) a leader can only schedule a block proposal after it has been elected as a leader, and (iii) since commands arrive in a different order at different replicas, an additional processing is required to filter out proposed commands (the additional processing incur around  $150\mu$ s). Although, the stable-leader Sync HotStuff provides better throughput, it is worth to note the stable-leader approach does not provide fairness and censorship resistance. Next, we compare our protocol against Sync HotStuff with leader rotation after each block proposal which provides better fairness and censorship resistance.



■ **Figure 5** Latency of Sync HotStuff with leader rotation at varying batch sizes at  $\Delta = 50$ ms and  $f = 1$ .

**Comparison with Sync HotStuff with leader rotation.** In Sync HotStuff with leader rotation, a leader proposes a block every  $5\Delta$  time i.e., every 250ms at  $\Delta = 50$ ms. Thus, the throughput of the protocol is essentially 4 times the proposal batch size. We first perform an experiment (Figure 5) to find a batch size for which the latency does not adversely worsen. We observe that at the batch size of 2000, the latency is similar to the latency at batch size of 400. At higher batch sizes, the latency of the protocol worsens as can be seen in the figure. For other experiments (Figures 3 and Figure 4), we set the batch size to be 2000; thus the throughput of the protocol was always 8000. The latency of the protocol is also very high at around 400ms. This is because the leader proposes a block every  $5\Delta$  while clients sent commands much earlier. Since, our protocol supports responsive leader rotation, the next

leader can propose as soon it receive a certificate for previous block. Thus, our protocol performs much better compared to Sync HotStuff with leader rotation in terms of latency and throughput.

## 6 Related Work

There has been a long line of work in designing efficient BFT SMR protocol [10, 8, 25, 21, 4, 3, 2, 5]. Most of these BFT SMR solutions [2, 3, 25] focus in increasing the performance of the system during steady state and hence follow a stable-leader paradigm. Our work focuses in improving the performance of system while rotating-leaders every epoch as rotating-leader protocol provide better fairness and censorship resistance compared to stable leader protocols. We review the most recent and closely related works below. Compared to all of these protocol, our work commits in  $2\Delta$  time when the leaders are honest and the new leaders can propose responsively in  $O(\delta)$  time. In addition, we do not require optimistic conditions to change leaders responsively.

The protocols due to PiLi [10] and Streamlet [8] provide BFT SMR protocols that change leaders after every epoch. Their leader selection is randomized. Both of these protocol assume lock-step execution in epochs. In PiLi, each epoch lasts for  $O(\delta)$  (resp.  $5\Delta$ ) under optimistic (resp. synchronous) conditions. PiLi commits 5 blocks after 13 consecutive honest epochs. PiLi has a responsive (resp. synchronous) latency of at least  $16\delta-26\delta$  (resp.  $40\Delta-65\Delta$ ). Streamlet commits a single blocks after 6 consecutive honest epochs with each epoch taking  $2\Delta$  time. However, under  $f < n/2$  corruption, getting 6 or 13 consecutive honest epochs is extremely unlikely and these protocols may never progress. In contrast, our protocol requires a single honest epoch to make progress.

The protocols due to OptSync [25] and Hybrid-BFT [21] provide rotating-leader BFT SMR protocols. However, they change leaders responsively only during optimistic conditions. During normal conditions when  $f < n/2$  Byzantine faults are present, these protocol wait for at least  $7\Delta$  in an epoch before moving to the next leader even when there is a sequence of honest leaders. With a sequence of honest leaders, our protocol can progress through epochs in  $O(\delta)$  time despite tolerating  $f < n/2$  Byzantine faults.

Apollo [4] provides a rotating-leader BFT SMR protocol. In their protocol, the leader is rotating after every proposal in  $\delta$  time and the proposed blocks are committed after  $f + 1$  epochs irrespective of whether the leader is honest or Byzantine. If all the leaders in the next  $f + 1$  epochs are honest, their protocol commits with a latency of  $(f + 1)\delta$ . However, when  $f = O(n)$ , even in the *good-case*  $O(f)$  out of the next  $f + 1$  leaders may be Byzantine, thus yielding a latency of  $O(f\Delta)$  even when messages between honest parties are instantaneous. Thus, in the good-case, they fail to satisfy the first condition (latency of a single-slot) of our lower bound.

RandPiper [5] also provides a rotating-leader BFT SMR protocol. In their protocol, they present a BFT SMR protocol that achieve quadratic communication without using threshold signatures. However, it incurs  $11\Delta$  in every epoch and cannot progress responsively.

A recent work Internet Computer Consensus [6] provides a rotating leader BFT SMR in partially synchronous model. Similar to our work, their protocol also progresses to higher epoch as soon as a certificate is formed in the current epoch while the committing a block in the hindsight only when sufficient parties acknowledge the block certificate. In our protocol, we commit only when no equivocation is detected for  $2\Delta$  time after receiving a block certificate. In both cases, the protocols check to see if the block certificate is unique and sufficient parties have received the block certificate.

## References

- 1 Ittai Abraham, Dahlia Malkhi, Kartik Nayak, and Ling Ren. Dfinity consensus, explored. *IACR Cryptol. ePrint Arch.*, 2018:1153, 2018.
- 2 Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Maofan Yin. Sync hotstuff: Simple and practical synchronous state machine replication. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 654–667, 2020.
- 3 Ittai Abraham, Kartik Nayak, Ling Ren, and Zhuolun Xiang. Good-case latency of byzantine broadcast: A complete categorization. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, PODC’21, pages 331–341, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3465084.3467899.
- 4 Adithya Bhat, Akhil Bandarupalli, Saurabh Bagchi, Aniket Kate, and Michael K Reiter. Apollo—optimistically linear and responsive smr. Cryptology ePrint Archive, Report 2021/180, 2021. URL: <https://eprint.iacr.org/2021/180>.
- 5 Adithya Bhat, Nibesh Shrestha, Aniket Kate, and Kartik Nayak. Randpiper – reconfiguration-friendly random beacons with quadratic communication. Cryptology ePrint Archive, Report 2020/1590, 2020. , To appear in ACM SIGSAC CCS 2021. URL: <https://eprint.iacr.org/2020/1590>.
- 6 Jan Camenisch, Manu Drijvers, Timo Hanke, Yvonne-Anne Pignolet, Victor Shoup, and Dominic Williams. Internet computer consensus. Cryptology ePrint Archive, Report 2021/632, 2021. URL: <https://eprint.iacr.org/2021/632>.
- 7 Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.
- 8 Benjamin Y Chan and Elaine Shi. Streamlet: Textbook streamlined blockchains. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, pages 1–11, 2020.
- 9 T-H Hubert Chan, Rafael Pass, and Elaine Shi. Pala: A simple partially synchronous blockchain. *IACR Cryptology ePrint Archive*, 2018:981, 2018.
- 10 T-H Hubert Chan, Rafael Pass, and Elaine Shi. Pili: An extremely simple synchronous blockchain. *IACR Cryptology ePrint Archive*, 2018:980, 2018.
- 11 Determinant. Sync-HotStuff. URL: <https://github.com/hot-stuff/librightstuff>.
- 12 Drand. Drand - a distributed randomness beacon daemon. URL: <https://github.com/drand/drand>.
- 13 Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.
- 14 Michael J Fischer, Nancy A Lynch, and Michael Merritt. Easy impossibility proofs for distributed consensus problems. *Distributed Computing*, 1(1):26–39, 1986.
- 15 Matthias Fitzi. *Generalized communication and security models in Byzantine agreement*. PhD thesis, ETH Zurich, 2002.
- 16 Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 51–68, 2017.
- 17 Timo Hanke, Mahnush Movahedi, and Dominic Williams. Dfinity technology overview series, consensus system. *arXiv preprint*, 2018. arXiv:1805.04548.
- 18 Jonathan Katz and Chiu-Yuen Koo. On expected constant-round protocols for byzantine agreement. In *Annual International Cryptology Conference*, pages 445–462. Springer, 2006.
- 19 Dahlia Malkhi, Kartik Nayak, and Ling Ren. Flexible byzantine fault tolerance. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1041–1053, 2019.
- 20 Zarko Milosevic, Martin Biely, and André Schiper. Bounded delay in byzantine-tolerant state machine replication. In *2013 IEEE 32nd International Symposium on Reliable Distributed Systems*, pages 61–70. IEEE, 2013.

- 21 Atsuki Momose, Jason Paul Cruz, and Yuichi Kaji. Hybrid-bft: Optimistically responsive synchronous consensus with optimal latency or resilience. *IACR Cryptol. ePrint Arch.*, 2020:406, 2020.
- 22 Rafael Pass and Elaine Shi. Thunderella: Blockchains with optimistic instant confirmation. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 3–33. Springer, 2018.
- 23 Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.
- 24 Elaine Shi. Streamlined blockchains: A simple and elegant approach (a tutorial and survey). In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 3–17. Springer, 2019.
- 25 Nibesh Shrestha, Ittai Abraham, Ling Ren, and Kartik Nayak. On the optimality of optimistic responsiveness. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 839–857, 2020.
- 26 Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 347–356, 2019.

## A Safety and Liveness

We say a block  $B_k$  is committed directly in epoch  $e$  if it is committed as a result of its own `commit-timere` expiring. We say a block  $B_k$  is committed indirectly if it is a result of directly committing a block  $B_l$  ( $l > k$ ) that extends  $B_k$ .

▷ **Claim 11.** If an honest replica enters an epoch  $e$  at time  $t$ , then all honest replicas enter epoch  $e$  by time  $t + \Delta$ .

*Proof.* Suppose an honest replica  $r$  enters epoch  $e$  at time  $t$  and broadcasts an epoch  $e - 1$  certificate. Suppose for the sake of contradiction, an honest replica  $r'$  does not enter epoch  $e$  by time  $t + \Delta$ . Since replica  $r$  broadcasts epoch  $e - 1$  certificate at time  $t$ , the epoch  $e - 1$  certificate arrives all honest replicas by time  $t + \Delta$ . This implies replica  $r'$  receives epoch  $e - 1$  certificate and moves to epoch  $e$  by time  $t + \Delta$ . A contradiction. ◁

► **Corollary 12.** *The epoch timers of two honest replicas may differ by up to  $\Delta$  time.*

▷ **Claim 13.** If an honest replica broadcasts  $\langle \text{clock}, e + 1 \rangle$  in epoch  $e$ , no honest replica directly commits a block in epoch  $e$ .

*Proof.* Suppose an honest replica  $r$  sends  $\langle \text{clock}, e + 1 \rangle_r$  in epoch  $e$ . Replica  $r$  sends  $\langle \text{clock}, e + 1 \rangle_r$  on two cases (i) when its `epoch-timere` expires before receiving epoch  $e$  block certificate, and (ii) when it receives an epoch  $e$  equivocation while still in epoch  $e$ .

Suppose for the sake of contradiction, an honest replica, say replica  $r'$ , commits a block  $B_k$  in epoch  $e$  at time  $t$ . Since replica  $r'$  committed block  $B_k$  in epoch  $e$  at time  $t$ , it must have received a  $\mathcal{C}_e(B_k)$  such that its `epoch-timere`  $\geq 2\Delta$  at time  $t - 2\Delta$  and did not detect an epoch  $e$  equivocation by time  $t$ . By Corollary 12, replica  $r$ 's `epoch-timere` must be  $\geq \Delta$  at time  $t - 2\Delta$  in the worst case. In addition, no honest replica detected an epoch  $e$  equivocation by time  $t - \Delta$  and sent a  $\langle \text{clock}, e + 1 \rangle_r$  due to epoch  $e$  equivocation; otherwise replica  $r'$  would have detected epoch  $e$  equivocation by time  $t$  and would not commit. All honest replicas will receive an epoch  $e$  certificate for  $B_k$  by time  $t - \Delta$ . Thus, replica  $r$  would not send  $\langle \text{clock}, e + 1 \rangle_r$  in epoch  $e$ . A contradiction. ◁

► **Corollary 14.** *If a clock certificate exists in epoch  $e$ , no honest replica directly commits a block in epoch  $e$ .*

## 27:18 Optimal Good-Case Latency Rotating Leader Synchronous BFT

► **Lemma 15.** *If an honest replica commits a block  $B_k$  in epoch  $e$ , then (i) an equivocating block certificate does not exist in epoch  $e$  (ii) every honest replica receives  $\mathcal{C}_e(B_k)$  before entering epoch  $e + 1$ .*

**Proof.** Suppose an honest replica  $r$  directly commits an epoch- $e$  block  $B_k$  at time  $t$ . Replica  $r$  must have received a  $\mathcal{C}_e(B_k)$  at time  $t - 2\Delta$  such that its epoch-timer $_e \geq 2\Delta$ . All honest replicas receive the proposal for  $\mathcal{C}_e(B_k)$  by time  $t - 2\Delta$ .

For part (i), observe that after time  $t - \Delta$ , no honest replica will vote for an equivocating block in epoch  $e$ . If an honest replica voted for an equivocating block  $B'_{k'}$  before  $t - \Delta$  in epoch  $e$ , replica  $r$  would have received the equivocating proposal for  $B'_{k'}$  by time  $t$  and would not commit. This contradicts the hypothesis of  $r$  committing  $B_k$  directly in epoch  $e$ . Therefore, an equivocating block will not get any honest vote and will not be certified in epoch  $e$ .

By part(i) of the Lemma, there does not exist an equivocating block certificate and by Corollary 14, epoch- $e$  certificate must be a block certificate. Thus, all honest replicas receive  $\mathcal{C}_e(B_k)$  and enter epoch  $e + 1$ . ◀

► **Lemma 16 (Unique Extensibility).** *If an honest replica directly commits  $B_e$  in epoch  $e$ , then any certified block that ranks higher than  $\mathcal{C}_e(B_k)$  must extend  $B_e$ .*

**Proof.** The proof is by induction on epochs  $e' > e$ . For an epoch  $e'$ , we prove that if  $\mathcal{C}_{e'}(B_{k'})$  exists then it must extend  $B_k$ . A simple induction shows that all later block certificates must also extend  $B_k$ .

For the base case, where  $e' = e + 1$ , the proof that  $\mathcal{C}_{e'}(B_{k'})$  extends  $B_k$  follows from Lemma 15. The only way  $\mathcal{C}_{e'}(B_{k'})$  forms is if some honest replica votes for  $B_{k'}$  using Step 3 in Protocol 1. Honest replica votes in epoch  $e'$  only if it extends a highest certified block. By Lemma 15, there does not exist an equivocating block certificate in epoch  $e$  and all honest replicas receive  $\mathcal{C}_e(B_k)$  before entering epoch  $e + 1$ . Thus, no honest will vote for a block that does not extend  $B_k$ .

Given that the statement is true for all epoch below  $e'$ , the proof that  $\mathcal{C}_{e'}(B_{k'})$  extends  $B_k$  follows from the induction hypothesis because the only way such a block certificate forms is if some honest replica votes for it. Since honest replicas vote in epoch  $e'$  with a valid epoch  $e' - 1$  certificate and by induction hypothesis on certificates of epoch  $e < e'' < e'$ ,  $\mathcal{C}_{e'}(B_{k'})$  must extend  $B_k$ . ◀

► **Theorem 17 (Safety).** *Honest replicas do not commit conflicting blocks for any epoch  $e$ .*

**Proof.** Suppose for the sake of contradiction two distinct blocks  $B_k$  and  $B'_k$  are committed in epoch  $e$ . Suppose  $B_k$  is committed as a result of  $B_{k'}$  being directly committed in epoch  $e'$  and  $B'_k$  is committed as a result of  $B'_{k''}$  being directly committed in epoch  $e''$ . Without loss of generality, assume  $k' < k''$ . Note that all directly committed blocks are certified. By Lemma 16,  $B'_{k''}$  extends  $B_{k'}$ . Therefore,  $B_k = B'_k$ . ◀

► **Theorem 18 (Liveness).** *All honest replicas keep committing new blocks.*

**Proof.** Each epoch lasts for  $7\Delta$  time. If the leader is Byzantine and does not propose any blocks or proposes equivocating blocks, an epoch change will trigger and change the leader. Due to round robin leader election, there will be an honest leader.

Consider an honest epoch  $e$  and its leader  $L_e$ . Let  $t$  be the time when the first honest replica enters epoch  $e$ . By Claim 11, all honest replicas enter epoch  $e$  by time  $t + \Delta$ . If  $L_e$  has  $\mathcal{C}_{e-1}(B_l)$ , it proposes immediately, otherwise it waits for  $2\Delta$  to receive the highest ranked block certificate  $\mathcal{C}_{e'}(B_l)$ . In any case,  $L_e$  proposes by time  $t + 3\Delta$  which arrives all honest replicas by time  $t + 4\Delta$ .

Since, leader  $L_e$  is honest, it proposes a block  $B_k$  that extends highest ranked certificate  $\mathcal{C}_{e'}(B_l)$ , all honest replicas will vote for it by time  $t + 4\Delta$ . Thus, all honest replicas will receive  $\mathcal{C}_e(B_k)$  by time  $t + 5\Delta$  and move to epoch  $e + 1$ . Observe that  $\text{epoch-timer}_e \geq 2\Delta$  for all honest replicas by time  $t + 5\Delta$ . Thus, all honest replicas start their  $\text{commit-timer}_e$ . Since leader  $L_e$  is honest, an epoch  $e$  equivocation does not exist. Thus, all honest replicas will commit. ◀





# Strongly Linearizable Linked List and Queue

Steven Munsu Hwang<sup>1</sup> ✉

University of Calgary, Canada

Philipp Woelfel ✉

University of Calgary, Canada

---

## Abstract

---

Strong linearizability is a correctness condition conceived to address the inadequacies of linearizability when using implemented objects in randomized algorithms. Due to its newfound nature, not many strongly linearizable implementations of data structures are known. In particular, very little is known about what can be achieved in terms of strong linearizability with strong primitives that are available in modern systems, such as the compare-and-swap (CAS) operation.

This paper kick-starts the research into filling this gap. We show that Harris’s linked list and Michael and Scott’s queue, two well-known lock-free, linearizable data structures, are not strongly linearizable. In addition, we give modifications to these data structures to make them strongly linearizable while maintaining lock-freedom. The algorithms we describe are the first instances of non-trivial, strongly linearizable data structures of their type not derived by a universal construction.

**2012 ACM Subject Classification** Theory of computation → Shared memory algorithms

**Keywords and phrases** Strong linearizability, compare-and-swap, linked list, queue, lock-freedom

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2021.28

**Funding** We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC) under Discovery Grant RGPIN-2019-04852, and the Canada Research Chairs program.

## 1 Introduction and Related Work

Linearizability [9] is the correctness condition of choice for asynchronous shared memory algorithms. Intuitively, it requires that an operation on a concurrent object appears to take effect instantaneously at some point (the linearization point) between the operation’s invocation and response. Arranging the operations by these points must result in a sequential history that is valid respective to the object’s specification. Due to this notion of “taking effect at a point in time”, linearizability was considered to be practically equivalent to atomicity. In fact, atomic and linearizable objects can be interchanged without altering the worst-case behaviour of an algorithm [9]. Importantly, linearizability has been proven to be a local property [9]. Informally, a property is local if the system satisfies the property given that each object used by the system satisfies the property. Linearizability is also composable, meaning that a linearizable object implemented using atomic objects is still linearizable when the atomic objects are replaced with linearizable ones. These two properties make linearizability desirable for modular programming.

Unfortunately, linearizability is not as suitable for use in randomized algorithms: Golab, Higham and Woelfel [5] showed that the probability distributions over the set of outcomes can change when atomic objects are replaced with linearizable ones.

They proposed *strong linearizability*, which when satisfied maintains the same probability distribution over the set of outcomes as with atomic objects, under a strong adaptive adversary. In fact, strong linearizability is sufficient and necessary for that. Strong linearizability demands that future events do not change the linearization points of the past. Strong

---

<sup>1</sup> Corresponding author



linearizability is also local and composable [5], further motivating the search for such implementations.

Until now, most work on strong linearizability assumed that processes communicate only using atomic (read/write) registers. Helmi, Higham and Woelfel [7] have shown that essentially no non-trivial object has a deterministic wait-free strongly linearizable implementation from *single-writer* registers. Using *multi-writer* registers, a number of strongly linearizable *lock-free* algorithms have been devised, such as bounded max-registers [7], counters [2] and single-writer snapshots [2, 12]. On the other hand, for none of these objects, strongly linearizable *wait-free* implementations exist [2].

The impossibility result of wait-free consensus [3, 10] established that atomic read/write registers are too weak to solve fundamental shared memory problems. Even simple data structures, such as queues or stacks, have no lock-free linearizable algorithms [8]. On the other hand, so-called universal constructions, such as Herlihy's [8], show that  $n$ -process consensus objects can be used to obtain wait-free linearizable implementations of any type with a deterministic sequential specification. In fact, Herlihy's universal construction is even strongly linearizable [5].

Almost all of today's systems provide strong synchronization primitives, such as atomic compare-and-swap, which allows wait-free solutions to the consensus problem for arbitrary many processes. Therefore, all types with a deterministic sequential specification have wait-free strongly linearizable implementations (using the universal construction). But the universal construction is not practical, as it is neither space nor time efficient.

Employing strong synchronization primitives (most commonly compare-and-swap), many efficient linearizable solutions to fundamental data structure problems have been devised. But it is generally not known whether these data structures are also strongly linearizable, and thus whether they can safely be used in randomized algorithms against a strong adaptive adversary.

Essentially no efficient strongly linearizable implementations are known for fundamental data structures that require strong synchronization primitives (or at least it is not known, whether existing linearizable implementations are also strongly linearizable). This paper aims to kick-start the research needed to fill this gap. We investigate two well known standard data structures: Harris's linked list [6], and Michael and Scott's queue [11]. Both algorithms use compare-and-swap objects, and are linearizable and lock-free. We show that they are not strongly linearizable (see Section 3 for Harris's linked list and Section 5 for Michael and Scott's queue). We then show that relatively simple modifications to these data structures yield strong linearizability (see Sections 4 and 6 respectively).

## 1.1 Other Related Work

As mentioned earlier, most non-trivial results on strong linearizability assume that processes cannot perform strong atomic operations (only atomic reads and writes are permitted). An exception is a recent randomized implementation of a double-compare-and-swap (DCAS) object from compare-and-swap objects [4], which uses as a build block (and implements) a strongly linearizable restricted DCAS object. Moreover, Attiya, Castañeda, and Hendler [1] proved that wait-free strongly linearizable implementations of stacks and queues from "readable" base objects, require that these base objects have consensus number infinity.

## 2 Preliminaries

We consider a distributed shared memory system with  $n$  processes communicating through shared objects. Each shared object has a *type*, which outlines a set of operations, and is defined by a *sequential specification*, a set of valid sequences of operations. An operation

$op$  consists of an *invocation event*, denoted  $inv(op)$ , and possibly a matching *response event* denoted  $rsp(op)$ .

A *transcript* is a sequence of invocation and response events of operations. For transcripts  $T$  and  $Y$ , we denote  $T \circ Y$  as the concatenation of the two transcripts. A *projection* of a transcript  $T$  onto a process  $p$  is denoted  $T|p$ , and is the sequence of invocation and response events by  $p$  in  $T$ . A projection of  $T$  onto an object  $O$ , denoted as  $T|O$ , is the sequence of invocation and response events of operations in  $T$  performed on  $O$ . An operation  $op$  is *complete* in some transcript  $T$  if  $T$  contains its invocation and matching response. A transcript is complete if every operation in the transcript is complete. In a transcript, an operation  $op$  is *atomic* if  $rsp(op)$  immediately follows  $inv(op)$ . An object  $O$  is atomic if every operation on  $O$  in any transcript is atomic. On the other hand, an object  $O$  is *implemented* if each operation  $op$  on  $O$  is a method using other implemented or base objects (objects provided by the system). Formally, a method is associated with a sequence of operations such that when  $op$  is called, the sequence of operations are executed.

A transcript  $T$  defines a *happens before* order  $\xrightarrow{T}$  for operations  $op_1, op_2 \in T$ , where  $op_1 \xrightarrow{T} op_2$  if and only if  $rsp(op_1)$  occurs before  $inv(op_2)$  in  $T$ . Note that this is a partial order.

A *history* is a transcript where for every process  $p$ , every operation in  $H|p$  is atomic. A history  $S$  is *sequential* if every operation in  $S$  is atomic. A sequential history  $S$  is *valid* if and only if for any object  $O$ ,  $H|O$  is in the sequential specification of the type of  $O$ . For every incomplete operation in  $H$ , if either the operation is discarded, or a response is appended, we obtain a complete history  $H'$  called a *completion* of  $H$ .

An *interpreted history*  $\Gamma(T)$  can be derived from a transcript detailing an algorithm execution using an implemented object  $O$ . It is obtained by iterating through all  $p$ , and removing all events by  $p$  after  $inv(op)$  and not after its matching response  $rsp(op)$  for any operation  $op$  by  $p$ . Intuitively, the interpreted history consists of invocation and response events of "high level" operations in  $T$ ; all intermediate steps for methods are removed from the transcript. For a set of transcripts  $\mathcal{T}$ ,  $\Gamma(\mathcal{T}) = \{\Gamma(T) \mid T \in \mathcal{T}\}$ .

Consider  $H'$ , a completion of a history  $H$ . A *linearization* [9] of  $H'$  is a sequential history  $S$  satisfying all of the following:

- All operations in  $H'$  are in  $S$ .
- For all operations  $op_1$  and  $op_2$  in  $H$ , if  $op_1 \xrightarrow{H'} op_2$ , then  $op_1 \xrightarrow{S} op_2$ .
- $S$  is valid.

For an implementation of a shared object to be *linearizable*, every possible history on the object must have a linearizable completion. A function  $f$ , mapping each history  $H$  from a set  $\mathcal{H}$  of histories to a linearization  $f(H)$  of  $H$ , is called *linearization function* for the set  $\mathcal{H}$ .

Given a transcript  $T$ , if an event  $e$  is the  $t$ -th element of  $T$ , then we say that  $e$  occurs at time  $t$  in  $T$ , or that  $time_T(e) = t$ . If  $e$  is not present in  $T$ , then we define  $time_T(e) = \infty$ . For an atomic operation  $am$  in a transcript  $T$ , we say that  $am$  occurs at  $time_T(rsp(am))$ . If an implemented operation  $op$  performs an atomic operation on line  $x$  of the method corresponding to the operation, we refer to this atomic operation as  $op_x$ . If  $op$  is complete in a transcript  $T$ , then by  $time_T(op_x)$ , we refer to the last time at which  $op_x$  is executed during  $op$ .

Linearizability can also be expressed through *linearization points*. Consider a transcript  $T$ , and a linearizable object  $O$ . A *linearization point function*  $pt$  for  $O$  maps  $op \in \Gamma(T|O)$  to  $\infty$  or a time in  $T$  such that

1.  $pt(op) \in [time_T(inv(op)), time_T(rsp(op))]$ , and
2. there is a valid sequential history  $S$  of  $\Gamma(T|O)$  where for all  $op_1, op_2 \in S$ , if  $op_1 \xrightarrow{S} op_2$  then  $pt(op_1) \leq pt(op_2)$ . (This property ensures that  $S$  preserve the happens-before-order)

## 28:4 Strongly Linearizable Linked List and Queue

of  $T$ . It is possible to have  $pt(op_1) = pt(op_2)$ : this simply means that  $op_1$  and  $op_2$  can appear in  $S$  in either relative order without violating the happens-before-order.)

We call  $pt(op)$  the linearization point of  $op$ . Intuitively, this is the point in time where operation  $op$  "appears" to take effect. For a set of transcripts  $\mathcal{T}$ , the *prefix closure* of  $\mathcal{T}$  is the set containing all prefixes of transcripts in  $\mathcal{T}$ . We denote the prefix closure of  $\mathcal{T}$  as  $close(\mathcal{T})$ . A function  $f : \mathcal{T} \rightarrow \mathcal{T}'$ , where  $\mathcal{T}$  and  $\mathcal{T}'$  are sets of transcripts, is *prefix preserving* if for any two transcripts  $T, Y \in \mathcal{T}$  where  $T$  is a prefix of  $Y$ ,  $f(T)$  is a prefix of  $f(Y)$ .

A function  $f$  is a *strong linearization* [5] function for a set of transcripts  $\mathcal{T}$  if:

- $f$  is a linearization function for  $\Gamma(close(\mathcal{T}))$ , and
- $f$  is prefix preserving.

An implemented object  $O$  is *strongly linearizable* if and only if the set of transcripts on  $O$  has a strong linearization function.

A method of an implemented object is *lock-free* if it guarantees that if a process executing a method takes infinitely many steps, then infinitely many method calls finish within a finite number of steps. An object is lock-free if every operation on the object is lock-free. An implemented object is *wait-free* if every method call terminates after taking finitely many steps.

Our algorithms use an atomic compare-and-swap object, which has an operation denoted as *CAS*. The operation takes two arguments: *old* and *new*. If the value of the object is equal to *old*, then the operation overwrites the value of the object to *new*. Otherwise, the operation has no effect. In addition, the object also allows reads and writes.

### 3 Harris's linked list is not strongly linearizable

Nodes in Harris's linked list implementation have the following fields: *key* and *succ*. Field *key* stores the value of the node, and is taken as the argument by the node's constructor. Once set, the value of *key* never changes for a particular node. The successor field, *succ*, is a *CAS* object which contains *next*, the next node in the list, and *marked*, a boolean value to indicate whether the node has been "logically deleted". A node is marked before being excised ("physically deleted") from the linked list. As a shorthand, we access different parts of the *succ* field of a node  $v$  by  $v.next$  or  $v.marked$ . The successor field is initialized to  $(null, false)$ .

There are two shared variables *Head* and *Tail*, which represent the head and tail sentinel nodes of the list. *Head* has a key value of  $-\infty$ , and the *Tail* has a key value of  $\infty$ . Initially, *Head* and *Tail* are the only nodes in the linked list, where  $Head.succ = (Tail, false)$  and  $Tail.succ = (null, false)$ .

The sequential specification of the linked list we consider is as follows. The linked list consists of three methods: *delete*, *insert* and *find*. All three operation return a boolean value to designate whether the operation has failed or succeeded. The nodes are sorted by their keys, and all keys in the linked list are unique. An *insert* operation fails if the key being inserted is already in the linked list, and otherwise it succeeds. Likewise, a *delete* operation fails if the key being deleted is not in the linked list. The return value of *find* indicates whether a key is in the linked list.

Harris's linked list uses helper function  $search(search\_key)$ , which returns nodes *left* and *right* with the following guarantees: at some point in time during the execution of *search*,

1.  $left.key < search\_key \leq right.key$ ,
2. *left* and *right* are unmarked and

### 3. $left.next = right$ .

The *search* method is used to locate the nodes of interest for each operation of the linked list. For example, a successful *insert* adds a new node between *left* and *right* returned by *search*. It is also used to verify existence of keys in the linked list. Since the keys are in sorted order, at points in time when the search conditions are met, *left* contains the largest key less than *search\_key* and *right* contains the smallest key less than or equal to *search\_key*. If  $right.key = search\_key$  then the linked list contains *search\_key* at a point in time when the search conditions held; if  $right.key \neq search\_key$ , then the linked list does not contain *search\_key* at such a point in time. These are precisely the points when *find*, a failed *delete* and a failed *insert* should linearize.

Recall that, intuitively, strong linearizability requires that future events do not affect the linearization order of the past. That is, events that occur after a linearization point should not affect the position of that linearization point in a history. However, at the point in time when the search conditions hold, Harris's linked list does not guarantee which nodes will be returned by *search*. In other words, if time  $t$  is when the search conditions are true, it is possible to change the history after  $t$  (i.e. change the "future") such that  $search(search\_key)$  returns a different pair of nodes, which can change the response of the operation invoking *search*. This suggests that the linked list is not strongly linearizable. By altering the response of an operation through future events, the linearization order of the past will likely need to change to maintain validity. We use this observation in our proof of Lemma 1.

Note that a successful *insert* linearizes when a new node is inserted by a successful *CAS* on line 61. Likewise, a successful *delete* linearizes when a node is marked by a successful *CAS* on line 45. These linearization points are already strongly linearizable; events that occur after a *CAS* cannot influence whether the *CAS* succeeds. Thus our modifications in the next section focus on the *search* function.

► **Lemma 1.** *The linked list implementation by Harris (Figure 1) is not strongly linearizable.*

A full proof of the lemma is provided in Appendix A. The high level idea is as follows.

Let  $f$  be a linearization function for the linked list. We denote  $node_i$  as the node containing key  $i$ ,  $in_p(x)$  as a transcript of an insert of key  $x$  by process  $p$  and  $del_p(x)$  as a transcript of a delete of key  $x$  by  $p$ .

Consider the following transcripts for processes  $p$  and  $q$ :

$$S = in_p(3) \circ (del_q(2) \text{ to the first execution of line 15}) \circ in_p(2)$$

$$T_1 = S \circ del_p(3) \circ (del_q(2) \text{ from line 16 to completion})$$

$$T_2 = S \circ (del_q(2) \text{ from line 16 to completion})$$

Note that in  $S$ , the  $search(2)$  call in  $del_q(2)$  will return *Head* and  $node_3$  if  $node_3$  is unmarked when line 16 is executed. Otherwise, *search* will restart.

Here transcripts  $T_1$  and  $T_2$  have the same prefix (the "past")  $S$ , with the only difference (in the "future") being that  $T_1$  has  $del_p(3)$  before  $q$  finishes  $del_q(2)$ . In  $T_2$ , when  $del_q(2)$  continues to completion,  $node_3$  is unmarked and is returned as *right*. Thus  $del_q(2)$  fails in  $T_2$  ( $search\_key < right.key$ ), and must be ordered before  $in_p(2)$  in  $f(T_2)$  to preserve validity. However in  $T_1$ ,  $del_p(3)$  marks  $node_3$  and the  $search(2)$  call in  $del_q(2)$  restarts its traversal. Eventually  $search(2)$  returns *Head* and  $node_2$ . In this case,  $del_q(2)$  succeeds and  $del_q(2)$  must be ordered after  $in_p(2)$  in  $f(T_1)$  to preserve validity. Then  $f$  cannot be a strong linearization function since  $del_q(2)$  must be ordered before  $in_p(2)$  for  $f(S)$  to be a prefix of  $f(T_1)$ , but then  $f(S)$  is not a prefix of  $f(T_2)$ . Transcript  $S$  is a prefix of both  $T_1$  and  $T_2$ , but  $f(S)$  is not a prefix of  $f(T_2)$  in this case.

```

Function search(search_key):
1  search_again
2  while true do
3    curr ← Head
4    (curr_next, curr_marked) ←
      Head.succ
5    repeat
6      if not curr_marked then
7        left ← curr
8        left_next ← curr_next
9        curr ← curr_next
10     if curr = Tail then
11       break
12     (curr_next, curr_marked) ←
      curr.succ
13  until curr_marked or curr.key <
      search_key
14  right ← curr
15  if left_next = right then
16    if right ≠ Tail and
      right.succ.marked then
17      goto search_again
18    else
19      return (left, right)
20  if left.succ.CAS(left_next, right) then
21    if right ≠ Tail and
      right.succ.marked then
22      goto search_again
23    else
24      return (left, right)

Function find(search_key):
32 left, right ← search(search_key)
33 if right = Tail or right.key ≠ search_key
   then
34   return false
35 else
36   return true

Function delete(search_key):
38 while true do
39   left, right ← search(search_key)
40   if right = Tail or right.key ≠
      search_key then
41     return false
42   end
43   (right_next, right_marked) ←
      right.succ
44   if not right_marked then
45     if right.succ.CAS(right_next,
      false), (right_next, true) then
46       break
47     end
48   end
49 end
50 if not left.succ.CAS(right, false),
   (right_next, false) then
51   left, right ← search(right.key)
52 end
53 return true

Function insert(search_key):
54 new_node ← new Node(search_key)
55 while true do
56   left, right ← search(search_key)
57   if right ≠ Tail and right.key =
      search_key then
58     return false
59   end
60   new_node.succ ← (right, false)
61   if left.succ.CAS(right, false),
      (new_node, false) then
62     return true
63   end
64 end

```

■ Figure 1 Harris's Linked List.

#### 4 A strongly linearizable linked list

In this section we describe modifications to Harris's linked list to yield a strongly linearizable variant. The modified algorithm (Figure 2) uses the same node object as in Harris's algorithm. In addition, the list elements are still sorted by their keys, and the list uses *Head* and *Tail* sentinels in the same manner as the original.

The largest modification can be seen in the *search* method. The changes guarantee different search conditions: at the last shared memory step when executing *search*,

1.  $left.key \leq search\_key < right.key$ ,
2. *left* is unmarked, and
3.  $left.next = right$ .

Recall that in Harris's list the condition guaranteed that  $left.key < search\_key \leq right.key$  and that  $right$  is also unmarked. Similar to Harris's implementation, if  $left$  is returned with  $left.key = search\_key$ , then the linked list contains  $search\_key$  when the search conditions are true; if  $left.key < search\_key$  then the linked list does not contain  $search\_key$  when the search conditions are true.

Since  $search$  can only exit on line 74, the last shared memory step in  $search$  is either line 67 or line 77, when  $curr.succ$  is read. If  $search$  exits, we know that  $left.key \leq search\_key < right.key$  (by line 72) and this was true at the last shared memory step (since  $key$  does not change). In addition, we know that  $left = curr$  was unmarked at the last shared memory step (by line 73), and that  $right = curr\_next$  was adjacent to  $left$ . Thus the search conditions are true. Observe that at every execution of line 67 or line 77, it is known whether it is the last shared memory step; when  $curr.succ$  is read, all values used in the exit conditions for  $search$  are known. Thus, events after the last shared of memory step of  $search$  do not influence which nodes are returned. This is the crux of why the new implementation is strongly linearizable.

The other methods are nearly identical to Harris's counterparts; the methods check whether a key is in the linked list by looking at the  $left$  node. One noteworthy change is that  $SLdelete$  no longer attempts to excise the marked node. This is because  $left$  is now the node to delete, and the predecessor of  $left$  is not readily available to "swing" the pointer to  $right$ .

We define the following terms to use in the proofs below. Let  $T$  be a transcript containing operations on  $O$ , an implementation of the algorithm in Figure 2. At time  $t$ , we say that a node  $v$  is reachable if either  $v = Head$ , or there exists a reachable node  $u$  such that  $u.next = v$ . A node  $v$  is *pre-inserted* if it was initialized in line 88 of  $SLinsert$ , but has not been an argument of a successful  $CAS$  operation in line 94.

For a transcript  $T$  containing operations on  $O$ , we say that at time  $t$ , the *interpreted value* of  $O$  is the sorted sequence of keys of all unmarked, reachable nodes excluding  $Head$  and  $Tail$ . Intuitively, the interpreted value describes the keys that are currently "in" the linked list. To prove strong linearizability, we will show that any operation that linearizes at time  $t$  should behave as if it is acting on a linked list with the keys in the interpreted value at  $t$  (Lemmas 7- 9). In addition, we show that the interpreted value at time  $t$  is consistent with the operations that have linearized before  $t$  (Lemma 10).

For the proof of strong linearizability, we assume without loss of generality that an operation  $op$  responds at the time of its last shared memory operation, i.e. if line  $x$  of  $op$  is the last shared memory operation,  $time_T(op_x) = rsp(op)$ . Note that the response of an operation is uniquely determined by the time of its last shared memory operation.

We define the function  $pt(op)$  for any operation  $op$  in a transcript  $T$  on a linked list outlined in Figure 2 in the following way:

1. If  $op$  is a successful  $SLinsert$  operation, then  $pt(op)$  is the time at which the  $CAS$  operation in  $SLinsert$  succeeds. That is,  $pt(op) = time_T(op_{61})$ .
2. If  $op$  is a successful  $SLdelete$  operation, then  $pt(op)$  is the time at which the  $CAS$  operation in  $SLdelete$  succeeds. That is,  $pt(op) = time_T(op_{45})$ .
3. If  $op$  is a failed  $SLinsert$  or  $SLdelete$ , or an  $SLfind$  operation, then  $pt(op) = rsp(op)$  (i.e. at its last shared memory step).
4. Otherwise,  $op$  is pending in  $T$  and  $pt(op) = \infty$ .

```

Function search(search_key):
65   while true do
66     curr ← Head
67     (curr_next, curr_marked) ←
        Head.succ
68     while true do
69       if not curr_marked then
70         start ← curr
71         start_next ← curr_next
72         if curr.key ≤ search_key <
            curr_next.key then
73           if not curr_marked then
74             return (curr, curr_next)
75           break
76       curr ← curr_next
77       (curr_next, curr_marked) ←
        curr.succ
78     while curr_marked do
79       curr ← curr_next
80       (curr_next, curr_marked) ←
        curr.succ
81     start.succ.CAS((start_next, false),
        (curr, false))

Function SLinsert(search_key):
88   new_node ← Node(search_key)
89   while true do
90     left, right ← search(search_key)
91     if left.key = search_key then
92       return false
93     new_node.succ ← (right, false)
94     if left.succ.CAS((right, false),
        (new_node, false)) then
95       return true

Function SLdelete(search_key):
96   while true do
97     left, right ← search(search_key)
98     if left.key ≠ search_key then
99       return false
100    if left.succ.CAS((right, false), (right,
        true)) then
101      return true

Function SLfind(search_key):
102  left, right ← search(search_key)
103  if left.key ≠ search_key then
104    return false
105  else
106    return true

```

■ **Figure 2** A Strongly linearizable linked list.

For any complete  $SLinsert$ ,  $SLdelete$  or  $SLfind$  operation  $op$ , note that  $pt(op)$  corresponds to the execution of an atomic operation in  $op$ . Thus if  $op_1, op_2 \in T$ ,  $pt(op_1) \neq \infty$ ,  $pt(op_2) \neq \infty$  and  $op_1 \neq op_2$ , then  $pt(op_1) \neq pt(op_2)$ . Also note that  $pt(op) \in [time_T(inv(op)), time_T(rsp(op))]$ .

Let  $\mathcal{T}$  be the set of all transcripts on an implementation of the algorithm in Figure 2. For all  $T \in \mathcal{T}$ , define a sequential history  $f(T)$  such that for all  $op_1, op_2 \in \Gamma(T)$  with  $pt(op_1) \neq \infty$  and  $pt(op_2) \neq \infty$ ,  $op_1 \xrightarrow{f(T)} op_2$  if and only if  $pt(op_1) < pt(op_2)$ . By the above observation that two different operations map to different times by  $pt$ , the history  $f(T)$  is unambiguous.

The following four claims show that the invariants (e.g. the linked list is always sorted) maintained by Harris's implementation hold for the modified implementation as well. The proofs of these lemmas are postponed to Appendix B.

- **Lemma 2.** *A marked node's succ field never changes.*
- **Lemma 3.** *Keys are strictly sorted; For any two nodes  $v_1$  and  $v_2$ , if  $v_1.next = v_2$  then  $v_1.key < v_2.key$ .*
- **Corollary 4.** *The linked list never contains duplicate keys.*
- **Lemma 5.** *All unmarked, not pre-inserted nodes are reachable.*
- **Lemma 6.** *Consider a search call that returns and let  $t$  be the last time line 67 or line 77 is executed. If  $curr.key < search\_key < curr\_next.key$  at  $t$ , then the interpreted value does not contain  $search\_key$  at  $t$ . Otherwise, if  $curr.key = search\_key$ , the interpreted value contains  $search\_key$  at  $t$ .*



**Proof.** Since *search* returns,  $curr\_marked = false$  was read on time  $t$ . Suppose  $curr.key < search\_key < curr\_next.key$ . Since *curr* is unmarked at  $t$ , by Lemma 5, it is reachable and *curr\_next* is also reachable. To show a contradiction, suppose that *search\_key* is in the interpreted value at time  $t$ . Consider the sequence of nodes

$$v_1, \dots, v_k, v_{k+1}, \dots, v_m$$

where  $v_1 = Head$ ,  $v_k = curr$ ,  $v_{k+1} = curr\_next$ ,  $v_m = Tail$  and  $v_i.next = v_{i+1}$  for all  $i < m$ . The sequence contains all reachable nodes at time  $t$ , thus  $i \notin \{k, k+1\}$  exists such that  $v_i.key = search\_key$ . However, if such an  $i$  existed then the linked list is not strictly sorted and Lemma 3 is violated.

Now suppose that  $curr.key = search\_key$ . Then the interpreted value at  $t$  contains *search\_key* since *curr* is unmarked. ◀

► **Lemma 7.** *A  $SLfind(k)$  operations fails if and only if the interpreted value does not contain  $k$  at  $pt(SLfind(k))$ .*

**Proof.** A *SLfind* fails if  $left.key \neq search\_key$ , and we know that either  $left.key = search\_key$  or  $left.key < search\_key < right.key$ . Then at  $pt(SLfind(k))$  the interpreted value does not contain  $k$  by Lemma 6.

For the converse, *SLfind* succeeds if  $left.key = search\_key$ . Similar to above, Lemma 6 implies that the interpreted value contains  $k$  at  $pt(SLfind(k))$ . ◀

► **Lemma 8.** *A  $SLdelete(k) = op$  operation fails if and only if the interpreted value does not contain  $k$  at  $pt(op)$ .*

**Proof.** When *op* fails, by the same reasoning as in the proof of Lemma 7, the interpreted value does not contain  $k$  at  $pt(op)$ .

Suppose *op* succeeds, meaning  $pt(op) = time_T(op_{100})$ , and the *CAS* on line 100 succeeds. This implies that *left* is unmarked at  $time(op_{100})$ , therefore  $left.key$  is in the interpreted value at this time. Since  $left.key = k$ , the interpreted value contains  $k$ . ◀

► **Lemma 9.** *An  $SLinsert(k) = op$  operation fails if and only if the interpreted value contains  $k$  at  $pt(op)$ .*

**Proof.** When *op* fails, by the same reasoning as in the proof of Lemma 7, the interpreted value contains  $k$  at  $pt(op)$ .

Suppose *op* succeeds, meaning  $pt(op) = time(op_{94})$ , and the *CAS* on line 94 succeeds. This implies that  $left.succ = (right, false)$  at  $time(op_{94})$ , therefore both *left* and *right* are reachable at this time. By Lemma 2 the interpreted value does not contain  $k$ . ◀

► **Lemma 10.** *The interpreted value contains  $k$  at time  $t$  if and only if there exists a successful insert  $SLinsert(k) = op_{in}$  such that  $pt(op_{in}) < t$  and no successful delete  $SLdelete(k) = op_{del}$  exists such that  $pt(op_{in}) < pt(op_{del}) < t$ .*

**Proof.** Suppose  $op_{in}$  with  $pt(op_{in}) < t$  exists such that no delete  $op_{del}$  exists with  $pt(op_{in}) < pt(op_{del}) < t$ . By Lemma 5, the interpreted value contains  $k$  after  $pt(op_{in})$ . To show a contradiction, suppose that the interpreted value at  $t$  does not contain  $k$ . A node is not in the interpreted value if it is not reachable, or it is marked. However, only marked nodes are unreachable (when it is not pre-inserted), thus the node containing  $k$  must have been marked between  $pt(op_{in})$  and  $t$ . However, nodes are only ever marked when the *CAS* on line 100 succeeds, with  $left.key = k$ . Such a successful *CAS* corresponds to a  $op_{del}$  operation with  $pt(op_{in}) < pt(op_{del}) < t$ , yielding a contradiction.

## 28:10 Strongly Linearizable Linked List and Queue

To show the converse, first suppose that no successful  $SLinsert(k) = op_{in}$  operation exists such that  $pt(op_{in}) < t$  in  $T$ . It is clear that the interpreted value does not contain  $k$  at  $t$  by parsing the code; the only method which initializes a new node with  $search\_key$  is  $SLinsert$ , and only a successful  $CAS$  on line 94 will make the node reachable. Now suppose that there exists a successful  $op_{del}$  such that  $pt(op_{in}) < pt(op_{del}) < t$  for any successful  $SLinsert(k)$  operation  $op_{in}$ . At  $pt(op_{del})$ , a reachable, unmarked node with key  $k$  is marked. There is only one such node at  $pt(op_{del})$  by *Corollary 4*. To show a contradiction, suppose that at  $t$ , the interpreted value contains  $k$ ; a reachable, unmarked node with key  $k$  exists. The interpreted value does not contain  $k$  immediately after  $pt(op_{del})$ , thus a new node was inserted by a successful  $CAS$  in  $SLinsert$  in  $(pt(op_{del}), t)$ . However, such a  $CAS$  corresponds to a successful  $SLinsert$  operation with  $search\_key = k$ . ◀

► **Theorem 11.** *The linked list implementation in Figure 2 is strongly linearizable;  $f$  is a linearization function for  $O$ , and  $f$  is prefix preserving.*

**Proof.** For an operation  $op$  on  $O$ , Lemmas 7, 8 and 9 show that  $op$  responds in a way that is consistent with the interpreted value of  $O$  at  $pt(op)$ . By Lemma 10, at any  $pt(op)$ , the interpreted value contains  $k$  if and only if a successful  $SLinsert(k)$  linearized before  $pt(op)$  with no successful  $SLdelete(k)$  that linearized between  $pt(op)$  and the insert. Therefore,  $f(T)$  is a linearization of the interpreted history  $\Gamma(T)$ .

Consider step  $t$  of  $T$  and operation  $op \in \Gamma(T)$  where  $pt(op) = t$ . Then

1. operation  $op$  is a successful  $SLinsert$  operation and  $t$  is when a successful  $CAS$  on line 94 is executed
2. operation  $op$  is a successful  $SLdelete$  operation and  $t$  is when a successful  $CAS$  on line 100 is executed
3. operation  $op$  is either a  $SLfind$  operation, a failed  $SLinsert$  operation, or a failed  $SLdelete$  operation and  $t$  is when  $op$  last executes line 67 or line 77. It is completely determined by step  $t$  whether  $t$  is the last execution of line 67 or line 77; all values used in the exit condition of  $search$  on lines 72 and 73 are known by  $t$ . Furthermore, the values used in the exit conditions for a failed  $SLinsert$  (line 98) and a failed  $SLdelete$  (line 91) are known by  $t$ .

At step  $t$  it is determined what operation  $op$  satisfies  $pt(op) = t$ . Therefore, if  $S$  is a prefix of  $T$ , then  $f(S)$  is a prefix of  $f(T)$ . ◀

We prove that the algorithm in Figure 2 is lock-free. For any operation  $op$ ,  $op$  finishes within a finite number of steps after  $pt(op)$ . Therefore, it suffices to show that if a process  $p$  takes infinitely many steps during a method call, then infinitely many operations have linearized.

The  $succ$  field of a reachable node is only changed by a  $CAS$  operation. We will call such successful  $CAS$  operations an *update* to the linked list. Note that the  $CAS$  in  $search$  may succeed, but if  $start\_next = curr$  then  $start.succ$  does not change and this is not an update. A successful  $CAS$  in  $search$  is an update if marked nodes were made unreachable by the operation. Thus the number of updates by  $search$  is upperbounded by the number of marked nodes, i.e. the number of successful  $SLdelete$  that have linearized. A successful  $SLinsert$  does a single update, and unsuccessful  $SLinsert$  and  $SLdelete$  do not update the linked list.

► **Lemma 12.** *The search method is lock-free.*

We prove this lemma in Appendix B.

► **Theorem 13.** *The linked list implementation in Figure 2 is lock-free.*

**Proof.** It is clear that since *search* is lock-free by Lemma 12, *SLfind* is lock-free.

Without loss of generality, consider a *SLdelete* execution that lasts at least  $k$  iterations (of the loop in *SLdelete*). For every iteration,  $left.key \neq search\_key$  and the *CAS* in *SLdelete* must have failed. However  $left.succ = (right, false)$  when last read in *search*. Thus an update occurred between when  $left.succ$  was read and *CAS* failed. Then at least  $k/2$  successful *SLinsert* or *SLdelete* have linearized. This implies that if infinitely many steps are taken by a process executing *SLdelete*, then infinitely many successful *SLinsert* or *SLdelete* have linearized. ◀

## 5 Michael and Scott's queue is not strongly linearizable

Michael and Scott's queue [11] is a linked list based algorithm. The node object consists of two fields; *value* and *next*, where *next* is a pair containing a node (the next node in the linked list) and a sequence number. The *value* contains the element that was enqueued, and the *next* field is a *CAS* object. The queue maintains *Head* and *Tail* *CAS* objects which are both initialized to  $(v_{dummy}, 0)$ , where  $v_{dummy}$  is a dummy node. The sequence numbers are present to prevent the ABA problem, but for brevity we will commonly refer to *Head*, *Tail* and the *next* field as if they refer to nodes, instead of a node-sequence number pair.

At a high level, enqueued elements are appended to the *Tail*, and the *Head* is set to *Head.next* to dequeue elements. The *Head* refers to the last element that was dequeued (hence the dummy node) to simplify cases when the queue is empty. The queue also prevents *Tail* from lagging behind *Head*. This ensures that freeing a dequeued node (by the call to *free* on line 130) does not corrupt the data structure.

The linearization point of *enqueue* is when a new node is successfully appended to the list (at *CAS* success on line 113). For a successful *dequeue*, it is when *Head* changes to *Head.next* (line 129); for a failed *dequeue* it is when *null* was found when reading *start.next* (line 121).

The linearization points for *enqueue* and successful *dequeue* are already strong linearization points; similar to successful *insert* and *delete* for Harris's implementation, they correspond to a successful *CAS*, after which the methods return. However, the linearization point of a failed *dequeue* operation is not a strong linearization point. If *Head* was changed between the execution of line 121 (the linearization point) and line 122, then the *dequeue* restarts and may no longer fail. Similar to a failed *delete* in Harris's linked list, events after the linearization point can change the response of the operation. We have only examined one particular linearization point for a failed *dequeue*, but this observation can be extended to prove that the implementation is not strongly linearizable similar to the proof of Lemma 1. We postpone the proof the next lemma to Appendix C.

► **Lemma 14.** *Michael and Scott's queue (Figure 3) is not strongly linearizable.*

## 6 A strongly linearizable queue

As previously stated, Michael and Scott's queue is not strongly linearizable only because the linearization point for a failed *dequeue* is not strongly linearizable. The problem was that because of the condition on line 122, events after the linearization point could change the response of a failed *dequeue* operation.

## 28:12 Strongly Linearizable Linked List and Queue

```

Function enqueue(x):
107   node ← new Node(x)
108   while true do
109     (end, endc) ← Tail
110     (next, nextc) ← end.next
111     if (end, endc) = Tail then
112       if next = null then
113         if end.next.CAS((next, nextc),
114           (node, nextc + 1)) then
115           break
116       else
117         Tail.CAS((end, endc), (next,
118           endc + 1))

117 Function dequeue():
118   while true do
119     (start, startc) ← Head
120     (end, endc) ← Tail
121     (next, nextc) ← start.next
122     if (start, startc) = Head then
123       if start = end then
124         if next = null then
125           return false
126         Tail.CAS((end, endc), (next,
127           endc+1))
128       else
129         value ← next.value
130         if Head.CAS((start, startc),
131           (next, startc+1)) then
132           free(start)
133         return true

```

■ **Figure 3** Michael and Scott's lock-free queue.

A simple modification that will yield a strong linearizable queue is to remove the condition on line 122. The linearization point remains the same; it is when *null* is read on line 121. Intuitively, if *null* is read then *start* refers to the last node, and so should *Head* and *Tail* (thus *start* = *end*). This means that the method commits to failing exactly when *null* is read, and at this point the queue is empty (recall that *Head* points to the last element dequeued). Not checking whether *Head* changed since its last read (line 122) will not corrupt the queue since if *Head* changed, the *CAS* on line 129 will fail. In our proof that the algorithm in Figure 4 is strongly linearizable, we disregard line 157 (the *free* function call). Thus, if the method exits on line 158, the *CAS* on line 156 is the last shared memory operation. Calling *free* does not affect strong linearizability. For the proofs below, we assume that no ABAs occur due to our use of sequence numbers. Let *T* be a transcript containing operations on *O*, an implementation of the queue. We define whether a node is reachable identically as with the linked list; a node is reachable at time *t* if it can be obtained by traversing the

```

Function dequeue():
146   while true do
147     (start, startc) ← Head
148     (end, endc) ← Tail
149     (next, nextc) ← start.next
150     if start = end then
151       if next = null then
152         return false
153       Tail.CAS((end, endc), (next, endc+1))
154     else
155       value ← next.value
156       if Head.CAS((start, startc), (next, startc+1)) then
157         free(start) // ignored in the proof of strong linearizability
158       return true

```

■ **Figure 4** Dequeue operation of a strongly linearizable lock-free queue.

sequence of nodes from  $Head$  (let no node be reachable if  $Head = null$ ). We say that  $Head$  (or  $Tail$ ) is *incremented* if  $Head = (v, \_)$  is changed to  $(v.next, \_)$ , where  $v$  is a node, and  $v.next \neq null$ . Suppose at time  $t$ , we have the following sequence of nodes

$$v_1, \dots, v_k$$

where  $Head = v_1$ ,  $v_{i-1}.next = v_i$  for  $i \in \{2, \dots, k\}$  and  $v_k.next = null$ . The sequence is well defined if  $Head \neq null$ , (guaranteed by Lemma 16). We define the interpreted value of  $O$  at time  $t$  as the following sequence of numbers:

$$v_2.value, \dots, v_k.value.$$

Once again, we assume without loss of generality an operation  $op$  responds at its last shared memory operation.

The linearization function  $pt(op)$  for an operation  $op$  in  $T$  on an implementation of the queue in Figure 4 (with *enqueue* from Figure 3 is defined as follows:

1. If  $op$  is an *enqueue* operation, then  $pt(op)$  is the time at which the *CAS* on line 113 succeeds.
2. If  $op$  is a *dequeue* operation, then  $pt(op)$  is the first time at which *null* is read on line 149 or the *CAS* on line 156 succeeds.
3. Otherwise,  $op$  did not perform its last shared memory step and  $pt(op) = \infty$ .

Let  $\mathcal{T}$  be the set of all transcripts on an implementation of the queue in Figure 4. For all  $T \in \mathcal{T}$ , we define a sequential history  $f(T)$  that orders operations according to  $pt$ , and excludes all operations  $op$  with  $pt(op) = \infty$ . That is, for  $pt(op_1) \neq \infty$  and  $pt(op_2) \neq \infty$ ,  $op_1, op_2 \in T$ ,  $op_1 \xrightarrow{f(T)} op_2$  if and only if  $pt(op_1) < pt(op_2)$ . Again,  $f(T)$  is unambiguous since for every operation  $op \in \Gamma(T)$  such that  $pt(op) \neq \infty$ , the step of  $T$  at  $pt(op)$  is performed by  $op$ .

The following two lemmas describe invariants of the queue which are used to argue strong linearizability. Their proofs can be found in Appendix D.

► **Lemma 15.** *Tail is always reachable.*

► **Lemma 16.** *Head is never null, and is only ever incremented.*

► **Lemma 17.** *Suppose the interpreted value of the queue is  $(x_1, \dots, x_k)$  at a *CAS* call on line 113. Let  $t$  be the time at which this *CAS* call occurs. If the *CAS* call succeeds, then the interpreted value immediately after  $t$  is  $(x_1, \dots, x_k, value)$ , where *value* is the argument of *enqueue*.*

**Proof.** Suppose the *CAS* operation on line 113 succeeds, meaning  $end.next = null$  at  $t$ . By Corollary 23,  $Tail.next$  is also *null* when it was assigned to  $end$ . Since  $Tail$  is only ever incremented, and  $Tail.next = null$  up until the *CAS* operation,  $Tail$  and  $end$  refer to the same node at  $t$ . By Lemma 15  $end$  is a reachable node. Since  $end.next = null$ ,  $end$  is the last reachable node by Observation 24. Thus the interpreted value immediately after  $t$  is  $(x_1, \dots, x_k, value)$ . ◀

► **Lemma 18.** *Suppose that at time  $t$ ,  $start.next = null$  is read on line 121. Then the interpreted value of the queue at  $t$  is empty.*

**Proof.** This follows immediately from Lemma 16 and Corollary 23; since  $start.next = null$  and  $Head$  is only ever incremented,  $Head$  cannot have changed between when it was assigned to  $start$  and when  $start.next$  was read. ◀

► **Lemma 19.** *If  $start.next = null$  was read on line 149 then  $start = end$ .*

**Proof.** As seen in the proof of Lemma 18,  $Head$  and  $start$  reference the same node when  $start.next = null$  was read. Since  $Tail$  is always reachable (Lemma 15) and  $Head$  references the last reachable node,  $Tail$  references the same node as  $Head$  during the execution of lines 147 and 148. Thus when  $Tail$  is read on line 148, it references the same node as  $start$ . ◀

► **Lemma 20.** *Suppose at time  $t$  the interpreted value of the queue is  $(x_1, \dots, x_k)$ , and a successful CAS on line 156 is executed. Then immediately after time  $t$ , the interpreted value of the queue is  $x_2, \dots, x_k$ .*

**Proof.** By Lemma 16, upon CAS success the  $Head$  changes to  $head.next$ . ◀

► **Theorem 21.** *The queue in Figure 3 but with the dequeue function from Figure 4 is strongly linearizable and lock-free.*

**Proof.** Michael and Scott showed that their queue (in particular *enqueue*) is lock-free [11]. The only method that was changed is *dequeue*, and the only change was the removal of a condition which could have caused another iteration of the loop. Thus *dequeue* is still lock-free.

We now show that the queue is strongly linearizable. For an *enqueue* and a successful *dequeue* on  $O$ , Lemmas 17 and 20 ensure that both operations modify the interpreted value appropriately at their linearization points. Lemma 18 guarantees that for a failed *dequeue*, the interpreted value is empty at its linearization point. Thus,  $f(T)$  is a linearization of the interpreted history  $\Gamma(T)$ .

Consider a step  $t$  of  $T$  and an operation  $op \in \Gamma(T)$  where  $pt(op) = t$ . Then

1. operation  $op$  is an *enqueue* operation and  $t$  is when a successful CAS on line 113 is executed
2. operation  $op$  is a *dequeue* operation and  $t$  is when a successful CAS on line 156 is executed
3. operation  $op$  is a *dequeue* operation and  $t$  is when *null* is read on line 149. Notice that by Lemma 19, reading *null* guarantees that  $op$  will fail.

At step  $t$  it is determined what operation  $op$  satisfies  $pt(op) = t$ . Therefore, if  $S$  is a prefix of  $T$ , then  $f(S)$  is a prefix of  $f(T)$ . ◀

## 7 Discussion

We proved that Harris's linked list and Michael and Scott's queue, two well-known lock-free data structures, are not strongly linearizable. We have carefully analyzed where the strong linearizability breaks, and gave modifications to derive strongly linearizable variants.

An observation we made on the original data structures is that an operation exists such that the response of the operation was not determined by the time of its linearization point. Using this observation, we constructed transcripts where events after an operation's linearization point changed the linearization order of the past. It is currently unknown whether such observations directly imply that a data structure is not strongly linearizable.

Simple modifications addressing these operations were given but the proofs of strong linearizability were non-trivial. The minor changes required gives hope for future work on deriving strongly linearizable data structures.

We hope that our insights can be used to develop techniques either for determining whether other linearizable implementations are strongly linearizable, or to derive strongly linearizable implementations from linearizable ones. For example, interpreted values have been used

to great effect in this paper and by others [4, 12] in proving whether implementations are strongly linearizable, albeit in an ad-hoc manner. A future direction could be to formalize the concept of interpreted values, then develop techniques around it.

---

## References

- 1 H. Attiya, A. Castañeda, and D. Hendler. Nontrivial and universal helping for wait-free queues and stacks. *Journal of Parallel and Distributed Computing*, 121:1–14, 2018.
- 2 O. Denysyuk and P. Woelfel. Wait-freedom is harder than lock-freedom under strong linearizability. In *International Symposium on Distributed Computing*, pages 60–74, 2015.
- 3 M.J. Fischer, N.A. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the Association for Computing Machinery*, 32(2):374–382, 1985. doi:db/journals/jacm/FischerLP85.html, 10.1145/3149.214121.
- 4 G. Giakkoupis, M.J. Giv, and P. Woelfel. Efficient randomized DCAS. In *Proceedings of the 53rd Symposium on Theory of Computing*, pages 1221–1234. ACM, 2021.
- 5 W. Golab, L. Higham, and P. Woelfel. Linearizable implementations do not suffice for randomized distributed computation. In *Proceedings of the 43rd Symposium on Theory of Computing*, pages 373–382, New York, NY, USA, 2011. Association for Computing Machinery.
- 6 T.L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing*, pages 300–314, Berlin, Heidelberg, 2001. Springer-Verlag.
- 7 M. Helmi, L. Higham, and P. Woelfel. Strongly linearizable implementations: possibilities and impossibilities. In *Proceedings of the 2012 ACM symposium on Principles of Distributed Computing*, pages 385–394, 2012.
- 8 M.P. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, January 1991.
- 9 M.P. Herlihy and J.M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.
- 10 M. Loui and H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing research*, 4:163–183, 1987.
- 11 M.M. Michael and M.L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 267–275, New York, NY, USA, 1996. Association for Computing Machinery.
- 12 S. Ovens and P. Woelfel. Strongly linearizable implementations of snapshots and other types. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019*, pages 197–206. ACM, 2019.

## A Proofs of claims from Section 3

► **Lemma 1.** *The linked list implementation by Harris (Figure 1) is not strongly linearizable.*

**Proof.** We denote  $node_i$  as the node containing key  $i$ ,  $in_p(x)$  as a transcript of an insert of key  $x$  by process  $p$  and  $del_p(x)$  as a transcript of a delete of key  $x$  by  $p$ .

Consider the following transcripts for processes  $p$  and  $q$ :

$$S = in_p(3) \circ (del_q(2) \text{ to the first execution of line 15}) \circ in_p(2)$$

$$T_1 = S \circ del_p(3) \circ (del_q(2) \text{ from line 16 to completion})$$

$$T_2 = S \circ (del_q(2) \text{ from line 16 to completion})$$

To show a contradiction, assume that the algorithm is strongly linearizable. Then there exists a strong linearization function  $f$  for  $\{S, T_1, T_2\}$ . In  $S$ , the insert of 3 happens before every other operation, thus  $in_p(3)$  is the first operation in  $f(S)$ . Either  $del_q(2)$  is ordered

## 28:16 Strongly Linearizable Linked List and Queue

before  $in_p(2)$  in  $f(S)$ , or it is not. We consider both cases below. For  $del_q(2)$ , we can see from tracing the code that during its execution up to line 15,  $node_3$  is assigned to  $right$ .

Suppose  $del_q(2)$  linearizes prior to  $in_p(2)$  in  $S$ . Then we have

$$f(S) = in_p(3) \circ del_q(2) \circ in_p(2).$$

In  $T_1$ ,  $p$  completes  $del_p(3)$  before  $q$  finishes its *delete*. Note that while  $p$  executes  $del_p(3)$ ,  $q$  takes no steps. By the post-conditions of *search*,  $node_3$  is assigned to  $right$  on line 39 of  $del_p(3)$ . This  $right$  will fail the if condition on the next line. During the execution of  $S$ , no node is marked, and  $right.next$  does not change after its insertion. Thus, the if condition on line 44 is satisfied during the execution of  $del_p(3)$ . For the same reason,  $del_p(3)$  will succeed its *CAS* call on line 45, and  $node_3$  is marked. Therefore, when  $q$  resumes its execution of  $del_q(2)$ ,  $right$  ( $node_3$ ) will be marked. The condition on line 15 succeeds, and  $q$  restarts *search*.

From the post-conditions of *search*, line 56 of  $in_p(2)$  assigns  $Tail$  to  $right$ , which will fail the if condition on line 57. No shared memory operations have completed between  $p$ 's execution of lines 56 and 61, thus the *CAS* call on line 61 will succeed, and  $in_p(2)$  will succeed (return *true*). Therefore, when  $q$  executes another *search*,  $right = node_2$ , and the *CAS* will succeed on line 45, thus  $del_q(2)$  will succeed.

From our assumption that  $f$  is strongly linearizable, if  $del_q(2)$  linearizes before  $in_p(2)$  in  $S$ , then  $del_q(2)$  also linearizes before  $in_p(2)$  in  $T_1$ . We have,

$$f(T_1) = in_p(3) \circ del_q(2) \circ in_p(2) \circ del_p(3).$$

However, this sequential history is not valid since  $del_q(2)$  succeeds when no node in the linked list contains 2. This contradicts the assumption that  $del_q(2)$  linearizes before  $in_p(2)$ .

Now suppose that  $del_q(2)$  does not linearize before  $in_p(2)$  in  $S$ ; either  $del_q(2)$  linearizes after  $in_p(2)$  in  $S$ , or it does not linearize in  $S$ . That is,

$$f(S) = in_p(3) \circ in_p(2) \circ del_q(2) \text{ or } f(S) = in_p(3) \circ in_p(2).$$

No *delete* operation occurs in  $T_2$  other than  $del_q(2)$ , thus when  $q$  continues  $del_q(2)$ , it fails the if condition on line 15 and returns  $node_3$  as  $right$ . The search key (2) and the key of  $right\_node$  (3) are different, and  $del_q(2)$  returns *false*.

From our assumption that  $f$  is strongly linearizable, if  $f(S) = in_p(3) \circ in_p(2) \circ del_q(2)$ , then  $f(T_2) = in_p(3) \circ in_p(2) \circ del_q(2)$ . Otherwise, since  $del_q(2)$  is complete in  $T_2$ ,  $del_q(2)$  linearizes in  $T_2$  but not in  $S$ . Since  $del_q(2)$  still linearizes after  $in_p(2)$  in  $T_2$ ,  $f(T_2)$  is the same as above. However, this sequential history is not in the sequential specification; a *delete* method fails when the key being deleted is in the linked list. This contradicts the assumption that  $del_q(2)$  does not linearize before  $in_p(2)$  in  $S$ .

Both cases contradict the assumption that  $f$  is a strong linearization function. Therefore, no strong linearization function can be defined over  $\{S, T_1, T_2\}$ , and Harris's linked list implementation is not strongly linearizable. ◀

## B Proofs of claims from Section 4

► **Lemma 2.** *A marked node's succ field never changes.*

**Proof.** A *succ* field is only changed by the three *CAS* operations and on line 94. It is clear that none of the three *CAS* operations will succeed if a node is marked. A node when constructed is by default unmarked, and when changed on line 94 it is left unmarked. Thus *new\_node* on line 94 is always unmarked when it changes. ◀



► **Lemma 3.** *Keys are strictly sorted; For any two nodes  $v_1$  and  $v_2$ , if  $v_1.next = v_2$  then  $v_1.key < v_2.key$ .*

**Proof.** Initially, there are only *Head* and *Tail* where  $Head.key < Tail.key$ , thus the lemma is true. The *next* field of a node is only ever altered on lines 94 and 95 in *SLinsert*, and on line 86 in *search*. We show that if the lemma is true before each of the listed operations, then it is true after the operation. For the lines corresponding to *CAS* operations, we assume that the call succeeds since otherwise no change takes place.

Consider an *SLinsert* operation. The *search* call return only if  $left.key \leq search\_key < right.key$ . Line 94 is executed if  $left.key \neq search\_key$ , thus  $left.key < search\_key < right.key$ . Then we have that after line 94,  $new\_node.next = right$  and  $new\_node.key = search\_key < right.key$ . Furthermore, if the *CAS* on line 95 succeeds,  $left.next = new\_node$  and we have already established that  $left.key < search\_key = new\_node.key$ .

For the *CAS* in *search*, consider the sequence of nodes

$$v_1, v_2, \dots, v_k$$

where  $v_1 = start$ ,  $v_k = curr$  at the execution of the *CAS*, and  $v_{i+1}$  is  $v_i.next$  when it was read on line 67, 80 or 84. The sequence is well defined since *curr* is set to *curr\_next* after *curr.succ* is read. After the *CAS* succeeds,  $start.next = curr$ . Since we assume that the lemma is true before the *CAS* succeeds,  $v_1.key < v_k.key$ , thus it is still maintained. ◀

► **Corollary 4.** *The linked list never contains duplicate keys.*

**Proof.** If it contained duplicate keys Lemma 3 is violated. ◀

► **Lemma 5.** *All unmarked, not pre-inserted nodes are reachable.*

**Proof.** Reachability is only affected by the *CAS* on line 95, when a node is inserted, and on line 86, when  $start.next$  is changed to *curr*.

At the *CAS* success on line 95, *left* is unmarked, and is thus reachable. The *next* field of *left* is *new\_node*, hence *new\_node* is reachable. The node *right* is still reachable since  $new\_node.next = right$ . For the *CAS* in *search*, we want to show that no unmarked node exists “between” *start* and *curr* on line 86. Again consider the sequence of nodes

$$v_1, v_2, \dots, v_k$$

where  $start = v_1$ ,  $v_k = curr$  and  $v_{i+1}$  is the node seen when  $v_i.next$  is read. Note that for  $1 < i < k$ ,  $v_i$  was seen to be marked when  $v_{i-1}.next$  was read. Thus by Lemma 2, such  $v_i$  are marked at the execution of the *CAS*. We show that at the *CAS* execution,  $v_i.next = v_{i+1}$  for all  $1 \leq i < k$ , proving that all nodes “between” *start* and *curr* are all marked.

Note that  $start\_next = v_2$ ; when  $v_1$  was assigned to *start*,  $curr\_next = v_2$  was assigned to *start\_next* (line 71). At *CAS* execution,  $start\_next = start.next$  since it succeeds, so  $v_1.next = v_2$  at this time. For all other  $v_i$ ,  $v_i.next = v_{i+1}$  at the *CAS* success since after  $v_i.succ$  has been read (and was seen to be marked), it cannot change by Lemma 2. ◀

► **Lemma 12.** *The search method is lock-free.*

**Proof.** Consider the first inner loop in *search*. After every iteration of the loop, *curr* advances down the linked list by one node. For every *search\_key*,  $Head.key < search\_key < Tail.key$ . The linked list is strictly sorted by their keys, and *Tail* is always reachable. Thus the exit condition on line 72 is always met before *Tail* is assigned to *curr*. Consider an

## 28:18 Strongly Linearizable Linked List and Queue

execution of the loop that lasts more than  $k$  iterations. At iteration  $k$ , the variable  $curr$  is not  $Tail$ , and  $curr$  has advanced down the linked list  $k$  times. Since initially the linked list consists only of  $Head$  and  $Tail$ , at least  $k$   $SLinsert$  operations have linearized.

Now consider the second inner loop in  $search$ . After every iteration of the loop,  $curr$  advances down the linked list by one node. The  $Tail$  node is unmarked, thus by the time  $curr = Tail$ , the exit condition of the loop is met. By similar reasoning as above, if the loop execution lasts more than  $k$  iterations, then at least  $k$  successful  $SLinsert$  have linearized.

Finally, consider the outer loop execution that lasts at least  $k > 2$  iterations. For clarity, we denote the node assigned to variable  $x$  on iteration  $j$  as  $x_j$ . For an iteration  $i < k$ , suppose that the  $CAS$  on line 81-86 fails;  $start.succ \neq (start\_next, false)$ . Then  $start.succ$  was modified by an update between the  $CAS$  execution and when  $start.succ$  was read on line 67 or 77. Now suppose that the  $CAS$  succeeds. Observe that on line 81,  $start.key \leq search\_key < curr_i.key$ , and at this point  $start$  is unmarked and therefore reachable; no reachable node with key between  $start.key$  and  $curr_i.key$  exists. However on iteration  $i + 1$ , when line 73 is executed to exit from the first inner loop,  $curr_{i+1}$  is marked. One of the following updates must have occurred between the  $CAS$  on iteration  $i$  and when  $curr_{i+1}.marked$  was read:

1.  $start_i$  was marked, or
  2. A node  $v$  with  $start.key \geq v.key < search\_key$  was inserted.
- Otherwise,  $curr_{i+j} = start_i$ ,  $start_i$  is unmarked and  $search\_key < curr_{i+1}.key$ , meaning that  $search$  should return on iteration  $i$ . An update occurred for all cases, thus for  $k$  iterations of the outer loop,  $k$  updates occurred. For  $k$  updates, at least  $k/2$  successful  $SLinsert$  or  $SLdelete$  operations have linearized. Then we have that if a process takes infinitely many steps (infinitely many iterations of any loop) while executing the  $search$  function, then infinitely many successful  $SLinsert$  or  $SLdelete$  operations have linearized. ◀

## C Proofs of claims from Section 5

► **Lemma 14.** *Michael and Scott's queue (Figure 3) is not strongly linearizable.*

**Proof.** Again, we denote  $node_i$  as a node containing value  $i$ . Similarly,  $deq_p()$  as a transcript of a *dequeue* operation by process  $p$ , and  $enq_p(x)$  as a transcript of an *enqueue* operation of value  $x$  by process  $p$ . For clarity, when tracing the execution of different processes, we denote variable  $var$  from  $p$ 's execution  $var_p$ .

Consider the following transcripts for processes  $p$  and  $q$ :

$$S = (deq_p() \text{ to the first execution of line 121}) \circ enq_q(1) \circ enq_q(2)$$

$$T_1 = S \circ deq_q() \circ (deq_p() \text{ from line 122 to completion})$$

$$T_2 = S \circ (deq_p() \text{ from line 122 to completion})$$

To show a contradiction, suppose that the algorithm is strongly linearizable with a strong linearization function  $f$  over  $\{S, T_1, T_2\}$ . When we trace the execution outlined by  $S$ ,  $v_{dummy}$  is assigned to  $start_p$  and  $end_p$ , and  $null$  is assigned to  $next$  (lines 119-121). In addition,  $enq_q(1)$  and  $enq_q(2)$  append their respective nodes to the linked list.

Now consider the rest of the execution in  $T_1$ . The  $deq_q()$  operation terminates successfully.  $Head$  is assigned to  $start_q$  and  $node_2$  is assigned to  $end_q$  (thus  $start_q \neq end_q$ ).  $Head$  never changed ( $Head = v_{dummy}$ ) thus the  $CAS$  on line 129 succeeds. When  $deq_p()$  resumes its execution, it fails the condition on line 122 (since  $Head$  was changed by  $deq_q()$ ) and restarts.

During the next iteration of the loop,  $Head$  does not change, as  $q$  does not execute any operations. In addition,  $start_q = node_1$  and  $end_q = node_2$  are read on lines 119-120. The  $CAS$  on line 129 is therefore reached, and succeeds to change  $Head$  to  $node_2$ .

For  $f(T_1)$  to be a linearization of  $T_1$ ,  $deq_p()$  cannot be ordered first. Otherwise, a *dequeue* operation succeeded (as we saw when tracing the execution) when no *enqueue* operation preceded before it. As  $S$  is a prefix of  $T_1$ , for  $f(S)$  to be prefix-preserving,  $f(S)$  also cannot start with  $deq_p()$ . Now, we consider the transcript  $T_2$ . Continuing from our tracing of  $S$ ,  $start_p = v_{dummy}$  and  $next_p = null$ .  $Head$  has yet to change (is still  $v_{dummy}$ ), thus the condition on line 122 passes. The next two if statements (line 123-124) is also satisfied, and the  $deq_p()$  fails.

In order to preserve validity,

$$f(T_2) = deq_p() \circ enq_q(1) \circ enq_q(2).$$

Since  $f(S)$  is a prefix of  $f(T_2)$  ( $S$  is a prefix of  $T_2$ , and  $f$  is prefix-preserving, and  $S$  contains complete operations  $enq_q(1)$  and  $enq_q(2)$ ),

$$f(S) = deq_p() \circ enq_q(1) \circ enq_q(2).$$

However  $f(S)$  cannot start with  $deq_p()$ , yielding a contradiction. ◀

## D Proofs of claims from Section 6

► **Observation 22.** *For a node  $v$ , if  $v.next \neq null$ , then  $v.next$  does not change.*

**Proof.** Initially,  $v.next = null$ . The *next* field of a node is only ever altered in *enqueue* by a  $CAS$  in line 113. Such a  $CAS$  only succeeds if  $v.next = null$ , and after the  $CAS$ ,  $v.next \neq null$ . ◀

► **Corollary 23.** *For a node  $v$ , if  $v.next = null$ , then  $v.next$  never changed since  $v$  was constructed.*

**Proof.** Otherwise  $v.next$  was changed to a node  $u$  between  $v$ 's initialization and when  $v.next = null$ . However by Lemma 22  $v.next$  can never change back to  $null$ . ◀

► **Observation 24.** *If node  $v$  is reachable and  $v.next = null$ , then  $v$  is the last reachable node.*

► **Lemma 25.** *Tail is only ever incremented; if  $Tail = node$ , then  $Tail$  only ever changes to  $node.next$  where  $node.next \neq null$ .*

**Proof.**  $Tail$  is only ever changed through  $CAS$  operations on lines 153 and 116. We show that if such a  $CAS$  succeeds on either line,  $Tail$  is incremented.

For line 116, a successful  $CAS$  changes  $Tail$  from  $(end, endc)$  to  $(next, endc + 1)$ , where  $next \neq null$  by the prior if condition. By Observation 22, at the time of  $CAS$  success,  $end.next = (next, nextc)$ . Next, we show that  $end.next = (next, nextc)$  on line 153. We know that  $next \neq null$  since the if condition on line 151 was not satisfied (otherwise the method call would not reach line 153). By the if condition on line 150,  $start = end$ , meaning  $start.next = end.next = next$  at  $CAS$  success (line 153) by Observation 22. ◀

► **Observation 26.** *If  $Head$  changes from node  $v$  to node  $u$ , then  $v.next = u$  when  $Head$  was changed.*

## 28:20 Strongly Linearizable Linked List and Queue

**Proof.** *Head* is only altered by a successful *CAS* on line 156, and it is changed to *next*. Suppose the *CAS* operation succeeds and changes *Head* from *v* to *u*. Then, *v* was assigned to *start* on line 147 and  $u \neq \text{null}$  was assigned to *next* on line 149. By Observation 22,  $v.\text{next} = u$  at the time of *CAS* success. ◀

► **Lemma 15.** *Tail is always reachable.*

**Proof.** Initially, *Tail* and *Head* refer to the same node. By induction, we show that the lemma continues to hold even after *Tail* or *Head* is change by a successful *CAS* operation. For every *CAS* operation that changes *Tail* or *Head*, suppose that *Tail* is reachable up until the *CAS* operation. By Lemma 25, any time *Tail* changes it is changed to *Tail.next*. Since *Tail* is reachable, *Tail.next* is also reachable. Thus, successful *CAS* operations on line 153 and 116 maintain the lemma.

We first prove that if the *CAS* on line 156 is reached, then  $\text{next} \neq \text{null}$ . To show a contradiction, suppose otherwise. By the induction hypothesis, *Tail* is reachable during the execution of lines 147-148. By the assumption that  $\text{next} = \text{null}$  and Corollary 23,  $\text{start.next} = \text{null}$  on lines 147-148 and *start* is the last node in the list in this duration. If  $\text{Head} = \text{start}$  on line 148, then *start* is the last reachable node and *start* is assigned to *end* on line 148 (since *Tail* is reachable at this line). Then *dequeue* exits on line 152 and line 156 is never reached, yielding a contradiction. Otherwise,  $\text{Head} \neq \text{start}$  on line 148. *Head* must have changed since its assigned to *start*, but  $\text{Head} \neq \text{null}$  for *Tail* to be reachable. Then we have that *Head* changed from *start* to a node *u*. However, this contradicts Lemma 26; *Head* changed from  $\text{start} \neq \text{null}$  to  $u \neq \text{null}$ , but  $\text{start.next} = \text{null} \neq u$ .

We now have that  $\text{next} \neq \text{null}$  at the *CAS* operation on line 156. By Lemma 26, if the *CAS* operation succeeds, then *Head* changes to *Head.next*. The only way such a change can make *Tail* unreachable from *Head* is if  $\text{Tail} = \text{Head}$  at *CAS* success. To show a contradiction, suppose that this is the case. For the *CAS* on line 156 to succeed, *Head* does not change after it was assigned to *start* on line 147. *Tail* was assigned to *end* on line 148, and *start* was evaluated to not equal *end* on line 150. The node pointed to by *Tail* must have changed for *Tail* and *Head* to reference the same node at the *CAS*, but such a change can only be an increment by Lemma 25. Thus *Tail* was unreachable from *Head* prior to the *CAS* execution, which contradicts the inductive hypothesis. ◀

► **Lemma 16.** *Head is never null, and is only ever incremented.*

**Proof.** If *Head* was *null*, then *Tail* would be unreachable. Thus if *Head* changes, then it changes from a node *v* to a node *u*. By Lemma 26, *Head* is then only ever incremented. ◀

# Recoverable and Detectable Fetch&Add

Liad Nahum ✉ 

Department of Computer Science, Ben Gurion University, Beer Sheva, Israel

Hagit Attiya ✉ 

Department of Computer Science, Technion, Haifa, Israel

Ohad Ben-Baruch ✉ 

Department of Computer Science, Ben Gurion University, Beer Sheva, Israel

Danny Hendler ✉ 

Department of Computer Science, Ben Gurion University, Beer Sheva, Israel

---

## Abstract

The emergence of systems with non-volatile main memory (NVRAM) increases the need for persistent concurrent objects. Of specific interest are recoverable implementations that, in addition to being robust to crash-failures, are also *detectable*. Detectability ensures that upon recovery, it is possible to infer whether the failed operation took effect or not and, in the former case, obtain its response.

This work presents two recoverable detectable *Fetch&Add* (FAA) algorithms that are *self-implementations*, i.e. use only a *fetch&add* base object, in addition to read/write registers. The algorithms target two different models for recovery: the *global-crash* model and the *individual-crash* model. In both algorithms, operations are *wait-free* when there are no crashes, but the recovery code may block if there are repeated failures. We also prove that in the individual-crash model, there is no implementation of recoverable and detectable FAA using only read, write and *fetch&add* primitives in which all operations, including recovery, are lock-free.

**2012 ACM Subject Classification** Theory of computation → Shared memory algorithms

**Keywords and phrases** Multi-core algorithms, persistent memory, non-volatile memory

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2021.29

**Funding** Supported by the Israel Science Foundation (grant number 380/18).

## 1 Introduction

Systems with byte-addressable *non-volatile main memory* (NVRAM) combine the performance benefits of conventional main memory with the durability of secondary storage. The emergence of commercial systems with NVRAM increased the interest in the *crash-recovery* model, in which failed processes may be resurrected after they crash. For this model, the goal is to design *recoverable concurrent objects* (also called *persistent* or *durable*): Objects that are made robust to crash-failures by allowing operations to recover from such failures.

Persistent objects were hand-crafted for specific data structures, e.g., [15, 26, 28, 29]. Other work introduces general mechanisms to port existing algorithms and make them persistent, e.g., by using transactional memory [6, 9, 24, 27], universal constructions [5, 8, 10], or for specific families of algorithms [4, 11, 13]. These transformations rely on strong primitives such as *compare&swap*, while their non-persistent counterparts may use only weaker primitives, in terms of their level in the *consensus hierarchy* [21].

An alternative approach is to design persistent *self-implementations*, in which a recoverable operation is implemented by using non-recoverable instances of *the same primitive operation*, possibly with additional reads and writes on shared variables. Self-implementations can be used to implement high-level persistent objects by plugging them within existing object implementations. A recoverable implementation is *detectable* [15] if, in addition to being robust to crash-failures, it ensures that it is possible to infer, upon recovery, whether the failed operation took effect or not and, in the former case, obtain its response.



© Liad Nahum, Hagit Attiya, Ohad Ben-Baruch, and Danny Hendler;  
licensed under Creative Commons License CC-BY 4.0

25th International Conference on Principles of Distributed Systems (OPODIS 2021).

Editors: Quentin Bramas, Vincent Gramoli, and Alessia Milani; Article No. 29; pp. 29:1–29:17



Leibniz International Proceedings in Informatics

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

For example, a detectable *compare&swap* (CAS) was used in a CAS-based generic transformation that makes algorithms recoverable [4]. Detectable self-implementations were presented in prior works for read, write, *test&set*, and CAS objects [2, 4]. An important primitive, which is useful in several data structures, is *fetch&add*, whose consensus number is two, i.e., it allows exactly two processes to solve consensus [21].

**Our Contributions.** This paper presents two detectable self-implementations of *Fetch&Add* (FAA). The first algorithm is for the *global-crash* model, where the whole system crashes and a single process is responsible for the recovery of all failed operations. The second algorithm is for the *individual-crash* model, where some processes may crash while others might not, and each process is responsible for restoring its own state in a consistent manner. However, recovering processes may have to wait for other processes to make progress with their operations or with their recovery code, and may never complete if such progress does not occur. We also prove that in the individual-crash model, there is no lock-free implementation of recoverable FAA objects from read, write and *fetch&add* primitives. In other words, for every detectable self-implementation of an FAA object in this model, either the FAA operation or the recovery code is not lock-free.

Our implementations satisfy *nesting-safe recoverable linearizability* (NRL) [2]. NRL was originally defined for the individual-crash model, and we extend it to the global-crash model. NRL implies that, following recovery, an implemented (higher-level) recoverable operation is able to complete its invocation of a base-object operation and obtain its response.

**Related Work.** The notion of detectability was presented in [14, 15]. A strict version of detectability, named *nesting-safe recoverable linearizability* (NRL), was formally defined by [2]. It requires that each process complete its operation and obtain its response before invoking another operation even when it incurs crash-failures. There are NRL self-implementations of recoverable *read*, *write*, *test&set* and *compare&swap* [2]. In a sense, FAA is a more complex object since, in most cases, an FAA operation has a unique place in history where it must be linearized, and its response is also unique based on this linearization point. Unlike FAA, in *compare&swap* and *test&set* we have more freedom in choosing where to linearize operations. For example, we can linearize a *test&set* operation that returns 1 at any point after the first operation in the linearization order. Tracking this unique linearization point of every crashed FAA operation and restoring the response based on it is the core challenge of our self-implementations. It is known that there is no wait-free self-implementation of a detectable *test&set* object [2]. Both this proof and our impossibility proof for FAA employ valency arguments that rely on the loss of response values incurred by processes following crash-failures.

Golab [16] defined *recoverable consensus* and revised the consensus hierarchy in the presence of crash-recovery failures, for both the individual-crash model and the global-crash model. (Recall that the *wait-free consensus hierarchy* [21] ranks shared objects according to the maximum number of processes that can use them to solve consensus; *fetch&add* and *test&set* are at level 2 of the hierarchy.) Golab showed that *test&set* drops to level 1 for the individual-crash model, if the number of crashes is unbounded. Our impossibility result can be adapted to prove an analogous result for *fetch&add*.

Other correctness conditions were suggested for shared objects that tolerate crash-recovery failures. *Strict linearizability* [1] treats the crash of a process as a response, either successful or unsuccessful, to the interrupted operation. *Persistent atomicity* [20] is similar to strict linearizability, but allows an operation interrupted by a failure to take effect before the next

invocation of the same process, possibly after the failure. Both conditions ensure that the state of an object is consistent after a crash. *Recoverable linearizability* [5] ensures object implementations can be composed, but may compromise program order following a crash. They also present a universal construction using *compare&swap* in the individual-crash model.

In the *recoverable mutual exclusion* problem (RME) [18], processes may undergo individual crashes during the execution of a mutual exclusion protocol. This paper also presents an RME algorithm using only reads and writes whose *remote memory references* (RMRs) complexity is logarithmic in the number of processes,  $n$ . There is an RME algorithm for the cache-coherent (CC) model [17], using *fetch&store* and *compare&swap*, which incurs  $O(\frac{\log N}{\log \log N})$  RMRs. An RME algorithm with the same asymptotic RMR complexity was proposed for the distributed shared memory (DSM) and CC models [25]; it uses *fetch&store*. There is also an RME algorithm using only *compare&swap* and *fetch&increment*, with constant amortized RMR complexity [7]. In the global crash model, RME can be solved in a constant number of RMRs [19].

## 2 Model of Computation

We consider a system with  $N$  processes,  $p_0, \dots, p_{N-1}$ , which communicate by applying atomic *primitive* operations (also called *primitives*) to *shared* base objects; the primitives applied are read, write and *fetch&add*. All shared base objects are non-volatile. The *state* of each process comprises its program counter and local variables; all local variables are volatile.<sup>1</sup> A *configuration* consists of the states of all processes and the values of all shared base objects. Two configurations  $C_1$  and  $C_2$  are *indistinguishable* to a set of processes  $P$ , denoted  $C_1 \stackrel{P}{\sim} C_2$ , if every process in  $P$  has the same state in  $C_1$  and  $C_2$ , and all shared objects hold the same values in  $C_1$  and  $C_2$ . The state of the system changes when processes take *steps*, each of which is a local computation followed by an atomic operation on one shared object (*ordinary step*), a *crash* step, or a *recovery* step.

Base objects and primitives are used to implement more complex objects, by specifying an algorithm for each operation of the implemented object using primitives on base objects. In this work, we implement a recoverable *Fetch&Add* (FAA) object that is detectable. The *sequential specification* of *fetch&add* contains all sequences of  $FAA(v)$  operations in which each operation returns the sum of the arguments of all preceding  $FAA$  operations. We refer to the recoverable detectable operation that is implemented as *Fetch&Add*, while *fetch&add* is the primitive operation supported by the system, which can be applied to non-volatile variables.

An *execution*  $\alpha$  is an alternating sequence of configurations and steps that follow the algorithm. An execution  $\alpha$  is *crash-free* if it contains no crash steps, and hence, also no recovery steps. If a step  $s$  is possible in a configuration  $C$  at the end of a finite execution  $\alpha$ , then the sequence obtained by appending  $s$  to  $\alpha$  is also an execution, denoted  $\alpha \circ s$ , whose final configuration is denoted  $C \circ s$ . Let  $\alpha$  be an execution ending in configuration  $C$  and let  $p$  be a process. If  $p$ 's last step in  $\alpha$  is a crash step, then the only step by  $p$  that is possible in  $C$  is a recovery step.

<sup>1</sup> We assume the simple mode of *shared caches*, where updates to the persistent shared base objects are immediate. There are standard ways to port algorithm from this model to more realistic models capturing existing architectures [23].

Process  $i$  invokes an operation  $Op$  on an object with an *invocation step*, and it *completes* with a *response step*, in which  $Op$ 's response is stored to a local (volatile) variable of process  $i$ . The return value is lost if process  $i$  crashes, unless process  $i$  writes it to a non-volatile variable before the crash, thereby *persisting* it. An operation  $Op$  is *pending* if it is invoked but not yet completed; each process has at most one operation pending.

We consider two models of recovery from crashes. In the *individual crash* model, at any point during an execution, each process can incur a crash that resets all its local variables to arbitrary values, but preserves the values of shared non-volatile variables. Recovery is done by the same crashed process, so each *recoverable operation*,  $Op$ , is associated with a *recovery function*,  $Op.RECOVER$ . In the *global crash* model, at any point during the execution, a global crash can occur that resets all local variables of all processes, but preserves the values of shared non-volatile variables. The recovery is done by a single process that is responsible for recovering all processes. In this case, we have a global *RECOVER* function; once *RECOVER* completes, processes can resume their execution.

One component of the processes' state is  $Seq[N]$ . For each process  $i$ ,  $Seq[i]$  holds the sequence number of its current *FAA* operation. Before an *FAA* operation is invoked,  $Seq[i]$  is incremented by 1 by the process (or the system), externally to the operation itself. This is essential in our model for determining, upon recovery, the progress made by *FAA* operations before the crash. It has been proved [3] that detectable algorithms must keep auxiliary state, provided from outside the operation, either by the system or by the caller of the operation via arguments or a non-volatile variable accessible by them. This auxiliary state is used to infer where the failure occurred.

We say that an operation  $op_1$  *precedes* another operation  $op_2$  *in the real-time order* of an execution  $\alpha$ , if  $op_1$  completes before  $op_2$  is invoked. Informally, a crash-free execution  $\alpha$  is *linearizable* [22] if we can order all completed operations, as well as a subset of the pending operations, in a way that preserves the real-time order of the operations, and the return values respect the sequential specification of the *Fetch&Add* object.

An execution  $\alpha$  satisfies *nesting-safe recoverable linearizability* (NRL) if the execution obtained by removing all crash and recovery steps from  $\alpha$  is linearizable. NRL implies *detectability* [15], namely, a recovering operation has an appropriate response.

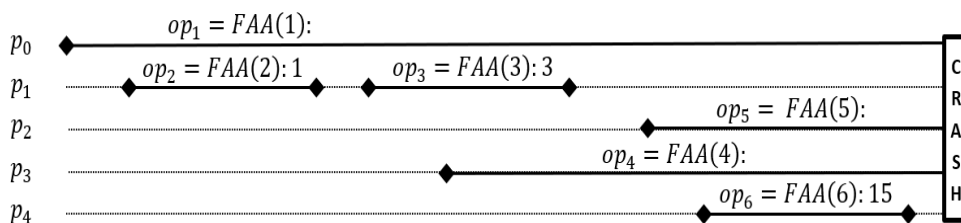
In our algorithms, the operations are *wait-free*, i.e., the execution of an operation by a process that does not incur a crash (global or individual) is guaranteed to complete in a finite number of its steps, regardless of the steps or crashes of other processes. An algorithm is *lock-free* if, whenever a set of processes take a sufficient number of steps and none of them crashes, then it is guaranteed that one of them will complete its operation.

### 3 FAA Implementation in the Global-Crash Model

Our algorithm implements a recoverable detectable *FAA* operation using a *fetch&add* base object of unbounded size. An *FAA* operation receives, as its single argument, a value  $val$  that should be added to the global unbounded counter. *FAA* atomically adds  $val$  and returns the previous value of the counter.

The challenge in implementing a recoverable and detectable *FAA* operation is that some return values may be lost upon a crash, if they were not persisted. Such operations may have already affected the global counter, i.e., the return values of other operations. Upon recovery, it is necessary to figure out the return values of incomplete operations so that all operations (completed and pending) can be linearized. For example, in Figure 1, all operations, including pending ones, must be linearized after the system recovers from the





■ **Figure 1** Challenges in detectable  $FAA$  implementation. We use the notation  $op = FAA(V_1) : V_2$  to denote an  $FAA$  invocation with argument  $V_1$  that returns  $V_2$  as its response. An empty  $V_2$  means that  $op$  is pending when the crash occurs.

crash. This is because all of them succeeded in adding their argument to the counter, and thus, have impacted other operations. Note that  $p_0$ 's pending operation must be linearized right before the first operation of  $p_1$  that is affected by it. On the other hand, we can choose where to linearize the pending operations of  $p_2$  and  $p_3$ , as long as they are linearized before the operation of  $p_4$ , which is affected by both. We track and identify this information using a combination of sequence numbers and vector timestamps, as explained next.

Each  $FAA$  operation by a process has a strictly increasing sequence number; thus, it is uniquely identified by the pair  $\langle \text{process id, sequence number} \rangle$ . Linearizing  $FAA$  operations is facilitated with a global unbounded *fetch&add* base object. This object holds a *vector timestamp* with the sequence numbers of the last  $FAA$  operation of each process. Each  $FAA$  operation updates this object, and obtains a timestamp that precisely tracks the  $FAA$  operations affecting it. This tracking allows to ensure consistency with completed operations whose arguments were already added to the implemented counter. Since sequence numbers are unbounded, the *fetch&add* base object is unbounded as well.

The algorithm also uses two arrays: the first holds details of each  $FAA$  operation and the other helps to persist the operation and ensure consistency during recovery.

An  $FAA$  operation has three stages: it is first announced in the first array; then, its sequence number is updated at the *fetch&add* base object; finally, its vector timestamp, which can be used to compute its return value, is persisted in the second array.

Recovery is done by a single process, which is responsible for determining the return values of all operations that were pending when the crash occurred, depending on which stage they were at. If the operation did not modify the base *fetch&add* object, its return value indicates that it should be re-executed. If the operation wrote to the second array, it has been persisted and its return value is known. The main challenge is to determine the return value of an operation that modified the base *fetch&add* object but did not persist its return value in the second array. To handle these operations, the recovery process first orders the persisted operations, and then finds a place to insert each of these operations, so that its return value is consistent with the return values of the persisted operations.

### 3.1 Shared Data Structures

The algorithm maintains a global array  $Seq[N]$  holding the sequence numbers of the current  $FAA$  operation of each process, all initially zero.

The  $Res[N]$  array is used only in case of a crash, and it holds return values for all processes after the recovery ends;  $\perp$  indicates that the  $FAA$  operation did not take effect and should be re-executed. All components in  $Res[N]$  are initially  $\perp$ .

29:6 Recoverable and Detectable Fetch&Add

0	...	$N - 1$	$N$	...	$2N - 1$	...	$kN$	...	$(k + 1)N - 1$	...
1	0	0	1	0	1	0	0	0	0	0

■ **Figure 2** The organization of  $W$ : Process 0’s assigned bits, 0 and  $N$ , store  $11_{(2)}$ , while process  $N - 1$ ’s assigned bits,  $N - 1$  and  $2N - 1$ , store  $10_{(2)}$ .

The  $TotalContrib[N][\infty]$  array stores the intermediate sums of contributions of each process.  $TotalContrib[i][k]$  holds the sum of the additions of all FAA operations executed by process  $i$  until and including the  $k$ -th operation. All components in  $TotalContrib[i]$  are initially zero.

► **Definition 1.** A vector timestamp (VTS) holds  $N$  sequence numbers, one for each process.  $VTS_1 < VTS_2$  if  $VTS_2$  is larger than or equal to  $VTS_1$  in each component and  $VTS_1 \neq VTS_2$ . Two VTSs are comparable if one of them is larger than the other, in the  $<$  order.

Each FAA  $op$  by process  $i$  has an associated  $VTS_{op}$ , indicating the sequence numbers of the FAA operations that precede it:  $VTS_{op}[j]$  is the sequence number of the last FAA operation by process  $j$  that precedes  $op$ .

An  $OpVTS[N][\infty]$  array stores the persistence information of operations of each process.  $OpVTS[i][k]$  holds the VTS associated with process  $i$ ’s  $k$ -th FAA operation; all components in  $OpVTS[i]$  are initially  $\perp$ .

These data structures are accessed only with read and write primitives: process  $i$  reads and writes only the  $i$ -th component of each of them, while the recovery process reads and writes all the components.

**The base fetch&add object.** The algorithm uses an unbounded-size register  $W$  that is accessed by all processes with  $fetch\&add$  primitives.  $W$  holds for each process  $i$  the sequence number of the last FAA operation process  $i$  executed.  $W$ ’s value is numeric and is changed with  $fetch\&add$ . Since the sequence numbers stored in  $W$  are unbounded, they are stored and manipulated as follows (see Figure 2):

The sequence number of process  $i$  is stored in bits  $k * N + i$ ,  $k \in [0, \infty)$ . To increment its sequence number in  $W$ , process  $i$  has to add a value that will set and clear corresponding bits, considering the previous stored value, so the new value is stored correctly in  $i$ ’s bits. For example, assume process 0’s bits store  $101_{(2)}$ , i.e, bits 0 and  $2N$  are set. After the increment, process 0’s bits should store  $110_{(2)}$  so bit  $N$  should be set and bit  $2N$  should be cleared, which is done by applying  $fetch\&add$  with argument  $2^N - 2^{2N}$ .<sup>2</sup>

Process  $i$  uses two functions to manipulate  $W$ :

**ReadVTS:** applies  $W.faa(0)$  in order to read  $W$ ’s content and returns a VTS of the  $N$  sequence numbers stored in it.

**IncrementSeqAndGetVTS:** increments process  $i$ ’s sequence number stored in  $W$  using a single primitive atomic  $fetch\&add$ . The function returns a VTS out of the previous value of  $W$  returned by the  $fetch\&add$ .

<sup>2</sup> Although the number of bits assigned in  $W$  to each process is unbounded, the number of bits that are actually used by process  $i$  can be determined according to the value of  $Seq[i]$

### 3.2 Code Description

The pseudo code appears in Algorithm 1. Recall that  $Seq[i]$  is incremented by 1, before the FAA operation is invoked. Process  $i$  starts an FAA operation by reading its prior total contribution, until the previous operation (Line 2). In Line 3, it declares the new operation by storing the new total contribution in  $TotalContrib[i][Seq[i]]$ ; this value is the sum of  $prevTotalContrib$  and  $val$ , the argument of the current FAA operation by process  $i$ . In Line 4, process  $i$ 's sequence number stored in  $W$  is incremented using  $IncrementSeqAndGetVTS$ , which returns a vector,  $VTS$ , of  $N$  sequence numbers. These sequence numbers indicate, for each process, the last FAA operation prior to process  $i$ 's operation  $\langle i, Seq[i] \rangle$ . Line 5 stores  $VTS$  in  $OpVTS[i][Seq[i]]$ , thereby persisting the operation. Finally, the FAA operation returns  $ComputeVal(VTS)$ . As shown in Algorithm 1,  $ComputeVal$  with argument  $VTS$  sums  $TotalContrib[p][VTS[p]]$ , for all processes  $p$ , i.e.,  $p$ 's total contribution until and including the operation whose sequence number is stored in  $VTS[p]$ .

► **Definition 2.** *An operation is invisible if it did not perform the fetch&add primitive in Line 4. An invisible operation does not update its sequence number in  $W$  and does not affect other processes. An operation is effective if it performed the fetch&add primitive in Line 4, and updated its sequence number in  $W$ . An effective operation is persisted if it performed Line 5, so its  $VTS$  is persisted and its return value can be calculated based on it.*

Incrementing  $Seq[i]$  before an FAA operation is invoked allows to distinguish, during recovery, between an invisible operation that was just invoked and a prior persisted operation.

The *RECOVER* function is executed by a single process. In Line 10, it collects all persisted operations in  $persistedOps$ . These are the operations whose  $VTS$ s appear in  $OpVTS$ . Then, it creates an order,  $L_0$ , of  $persistedOps$  according to the order of their  $VTS$ s (Line 11). Note that the  $VTS$ s of persisted operations are comparable. The main loop (Line 13) recovers the last operation of each process  $p$ , depending on its type; The sequence number of the operation,  $seq_p$ , is read from  $W$ , using  $ReadVTS(W)[p]$  (Line 14). If  $seq_p$  is smaller than  $Seq[p]$  (Line 15), then process  $p$  did not execute the *fetch&add* in Line 4 before the crash. Thus,  $op_p = \langle p, Seq[p] \rangle$  is invisible and should be re-executed. Otherwise (Line 17), if  $VTS$  is in  $OpVTS[p][seq_p]$ , then process  $p$  executed Line 5 and persisted its operation by storing the corresponding  $VTS$  in  $OpVTS[p][seq_p]$ . In this case,  $ComputeVal$  is applied to the corresponding  $VTS$  in order to compute the return value. We note that processes that did not invoke any FAA operation before the crash, with  $Seq[p] == 0$ , are skipped. The remaining case is when the operation is effective but non-persisted. In this case, *RECOVER* extends the order created in the previous iteration of the loop (initially,  $L_0$ ) by inserting the operation into it. This is done with *InsertOperationIntoOrder*, explained next.

The function *InsertOperationIntoOrder* gets an operation  $op_p = \langle p, seq_p \rangle$  to insert,  $L_0$ , the ordering of persisted operations, and  $L_{k-1}$ , the ordering after the previous effective non-persisted operation was inserted. The function finds the *barrier* of  $op_p$ , which is the smallest operation in  $L_0$  that follows  $op_p$ , i.e., with  $VTS_{barrier}[p] = seq_p$ . The function creates  $L_k$  by placing  $op_p$  as the immediate predecessor of its barrier; if no such barrier operation exists, then  $op_p$  is placed at the end of  $L_{k-1}$  to create  $L_k$ . This ensures consistency with the persisted operations.

In Lines 25-30, the function derives the  $VTS$  corresponding to  $op_p$  from its immediate predecessor in  $L_k$ . If there is no immediate predecessor, then  $op_p$  is the first operation that applied *fetch&add* to  $W$ , and  $VTS_p$  is defined as all zeros (Line 26). Otherwise, let  $op_t = \langle t, seq_t \rangle$  be the immediate predecessor. The function sets  $VTS_p$  to be  $op_t$ 's corresponding vector time stamp,  $OpVTS[t][seq_t]$ , except that its  $t$ -th component is set to  $seq_t$ , indicating that  $op_p$  is the immediate successor of  $op_t$ . In Line 30, the function persists  $VTS_p$  in  $OpVTS[p][seq_p]$ , to safeguard against future crashes, and returns  $L_k$ .

■ **Algorithm 1** Recoverable detectable FAA, for the global-crash model.

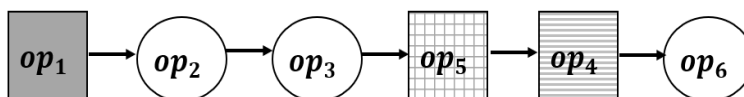
---

```

1: procedure FAA(val) ▷ executed by process i
2:   prevTotalContrib  $\leftarrow$  TotalContrib[i][Seq[i]-1]
3:   TotalContrib[i][Seq[i]]  $\leftarrow$  prevTotalContrib + val ▷ Store current total contrib
4:   VTS  $\leftarrow$  IncrementSeqAndGetVTS(W, i)
5:   OpVTS[i][Seq[i]]  $\leftarrow$  VTS ▷ Store VTS
6:   return ComputeVal(VTS) ▷ Compute return value
7: procedure ComputeVal(VTS)
8:   return  $\sum_{\text{process } p} \textit{TotalContrib}[p][\textit{VTS}[p]]$ 
9: procedure RECOVER() ▷ executed by recovery process
10:  persistedOps  $\leftarrow$  all operations whose VTSs appear in OpVTS
11:  L0  $\leftarrow$  persistedOps ordered according to their VTSs
12:  prevOrder  $\leftarrow$  L0
13:  for p from 0 to N - 1 do
14:    seqp  $\leftarrow$  ReadVTS(W)[p]
15:    if seqp < Seq[p] then ▷ invisible operation
16:      Res[p] =  $\perp$ 
17:    else if OpVTS[p][seqp]  $\neq$   $\perp$  then ▷ persisted operation
18:      Res[p] = ComputeVal(OpVTS[p][seqp])
19:    else ▷ effective but non-persisted operation
20:      prevOrder  $\leftarrow$  InsertOperationIntoOrder( $\langle p, seq_p \rangle$ , L0, prevOrder)
21:      Res[p] = ComputeVal(OpVTS[p][seqp])
22: procedure InsertOperationIntoOrder( $\langle p, seq_p \rangle$ , L0, Lk-1)
23:  barrier  $\leftarrow$  smallest operation in L0 such that VTSbarrier[p] = seqp
24:  Insert  $\langle p, seq_p \rangle$  as the immediate predecessor operation to barrier in Lk-1 to get Lk.
  If there is no such barrier, append  $\langle p, seq_p \rangle$  at the end of the order to get Lk.
25:   $\langle t, seq_t \rangle$   $\leftarrow$  immediate predecessor operation to  $\langle p, seq_p \rangle$  in Lk
26:  if  $\langle t, seq_t \rangle = \perp$  then VTSp  $\leftarrow$  all N components are zeros
27:  else
28:    VTSp = OpVTS[t][seqt]
29:    VTSp[t] = seqt
30:  OpVTS[p][seqp]  $\leftarrow$  VTSp
31:  return Lk

```

---



■ **Figure 3** Linearization construction.  $L_0$  orders  $op_2, op_3, op_6$ ;  $L_1$  is obtained by adding  $op_1$ ;  $L_2$  is obtained by adding  $op_5$ ;  $L_3$  is obtained by adding  $op_4$ .

Then, *RECOVER* applies *ComputeVal* to  $OpVTS[p][seq_p]$ , in order to compute the return value. Notice that there are at most  $N$  effective non-persisted operations, at most one for each process. The function recovers the processes from 0 to  $N - 1$ . By Line 24, effective non-persisted operations with the same barrier are ordered in an ascending order of their processes' ids. Moreover, after the function finds the place for such operation, it builds its corresponding *VTS* based on the *VTS* of its immediate predecessor. This implies that the *VTS* of the immediate predecessor operation is already written when we use it.

In Lines 16, 18 and 21, the function saves the return values of each process in the *Res* array so when the processes resume, they can read their correct return values. If  $Res[i] == \perp$  then the process should re-execute the FAA operation, i.e., proceed from Line 2. Otherwise,  $Res[i]$  holds the correct return value for process  $i$ .

► **Example 3.** Consider the execution of Figure 1, ending with a crash. Assume all pending operations are effective and non-persisted. Figure 3 illustrates the order of the operations build by the *RECOVER* procedure, as follows.  $L_0$  orders the persisted operations  $\{op_2, op_3, op_6\}$  (empty circles). The effective non-persisted operations  $op_1, op_5, op_4$  which are represented by squares, executed by processes  $p_0, p_2, p_3$ , respectively, are considered in the order of their processes' identifiers. The barrier of  $op_1$  is  $op_2$ , so  $op_1$  (filled square) is placed as  $op_2$ 's immediate predecessor to obtain  $L_1$ . Next, the barrier of  $op_5$  is  $op_6$ , so  $op_5$  (squares pattern) is placed as  $op_6$ 's immediate predecessor to obtain  $L_2$ . Finally,  $op_6$  is also the barrier of  $op_4$ , so  $op_4$  (stripes pattern) is placed as  $op_6$ 's new immediate predecessor with  $op_5$  as its immediate predecessor to obtain  $L_3$ . Note that operations with the same barrier could be linearized arbitrarily, and we chose to follow an ascending order of process' ids.

By Figure 1, the return values of  $op_2, op_3, op_6$  are 1, 3, 15, respectively.  $op_1$ 's return value is 0, as it is the first operation applied to the *fetch&add* object.  $op_5$ 's return value is 6, which is the sum of the return value of its predecessor  $op_3$  and  $op_3$ 's contribution, that is, the sum of the contributions of all  $op_5$ 's preceding operations.  $op_4$ 's return value is 11, which is the sum of the return value of its predecessor  $op_5$  and  $op_5$ 's contribution.

### 3.3 Correctness Proof

Let  $\alpha$  be a crash-free execution of Algorithm 1. Each effective operation has an associated *VTS*. We order *VTS*s by coordinate-wise comparison as defined in Definition 1.

► **Lemma 4.** *The VTSs of two effective operations are comparable; furthermore, this order respects the order in which the corresponding operations executed the atomic fetch&add in Line 4, and hence, their real-time order.*

**Proof.** The lemma follows from the atomicity of the *fetch&add* performed in Line 4 and from the fact that each process executing Line 4 increments its sequence number stored in  $W$  by 1. Thus, every time Line 4 is executed, exactly one sequence number in  $W$  is changed and increased. That is, every time Line 4 is executed, the returned *VTS* is larger than its

## 29:10 Recoverable and Detectable Fetch&Add

immediate predecessor in exactly one component. Thus, comparing two *VTS*s, at least one component is larger than the other and all other components are larger or equal. Therefore they are *comparable*. Furthermore, the *VTS* of an operation  $op_2$  that executes Line 4 after another operation  $op_1$  is larger than the *VTS* of  $op_1$  by the  $\prec$  order. Consequently,  $\prec$  respects the execution's real-time order.  $\blacktriangleleft$

We linearize the effective operations in  $\alpha$  according to their *VTS*s. Call this order  $L$ . Since all persisted operations are effective, they are linearized. Non-effective operations, i.e., invisible operations, are omitted and not linearized.

► **Lemma 5.** *The return values in  $L$  respect the sequential specification of Fetch&Add, i.e., they are the sum of the arguments of all preceding operations.*

**Proof.** Let  $op'$  be an effective operation and let  $VTS_{op'}$  be its associated *VTS*. The proof is by induction on the position of  $op'$  in  $L$ . The base case is that  $op'$  is the smallest operation in the order, i.e., is the first FAA operation that executed Line 4 on  $W$ . Therefore, all components in  $VTS_{op'}$ , returned in Line 4, are zeros. For each  $i$ ,  $TotalContrib[i][0] = 0$  by the initialization of *TotalContrib*. Therefore, *ComputeVal* returns 0, which respects the sequential semantics of *Fetch&Add* because initially, the value of the implemented object is 0.

Assume the lemma holds for all operations smaller than  $op'$  in  $L$ . Assume operation  $op$  is the immediate predecessor operation to  $op'$  in  $L$ . Thus, by induction this implies that  $res_{op} = ComputeVal(VTS_{op})$  respects the sequential semantics of *Fetch&Add*. The previous value of the implemented object, before operation  $op$  is performed, is  $res_{op}$ . Assume  $op$  is executed by process  $i$ , therefore by executing Line 4,  $i$ 's sequence number is incremented by 1. Therefore,  $VTS_{op'}[i]$  is larger than  $VTS_{op}[i]$  by 1 and equals to the sequence number of  $op$ ,  $seq_i$ . All other components are equal. For each process  $p$ , let  $contrib_p$  and  $contrib'_p$  be the total contributions of  $p$ , as functions  $ComputeVal(VTS_{op})$  and  $ComputeVal(VTS_{op'})$  considered in their calculation, respectively. For each  $p \neq i$ ,  $contrib_p = contrib'_p$ . For  $i$ ,  $TotalContrib[i][seq_i] = TotalContrib[i][seq_i - 1] + val_{op}$ , therefore,  $contrib'_i = contrib_i + val_{op}$ . Therefore,  $ComputeVal(VTS_{op'})$  equals  $res_{op}$  plus the new value was added by  $op$ .

This respects the sequential semantics of *Fetch&Add* because the value of the implemented object before  $op'$  is applied to it is its value immediately before  $op$  is applied to it plus the value added by  $op$ .  $\blacktriangleleft$

Let  $\alpha'$  be an execution that ends with a global crash. The *VTS* of any persisted operation appears in  $OpVTS$ . Let  $L_0$  be the sequence of all persisted operations, ordered according to their *VTS*s.

Next consider the last *FAA* operation of each process. If it is invisible, we assign the return value  $\perp$ . If it is persisted, we assign the return value computed by *ComputeVal* on its associated *VTS*. Finally, for an effective operation that is non-persisted, we find a place among the persisted operations and extend  $L_0$ . Since each process has at most one effective non-persisted operation, we can consider them by the order of their ids. Finding a place for the  $k$ -th effective non-persisted operation yields  $L_k$  which extends  $L_{k-1}$ .

Given  $L_{k-1}$ , let  $op = \langle i, seq_i \rangle$  be the  $k$ -th effective non-persisted operation. The *barrier* of  $op$  is the smallest operation in  $L_0$  that follows  $op$ , i.e.,  $VTS_{barrier}[i] = seq_i$ .  $op$  is inserted right before its barrier in  $L_{k-1}$ , to get  $L_k$ . If there is no such operation,  $op$  is appended at the end of  $L_{k-1}$  to get  $L_k$ . Let  $L$  be the final linearization, after all effective non-persisted operations were inserted. A simple induction on  $k$  proves the next claim:

▷ **Claim 6.** Let  $op$  be the effective non-persisted operation that is inserted in  $L_k$ . Its immediate predecessor in  $L_k$  stays the same in  $L$ .

We compute the return values of effective non-persisted operation, based on the return value of the operation linearized immediately before them and its associated VTS, as follows: assume  $op'$  is an effective non-persisted operation and let  $op$  be its immediate predecessor executed by process  $i$ . The return value of  $op'$  is  $ComputeVal(VTS_{op'})$  while  $VTS_{op'}$  equals  $VTS_{op}$  except for the  $i$ -th component in which  $VTS_{op'}[i] = seq_i$ , the sequence of  $op$ .

▷ **Claim 7.** Let  $op$  be a persisted operation ordered in  $L_0$ . The operations preceding  $op$  in  $L$  are consistent with its stored  $VTS_{op}$ . That is, an operation of some process  $p$  precedes  $op$  in  $L$  if and only if its sequence number is smaller than or equal to  $VTS_{op}[p]$ .

► **Lemma 8.** *The return values in  $L$  satisfy the sequential specification of  $Fetch\&Add$ .*

**Proof.** The proof is by induction on the position of every effective operation  $op'$  in  $L$ ; note that non-effective operations are not linearized in  $L$ .

Consider the operations in  $L$ ; the base case is that  $op'$  is the first operation in  $L$ . If  $op'$  is persisted, the VTS associated with  $op'$  was stored in  $OpVTS$  before the crash. By Claim 7, for each persisted  $op'$ , the operations preceding  $op'$  in  $L$  are consistent with its stored  $VTS_{op'}$ . That is, the return value of  $op'$ , which is  $ComputeVal(VTS_{op'})$  is the sum of the arguments of all preceding operations in  $L$ .

Otherwise,  $op'$  is non-persisted without preceding operations. By Line 26, all components in  $VTS_{op'}$  are zeros. For every  $i$ ,  $TotalContrib[i][0] = 0$  by the initialization and  $ComputeVal$  returns 0, respecting the sequential specification of  $Fetch\&Add$ .

Assume the lemma holds for all operations that appear in  $L$  before  $op'$ . If  $op'$  is persisted, then the lemma holds as in the base case. Otherwise, let operation  $op$  executed by process  $t$  be the operation immediately preceding  $op'$  in  $L$ . By Claim 6, it is the same immediate predecessor that was used to build  $VTS_{op'}$ . By the assumption,  $res_{op} = ComputeVal(VTS_{op})$  satisfies the lemma. The previous value of  $W$  before  $op$  was  $res_{op}$ . By the construction in Lines 28-29,  $VTS_{op}$  and  $VTS_{op'}$  differ in the  $t$ -th component.  $VTS_{op}[t] < VTS_{op'}[t] = seq_t$ , the sequence number of  $op$ . Thus,  $ComputeVal(VTS_{op'})$  takes for each process  $k \neq t$ , the same total contribution as  $ComputeVal(VTS_{op})$  and for  $t$ , takes the total contribution of operation  $\langle t, seq_t \rangle$ . The latter is equal to the total contribution of operation  $\langle t, seq_t - 1 \rangle$  plus  $val_{op}$ . That is,  $ComputeVal(VTS_{op'})$  equals  $res_{op}$  plus the argument added by  $op$ . This respects the sequential specification because the value of  $W$  before  $op'$  should be the value before  $op$  plus the value added by  $op$ . ◀

► **Theorem 9.** *Algorithm 1 implements a recoverable detectable FAA in the global-crash model using only read, write and  $fetch\&add$  primitive operations, and satisfies NRL.*

**Complexity.** In a crash-free execution, an FAA operation executes one  $fetch\&add$  operation, a constant number of writes and  $O(N)$  reads from shared memory during the  $ComputeVal$  function. The algorithm can be modified to store the total contribution values in  $W$ , using a similar encoding scheme. This would allow to read all contributions using a single shared memory access, yielding an FAA implementation with  $O(1)$  crash-free complexity.

## 4 FAA Implementation in the Individual-Crash Model

In the individual-crash model, each process can crash individually without affecting executions of other processes, and then it also recovers individually. Thus, a process may crash and recover without the other processes being aware of this.

## 29:12 Recoverable and Detectable Fetch&Add

In addition to the data structures of Algorithm 1, we use the following data structures:  $isInRecovery[N]$  holds in the  $i$ -th entry the sequence number of the last operation during whose execution process  $i$  crashed; all entries are initially 0. We also use  $mutex$ , an RME lock implemented using only read and write primitives [18]. RME ensures that if process  $i$  crashes while holding  $mutex$ , no other process can acquire the lock until  $i$  tries to acquire it again; in this case  $i$  will succeed, i.e.,  $mutex$  will still be held by  $i$ .

The pseudo code appears in Algorithm 2.  $FAA(val)$  is identical to the one for global crashes (Algorithm 1) and is omitted. Process  $i$  manipulates  $W$  and the other data structures using the same functions as in Section 3.1.

■ **Algorithm 2** Recoverable Detectable FAA, for the individual-crash model.

---

```

32: procedure  $FAA.RECOVER(val)$  ▷ executed by process  $i$ 
33:    $seq_i \leftarrow ReadVTS(W)[i]$ 
34:   if  $seq_i < Seq[i]$  then ▷ invisible operation
35:     re-execute  $FAA(val)$ 
36:   else ▷ effective operation
37:      $isInRecovery[i] \leftarrow seq_i$ 
38:      $await(mutex.lock())$  ▷ lock robust to failures
39:     if  $OpVTS[i][seq_i] \neq \perp$  then ▷ persisted operation
40:        $mutex.release()$ 
41:       return  $ComputeVal(OpVTS[i][seq_i])$  ▷ from Algorithm 1
42:      $recoveryW \leftarrow ReadVTS(W)$ 
43:     for  $k$  from 0 to  $N-1$  do  $await(OpVTS[k][recoveryW[k]] \neq \perp$  or
        $isInRecovery[k] == recoveryW[k])$ 
44:      $Ops \leftarrow$  all operations until sequence numbers in  $recoveryW$ .
45:      $persistedOps \leftarrow$  all operations in  $Ops$ , whose  $VTS$ s appear in  $OpVTS$ 
46:      $L_0 \leftarrow$  order  $persistedOps$  according to their  $VTS$ s
47:      $InsertOperationIntoOrder(\langle i, seq_i \rangle, L_0, L_0)$  ▷ from Algorithm 1
48:      $mutex.release()$ 
49:     return  $ComputeVal(OpVTS[i][seq_i])$  ▷ from Algorithm 1

```

---

$FAA.RECOVER(val)$  is executed by each crashed process independently and includes acquiring a lock. Thus, the algorithm must consider repeated crashes during  $FAA.RECOVER$  of the same process such that a process that acquires the lock, crashes and then invokes  $FAA.RECOVER$  again (possibly several times), will eventually release the lock at the end of its recovery. Lines 33-35 have the same logic as Lines 14-16 in  $RECOVER()$  (Algorithm 1). In Line 33, the function reads the sequence number of the last operation of process  $i$ ,  $seq_i$ , from  $W$  using  $ReadVTS(W)[i]$ . In Line 34, the function checks if  $seq_i$  is smaller than  $Seq[i]$ . If it is, process  $i$  did not execute the critical  $fetch\&add$  inside  $IncrementSeqAndGetVTS$  before crashing, implying  $op_i = \langle i, seq_i \rangle$  is invisible and should be re-executed. Otherwise,  $op_i$  is effective and can be persisted or non-persisted. In Line 37, the function declares that  $\langle i, seq_i \rangle$  is in recovery by writing the sequence number of the current operation,  $seq_i$ , in  $isInRecovery[i]$ . Then, the process repeatedly attempts to acquire the  $mutex$  lock in Line 38 until it succeeds (if it ever does).

Once process  $i$  acquires  $mutex$  it proceeds to recover its operation  $op_i$ . First, it checks if  $op_i$  is already persisted (Line 39). This may occur in two scenarios: either  $op_i$  is a *pure* persisted operation, that is,  $i$  crashed only after updating its  $VTS$  in  $OpVTS$ ; or,  $op_i$  was an effective non-persisted operation but  $i$  has already recovered and persisted its  $VTS$  in



$OpVTS$ , but it crashed before returning, possibly while holding the lock. In both cases, process  $i$  releases the lock and returns a response based on its  $VTS$ . Otherwise, the  $VTS$  of  $op_i$  is not persisted in  $OpVTS$ . The function reads  $W$  using  $ReadVTS$  and stores the returned  $VTS$  in a local variable  $recoveryW[N]$  consisting of  $N$  sequence numbers (Line 42).  $recoveryW$  represents a value of  $W$  in a specific point in time. Process  $i$  treats  $recoveryW$  similarly to how  $W$  is treated by the recovery code for the global-crash model and will order  $op_i$  according to it. In Line 43, process  $i$  waits until each process  $k$  persists the operation with the sequence stored in  $recoveryW[k]$  or is in its recovery code on that operation. We note that processes that did not invoke an FAA operation before the await condition, with  $Seq[p] == 0$ , are skipped.

After the loop of Line 43 terminates, all operations in  $recoveryW$  are either persisted or effective and non-persisted but in recovery. In Line 44, the function stores in the set  $Ops$ , for each process, all its operations whose sequence number is smaller or equal to that stored for the process in  $recoveryW$ . We then proceed in a way similar to the global-crash algorithm. Process  $i$  collects into  $persistedOps$  all persisted operations in  $Ops$ , i.e., whose  $VTS$ s appear in  $OpVTS$  (Line 45), and creates the order  $L_0$  on  $persistedOps$  based on their  $VTS$ s (Line 46). Then, process  $i$  linearizes  $op_i$  using  $InsertOperationIntoOrder$  while the  $prevOrder$  parameter is also  $L_0$  because unlike the global-crash model, here, we insert a single operation,  $op_i$ . Note that each time a process acquires the lock and recovers its effective non-persisted operation,  $op$ , it persists it in  $OpVTS$ , such that the next process will consider  $op$  as a persisted operation and will order it as part of  $L_0$ . Therefore, effective non-persisted operations that have the same *barrier* are ordered by the real-time order of processes capturing the lock. Finally,  $i$  orders  $op_i$ , releases  $mutex$  (Line 48), and applies  $ComputeVal$  to the corresponding  $VTS$  of  $op_i$  to compute its response.

The correctness proof of this algorithm appears in Appendix A.1.

## 5 Impossibility of Wait-Free Recovery in the Individual-Crash Model

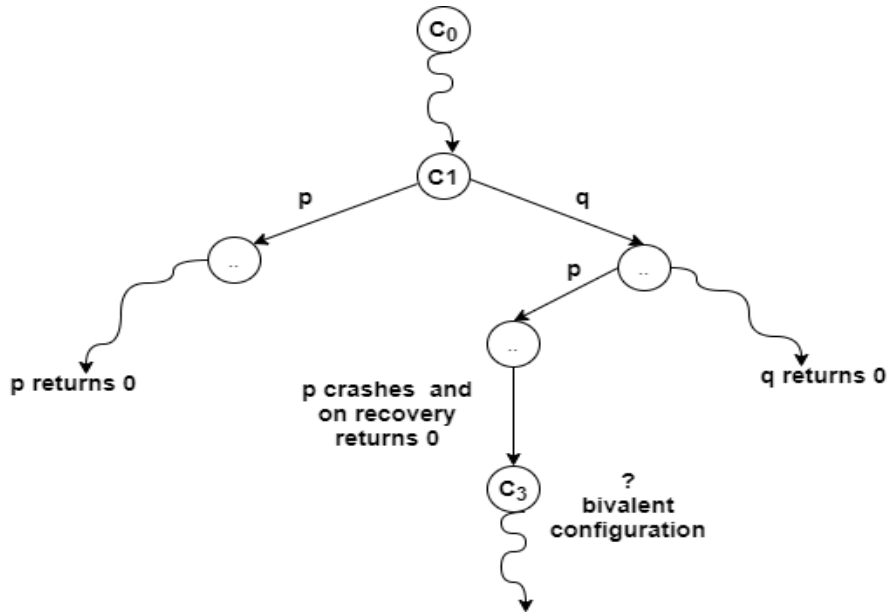
► **Theorem 10.** *There is no detectable self-implementation of FAA in the individual-crash model, such that both the FAA operation and the FAA.RECOVER function are lock-free.*

**Proof.** We prove the theorem using valency arguments [2, 12]. Assume, by way of contradiction, that there is such an implementation with lock-free  $FAA$  and  $FAA.RECOVER$ . In all executions we consider, the initial value of the  $FAA$  object is 0 and both processes,  $p$  and  $q$ , invoke  $FAA$  with argument 1. Hence, one process must return 0 and the other must return 1.

Given a configuration  $C$  and a process  $r \in \{p, q\}$ , we say that  $C$  is  $r$ -valent if there is a crash-free execution starting from  $C$  in which the return value of  $FAA$  or  $FAA.RECOVER$  by  $r$  is 0.  $C$  is *bivalent* if it is both  $p$ -valent and  $q$ -valent.  $C$  is  $p$ -univalent if it is  $p$ -valent and not  $q$ -valent, and symmetrically for  $q$ -univalent. We say that  $C$  is *univalent* if it is either  $p$ -univalent or  $q$ -univalent.

The initial configuration,  $C_0$ , is bivalent because a solo execution of each process returns 0. Following a standard valency argument and since we assume that the  $FAA$  operation is lock-free, there is an execution starting from  $C_0$  that leads to a bivalent configuration  $C_1$ , in which both  $p$  and  $q$  are about to take a *critical* step, i.e, a step that leads to a univalent configuration, one of which is  $p$ -univalent while the other is  $q$ -univalent. A standard argument can be used to show the following claim (see Appendix A.2):

▷ **Claim 11.** The critical steps of  $p$  and  $q$  apply *fetch&add* to the same base object.



■ **Figure 4** Illustration of one case in the proof of Theorem 10.

Assume, without loss of generality, that configuration  $C_1 \circ p$  is  $p$ -univalent while  $C_1 \circ q$  is  $q$ -univalent. Consider the execution from  $C_1$  in which  $p$  takes a step followed by a step of  $q$  and then  $p$  crashes:  $C_2 = C_1 \circ p \circ q \circ CRASH_p$ .

Consider also the execution from  $C_1$  in which  $q$  takes a step followed by a step of  $p$  and then  $p$  crashes:  $C_3 = C_1 \circ q \circ p \circ CRASH_p$ .

A solo execution of  $FAA.RECOVER$  by  $p$  from both configurations  $C_2$  and  $C_3$  must complete, since it is lock-free. Furthermore, these two configurations are indistinguishable to  $p$ , because  $p$ 's response from the primitive *fetch&add* is lost, while the value of the *fetch&add* base object is the same in both configurations. Therefore, an execution of  $FAA.RECOVER$  by  $p$  from both  $C_2$  and  $C_3$  returns the same value  $-v$ .

Assume  $v$  is 0, and thus  $C_3$  is  $p$ -valent. The configuration  $C_1 \circ q \circ p$  is  $q$ -univalent, while  $C_3 = C_1 \circ q \circ p \circ CRASH_p$  is  $p$ -valent. However, these configurations are indistinguishable to  $q$  because it is unaware of  $p$ 's crash, therefore a solo execution of  $q$  from  $C_3$  must return 0, that is,  $C_3$  is  $q$ -valent. This proves that  $C_3$  is bivalent (see Figure 4). The case  $v = 1$  is symmetric, since if  $p$  returns 1 this proves the configuration is  $q$ -valent, as a solo execution of  $q$  after  $p$  completes must return 0; a similar argument proves that  $C_2$  is bivalent.

In this manner, we can keep extending the execution to obtain a crash-free execution of  $q$  in which it performs an infinite number of steps without completing a single  $FAA$  operation, contradicting the assumption that the algorithm is lock-free. ◀

## 6 Discussion

We present two self-implementations of a recoverable detectable  $FAA$  operation, one for the global-crash model and the other for the individual-crash model. Both algorithms are wait-free in crash-free executions. Recovery in both algorithms is blocking. In the global-crash model, this is the result of a design choice to delegate recovery to a single process. For the individual-crash model, we prove that a lock-free self-implementation of a detectable  $FAA$  does not exist.

The proof of Theorem 10 constructs an execution in which one process, repeatedly crashing during its recovery, blocks another process that does not crash, from making progress. This leaves open the question of making progress once no process crashes. We note that Algorithm 2 may have an execution, where process  $i$  acquires the lock during its recovery and then crashes; this means that another process  $j$  that crashes and tries to recover, cannot complete its recovery, even when no further crashes occur. Finding an algorithm that makes progress when processes stop crashing, or proving such an algorithm does not exist, is an interesting question.

Our algorithms apply *fetch&add* primitives to a shared unbounded base object storing a vector timestamp with  $N$  entries. It would be interesting to see if the amount of memory storage can be bounded. This might be challenging, since the vector timestamp is used to precisely track which operations affected each persisted *FAA*, and detect where they should be linearized and with which return value. Another interesting open question is whether there exists a self-implementation of a recoverable *FAA* object for the global-crash model such that both the *FAA* operation and the recovery code are wait-free.

---

## References

---

- 1 M. K. Aguilera and S. Frølund. Strict linearizability and the power of aborting. HPL-2003-241.
- 2 Hagit Attiya, Ohad Ben-Baruch, and Danny Hendler. Nesting-safe recoverable linearizability: Modular constructions for non-volatile memory. In *PODC*, pages 7–16, 2018.
- 3 Ohad Ben-Baruch, Danny Hendler, and Matan Rusanovsky. Upper and lower bounds on the space complexity of detectable objects. In *PODC*, 2020.
- 4 Naama Ben-David, Guy E. Blelloch, Michal Friedman, and Yuanhao Wei. Delay-free concurrency on faulty persistent memory. In *SPAA*, pages 253–264, 2019.
- 5 Ryan Berryhill, Wojciech Golab, and Mahesh Tripunitara. Robust shared objects for non-volatile main memory. In *OPODIS*, pages 20:1–20:17, 2016.
- 6 Dhruva R. Chakrabarti, Hans-Juergen Boehm, and Kumud Bhandari. Atlas: leveraging locks for non-volatile memory consistency. In *OOPSLA*, pages 433–452, 2014.
- 7 D. Y. C. Chan and P. Woelfel. Recoverable mutual exclusion with constant amortized RMR complexity from standard primitives. In *PODC*, 2020.
- 8 Nachshon Cohen, Rachid Guerraoui, and Igor Zablotchi. The inherent cost of remembering consistently. In *SPAA*, pages 259–269, 2018.
- 9 Andreia Correia, Pascal Felber, and Pedro Ramalhete. Romulus: Efficient algorithms for persistent transactional memory. In *SPAA*, pages 271–282, 2018.
- 10 Andreia Correia, Pascal Felber, and Pedro Ramalhete. Persistent memory and the rise of universal constructions. In *EuroSys*, pages 5:1–5:15, 2020.
- 11 Tudor David, Aleksandar Dragojevic, Rachid Guerraoui, and Igor Zablotchi. Log-free concurrent data structures. In *USENIX*, pages 373–386, 2018.
- 12 Michael J. Fischer, Nancy A. Lynch, and Michael Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.
- 13 Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E. Blelloch, and Erez Petrank. NVTraverse: In NVRAM data structures, the destination is more important than the journey. In *PLDI*, pages 377–392, 2020.
- 14 Michal Friedman, Maurice Herlihy, Virendra J. Marathe, and Erez Petrank. Brief announcement: A persistent lock-free queue for non-volatile memory. In *DISC*, pages 50:1–50:4, 2017.
- 15 Michal Friedman, Maurice Herlihy, Virendra J. Marathe, and Erez Petrank. A persistent lock-free queue for non-volatile memory. In *PPoPP*, pages 28–40, 2018.
- 16 Wojciech Golab. The recoverable consensus hierarchy. In *SPAA*, pages 281–291, 2020.
- 17 Wojciech Golab and Danny Hendler. Recoverable mutual exclusion in sub-logarithmic time. In *PODC*, pages 211–220, 2017.

- 18 Wojciech Golab and Aditya Ramaraju. Recoverable mutual exclusion. In *PODC*, pages 65–74, 2016.
- 19 Wojciech M. Golab and Danny Hendler. Recoverable mutual exclusion under system-wide failures. In *PODC*, pages 17–26, 2018.
- 20 R. Guerraoui and R. R. Levy. Robust emulations of shared memory in a crash recovery model. In *ICDCS*, pages 400–407, 2004.
- 21 Maurice Herlihy. Wait free synchronization. *ACM Transactions on Programming Languages and Systems*, 11(1):124–149, 1991.
- 22 Maurice P. Herlihy and Jennette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.
- 23 J. Izraelevitz, H. Mendes, and M. L. Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In *DISC*, pages 313–327, 2016.
- 24 Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-atomic persistent memory updates via JUSTDO logging. In *ASPLOS*, pages 427–442, 2016.
- 25 Prasad Jayanti, Siddhartha V. Jayanti, and Anup Joshi. Recoverable mutex algorithm with sub-logarithmic RMR on both CC and DSM. In *PODC*, pages 177–186, 2019.
- 26 Faisal Nawab, Joseph Izraelevitz, Terence Kelly, Charles B. Morrey III, Dhruva R. Chakrabarti, and Michael L. Scott. Dalí: A periodically persistent hash map. In *DISC*, pages 37:1–37:16, 2017.
- 27 Pedro Ramalhete, Andreia Correia, Pascal Felber, and Nachshon Cohen. Onefile: A wait-free persistent transactional memory. In *DSN*, pages 151–163, 2019.
- 28 David Schwalb, Markus Dreseler, Matthias Uflacker, and Hasso Plattner. Nvc-hashmap: A persistent and concurrent hashmap for non-volatile memories. In *VLDB Workshop on In-Memory Data Management and Analytics*, pages 4:1–4:8, 2015.
- 29 Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen, and Erez Petrank. Efficient lock-free durable sets. In *OOPSLA*, pages 128:1–128:26, 2019.

## A Additional Proofs

### A.1 Sketch of Correctness Proof for Individual-Crash Model

Correctness for crash-free executions is the same as in Section 3.3. We now consider an execution  $\alpha$  with individual crashes.

► **Observation 12.** *Let  $\alpha$  be an execution where  $i$  crashes during operation  $op_i$ . If  $op_i$  is effective non-persisted, its VTS is recovered and stored in  $OpVTS$  exactly once.*

► **Lemma 13.** *The return values of all operations executing Algorithm 2 satisfy the sequential specification of Fetch&Add.*

**Proof.** Note that in the individual-crash model, we do not refer to a final linearization order  $L$ , as in the global-crash model because each process only recovers its own operation by inserting it to an  $L_0$  order of the persisted operations it currently read. Consider the  $L_0$  order by process  $i$ . We say that a return value of each persisted operation,  $op$ , satisfies the sequential specification of *Fetch&Add* although not all effective non-persisted operations that  $op$  follows necessarily appear in  $L_0$ .

The proof relies on the following argument. Assume process  $i$  orders its effective non-persisted operation  $op_i$  based on the  $L_0$  it computes, and let  $op_j$  be the immediate predecessor of  $op_i$  in the new order. Then,  $i$  sets  $VTS_{op_i}$  to be identical to  $VTS_{op_j}$  except for the  $j$ -th component where it is larger by 1. Therefore, the return value it computes is  $res_{op_i} = res_{op_j} + val_{op_j}$ . For any other non-persisted operation  $op_k$  one of the following holds. Either

$op_k$  has been observed by  $VTS_{op_j}$ , that is,  $VTS_{op_j}$  in its  $k$ -th component contains a sequence number larger or equal to the sequence number of  $op_k$ . In such case, the response of  $op_j$  took into consideration the value added by  $op_k$ , and thus also the response of  $op_i$ . Moreover, we are guaranteed that if process  $k$  acquires the lock it will order  $op_k$  before  $op_i$ , since it orders it before the first  $VTS$  that observed it. Otherwise,  $op_k$  was not observed by  $VTS_{op_j}$ , thus not observed also by  $VTS_{op_i}$ , and both return values do not consider the value added by  $op_k$ . However, once  $k$  acquires the lock and orders  $op_k$  it will order it after  $op_i$ , since none of the preceding operations observed  $op_k$ .

This proves that  $res_{op_i}$  satisfies the sequential specification of *Fetch&Add*, since its return value considers all operations that precede it, and those operations will be ordered before  $op_i$  (if they are not persisted yet), and no other operation will be ordered before  $op_i$ . ◀

## A.2 Proof of Claim 11

We consider all possible steps: read, write and *fetch&add*. Assume  $s_p$  and  $s_q$  are critical steps by process  $p$  and  $q$ , respectively, such that  $C_1 \circ s_p$  is  $p$ -univalent while  $C_1 \circ s_q$  is  $q$ -univalent.

- Steps  $s_p$  and  $s_q$  access distinct registers. In this case, these configurations are indistinguishable to  $p$  and  $q$ , that is,  $C \circ s_p \circ s_q \stackrel{p,q}{\sim} C \circ s_q \circ s_p$ .
- Steps  $s_p$  and  $s_q$  read the same register. Also in this case,  $C \circ s_p \circ s_q \stackrel{p,q}{\sim} C \circ s_q \circ s_p$ .
- Step  $s_p$  writes to some register  $r$  step and  $s_q$  reads  $r$ . In this case,  $C \circ s_p \stackrel{p}{\sim} C \circ s_q \circ s_p$  holds.
- Step  $s_p$  applies *fetch&add* and step  $s_q$  reads  $r$ . In this case,  $C \circ s_p \stackrel{p}{\sim} C \circ s_q \circ s_p$  holds.
- Steps  $s_p$  and  $s_q$  write to the same register. In this case,  $C \circ s_p \stackrel{p}{\sim} C \circ s_q \circ s_p$  holds.
- Step  $s_p$  applies *fetch&add*, step  $s_q$  writes to the same register. In this case,  $C \circ s_p \stackrel{p}{\sim} C \circ s_q \circ s_p$  holds.
- Step  $s_p$  applies *fetch&add* with  $val = 0$ , step  $s_q$  applies *fetch&add* to the same register. In this case,  $C \circ s_p \stackrel{p}{\sim} C \circ s_q \circ s_p$  holds.

In each of the above cases, the configurations are indistinguishable to at least one process, and therefore, must have the same valencies. Therefore, it must be that  $p$  and  $q$  apply *fetch&add* to the same base object.



# Using Nesting to Push the Limits of Transactional Data Structure Libraries

Gal Assa ✉

Technion – Israel Institute of Technology, Haifa, Israel

Hagar Meir ✉

IBM Research, Haifa, Israel

Guy Golan-Gueta ✉

Independent researcher, Israel

Idit Keidar ✉

Technion – Israel Institute of Technology, Haifa, Israel

Alexander Spiegelman ✉

Novi Research, USA

---

## Abstract

Transactional data structure libraries (TDSL) combine the ease-of-programming of transactions with the high performance and scalability of custom-tailored concurrent data structures. They can be very efficient thanks to their ability to exploit data structure semantics in order to reduce overhead, aborts, and wasted work compared to general-purpose software transactional memory. However, TDSLs were not previously used for complex use-cases involving long transactions and a variety of data structures.

In this paper, we boost the performance and usability of a TDSL, towards allowing it to support complex applications. A key idea is *nesting*. Nested transactions create checkpoints within a longer transaction, so as to limit the scope of abort, without changing the semantics of the original transaction. We build a Java TDSL with built-in support for nested transactions over a number of data structures. We conduct a case study of a complex network intrusion detection system that invests a significant amount of work to process each packet. Our study shows that our library outperforms publicly available STMs twofold without nesting, and by up to 16x when nesting is used.

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

**Keywords and phrases** Transactional Libraries, Nesting

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2021.30

**Related Version** *Full Version*: <https://arxiv.org/abs/2001.00363> [2]

**Supplementary Material** *Software (Library)*: <https://github.com/galassatech/TDSL-Nesting-Java/>  
*Software (Benchmark)*: <https://github.com/galassatech/NIDS-Java/>

**Funding** *Gal Assa*: Funded in part by the Hasso Plattner Institute.

## 1 Introduction

### 1.1 Transactional Libraries

The concept of memory transactions [25] is broadly considered to be a programmer-friendly paradigm for writing concurrent code [22, 39]. A transaction spans multiple operations, which appear to execute atomically and in isolation, meaning that either all operations commit and affect the shared state or the transaction aborts. Either way, no partial effects of on-going transactions are observed.



© Gal Assa, Hagar Meir, Guy Golan-Gueta, Idit Keidar, and Alexander Spiegelman;  
licensed under Creative Commons License CC-BY 4.0

25th International Conference on Principles of Distributed Systems (OPODIS 2021).

Editors: Quentin Bramas, Vincent Gramoli, and Alessia Milani; Article No. 30; pp. 30:1–30:17

Leibniz International Proceedings in Informatics



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

Despite their appealing ease-of-programming, software transactional memory (STM) toolkits [6, 24, 37] are seldom deployed in real systems due to their huge performance overhead. The source of this overhead is twofold. First, an STM needs to monitor all random memory accesses made in the course of a transaction (e.g., via instrumentation in VM-based languages [28]), and second, STMs abort transactions due to conflicts. Instead, programmers widely use concurrent data structure libraries [40, 30, 21, 5], which are much faster but guarantee atomicity only at the level of a single operation on a single data structure.

To mitigate this tradeoff, Spiegelman et al. [41] have proposed *transactional data structure libraries (TDSL)*. In a nutshell, the idea is to trade generality for performance. A TDSL restricts transactional access to a pre-defined set of data structures rather than arbitrary memory locations, which eliminates the need for instrumentation. Thus, a TDSL can exploit the data structures' semantics and structure to get efficient transactions bundling a sequence of data structure operations. It may further manage aborts on a semantic level, e.g., two concurrent transactions can simultaneously change two different locations in the same list without aborting. While the original TDSL library [41] was written in C++, we implement our version in Java. We offer more background on TDSL in Section 2.

Quite a few works [29, 9, 46, 31] have used and extended TDSL and similar approaches like STO [26] and transactional boosting [23]. These efforts have shown good performance for fairly short transactions on a small number of data structures. Yet, despite their improved scalability compared to general purpose STMs, TDSLs have also not been applied to long transactions or complex use-cases. A key challenge arising in long transactions is the high potential for aborts and the large penalty that such aborts induce as much work is wasted.

## 1.2 Our Contribution

**Transactional nesting.** In this paper we push the limits of the TDSL concept in an attempt to make it more broadly applicable. Our main contribution, presented in Section 3, is facilitating long transactions via *nesting* [33]. Nesting allows the programmer to define nested *child* transactions as self-contained parts of larger *parent* transactions. This controls the program flow by creating *checkpoints*; upon abort of a nested child transaction, the checkpoint enables retrying only the child's part and not the preceding code of the parent. This reduces wasted work, which, in turn, improves performance. At the same time, nesting does not relax consistency or isolation, and continues to ensure that the entire parent transaction is executed atomically. We focus on *closed nesting* [42], which, in contrast to so-called flat nesting, limits the scope of aborts, and unlike open nesting [35], is generic and does not require semantic constructs.

The flow of nesting is shown in Algorithm 1. When a child commits, its local state is migrated to the parent but is not yet reflected in shared memory. If the child aborts, then the parent transaction is checked for conflicts. And if the parent incurs no conflicts in its part of the code, then only the child transaction retries. Otherwise, the entire transaction does. It is important to note that the semantics provided by the parent transaction are not altered by nesting. Rather, nesting allows programmers to identify parts of the code that are more likely to cause aborts and encapsulate them in child transactions in order to reduce the abort rate of the parent.

Yet nesting induces an overhead which is not always offset by its benefits. We investigate this tradeoff using microbenchmarks. We find that nesting is helpful for highly contended operations that are likely to succeed if retried. We also find that nested variants of TDSL improve performance of state-of-the-art STMs with transaction friendly data structures.



---

**Algorithm 1** Transaction flow with nesting.
 

---

```

1: TXbegin()
2:   [Parent code]                                ▷ On abort – retry parent
3:   nTXbegin()                                  ▷ Begin child transaction
4:     [Child code]                              ▷ On abort – retry child or parent
5:   nTXend()                                    ▷ On commit – migrate changes to parent
6:   [Parent code]                              ▷ On abort – retry parent
7: TXend()                                       ▷ On commit – apply changes to shared state
  
```

---

**NIDS benchmark.** In Section 4 we introduce a new benchmark of a *network intrusion detection system (NIDS)* [19], which invests a fair amount of work to process each packet. This benchmark features a pipelined architecture with long transactions, a variety of data structures, and multiple points of contention. It follows one of the designs suggested in [19] and executes significant computational operations within transactions, making it more realistic than existing intrusion-detection benchmarks (e.g., [27, 32]).

**Enriching the library.** In order to support complex applications like NIDS, and more generally, to increase the usability of TDSLs, we enrich our transactional library in Section 3 with additional data structures – producer-consumer pool, log, and stack – all of which support nesting. The TDSL framework allows us to custom-tailor to each data structure its own concurrency control mechanism. We mix optimism and pessimism (e.g., stack operations are optimistic as long as a child has popped no more than it pushed, and then they become pessimistic), and also fine tune the granularity of locks (e.g., one lock for the whole stack versus one per slot in the producer-consumer pool).

**Evaluation.** In Section 5, we evaluate our NIDS application. We find that nesting can improve performance by up to 8x. Moreover, nesting improves scalability, reaching peak performance with as many as 40 threads as opposed to 28 without nesting.

**Summary of contributions.** This paper is the first to bring nesting into transactional data structure libraries and also the first to implement closed nesting in sequential STMs. We implement a Java version of TDSL with built-in support for nesting. Via microbenchmarks, we explore when nesting is beneficial and show that in some scenarios, it can greatly reduce abort rates and improve performance. We build a complex network intrusion detection application, while enriching our library with the data structures required to support it. We show that nesting yields significant improvements in performance and abort rates.

## 2 A Walk down Transactional Data Structure Lane

Our algorithm builds on ideas used in TL2 [6], which is a generic STM framework, and in TDSL [41], which suggests forgoing generality for increased efficiency. We briefly overview their modus operandi as background for our work.

The TL2 [6] algorithm introduced a version-based approach to STM. The algorithm's building blocks are version clocks, read-sets, write-sets, and a per-object lock. A *global version clock (GVC)* is shared among all threads. A transaction has its own *version clock (VC)*, which is the value of GVC when the transaction begins. A shared object has a version, which is the VC of the last transaction that modified it. The read- and write-sets consist of references to objects that were read and written, respectively, in a transaction's execution.

## 30:4 Using Nesting to Push the Limits of Transactional Data Structure Libraries

Version clocks are used for *validation*: Upon read, the algorithm first checks if the object is locked and then the VC of the read object is compared to the transaction's VC. If the object is locked or its VC is larger than the transaction's, then we say the validation *fails*, and the transaction aborts. Intuitively, this indicates that there is a *conflict* between the current transaction, which is reading the object, and a concurrent transaction that writes to it.

At the end of a transaction, all the objects in its write-set are locked and then every object in the read-set is revalidated. If this succeeds, the transaction commits and its write-set is reflected to shared memory. If any lock cannot be obtained or any of the objects in the read-set does not pass validation, then the transaction aborts and retries.

*Opacity* [18] is a safety property that requires every transaction (including aborted ones) to observe only *consistent* states of the system that could have been observed in a sequential execution. TL2's read-time validation (described above) ensures opacity.

In TDSL, the TL2 approach was tailored to specific data structures (skiplists and queues) so as to benefit from their internal organization and semantics. TDSL's skiplists use small read- and write-sets capturing only accesses that induce conflicts at the data structure's semantic level. For example, whereas TL2's read-set holds all nodes traversed during the lookup of a particular key, TDSL's read-set keeps only the node holding this key. In addition, whereas TL2 uses only optimistic concurrency-control (with commit-time locking), TDSL's queue uses a semi-pessimistic approach. Since the head of a queue is a point of contention, *deq* immediately locks the shared queue (although the actual removal of the object from the queue is deferred to commit time); the *enq* operation remains optimistic.

Note that TDSL is less general than generic STMs: STM transactions span all memory accesses within a transaction, which is enabled, e.g., by instrumentation of binary code [1] and results in large read- and write-sets. TDSL provides transactional semantics within the confines of the library's data structures while other memory locations are not accessed transactionally. This eliminates the need for instrumenting code.

### 3 Adding Nesting to TDSL

We introduce nesting into TDSL. Section 3.1 describes the correct behavior of nesting and offers a general scheme for making a transactional *data structure (DS)* nestable. Section 3.2 then demonstrates this technique in the two DSs supported by the original TDSL – queue and skiplist. We restrict our attention to a single level of nesting for clarity, as we could not find any example where deeper nesting is useful. However, deeper nesting could be supported along the same lines if required, via migrating the descendant's local state to its ancestor as described in *nCommit* below. In Section 3.3 we use microbenchmarks to investigate when nesting is useful and when less so, and to compare our library's performance with transactional data structures used on top of general purpose STMs. The nestable log, stack, and producer-consumer pool are described in Section 3.4

#### 3.1 Nesting Semantics and General Scheme

Nesting is a technique for defining child sub-transactions within a transaction. A child has its own local state (read- and write-sets), and it may also observe its parent's local state. A child transaction's commit migrates its local state to its parent but not to shared memory visible by other threads. Thus, the child's operations take effect when the parent commits, and until then remain unobservable.

**Correctness.** A nested transaction implementation ought to ensure that (1) nested operations are not visible in the shared state until the parent commits; and (2) upon a child's commit, its operations are correctly reflected in the parent's state exactly as if all these operations were executed as part of the parent. In other words, nesting part of a transaction does not change its externally visible behavior.

**Implementation scheme.** In our approach, the child uses its parent's VC. This way, the child and the parent observe the shared state at the same "logical time" and so read validations ensure that the combined state observed by both of them is consistent, as required for opacity.

Algorithm 2 introduces general primitives for nesting arbitrary DSs. The *nTXbegin* and *nCommit* primitives are exposed by the library and may be called by the user as in Algorithm 1. When user code operates on a transactional DS managed by the library for the first time, it is registered in the transaction's *childObjectList*, and its local state and *lockSet* are initialized empty. *nTryLock* may be called from within the library, e.g., a nested dequeue calls *nTryLock*. Finally, *nAbort* may be called by both the user and the library.

We offer the *nTryLock* function to facilitate pessimistic concurrency control (as in TDSL's queues), where a lock is acquired before the object is accessed. This function (1) locks the object if it is not yet locked; and (2) distinguishes newly acquired locks from ones that were acquired by the parent. The latter allows the child to release its locks without releasing ones acquired by its parent.

A nested commit, *nCommit*, validates the child's read-set in all the transaction's DSs *without* locking the write-set. If validation is successful, the child migrates its local state to the parent, again, in all DSs, and also makes its parent the owner of all the locks it holds. To this end, every nestable DS must support *migrate* and *validate* functions, in addition to nested versions of all its methods.

■ **Algorithm 2** Nested begin, lock, commit, and abort.

---

<pre> 1: <b>procedure</b> NTXBEGIN 2:   alloc childObjectList, init empty 3: <b>On first access to</b>    <b>obj in child transaction</b> 4:   add obj to childObjectList 5: <b>procedure</b> NABORT 6:   <b>for each</b> obj in childObjectList <b>do</b> 7:     release locks in lockSet 8:   parent VC ← GVC 9:   <b>for each</b> obj in childObjectList <b>do</b> 10:    validate parent 11:    <b>if</b> validation fails 12:      <b>abort</b> 13:   <b>Restart</b> child </pre>	<pre> 14: <b>procedure</b> NCOMMIT 15:   <b>for each</b> obj in childObjectList <b>do</b> 16:     validate obj with parent's VC 17:     <b>if</b> validation fails 18:       <b>nAbort</b> 19:   <b>for each</b> obj in childObjectList <b>do</b> 20:     obj.migrate           ▷ DS specific code 21:   <b>for each</b> lock in lockSet <b>do</b> 22:     transfer lock to parent 23: <b>procedure</b> NTRYLOCK(obj) 24:   <b>if</b> obj is unlocked 25:     lock obj with child id 26:     add obj to lockSet 27:   <b>if</b> obj is locked but not by parent 28:     <b>nAbort</b> </pre>
--	--

---

In case the child aborts, it releases all of its locks. Then, we need to decide whether to retry the child or abort the parent too. Simply retrying the child without changing the VC is liable to fail because it would re-check the same condition during validation, namely, comparing read object VCs to the transaction's VC. We therefore update the VC to the current GVC value (line 8) before retrying. This ensures that the child will not re-encounter past conflicts. But in order to preserve opacity, we must verify that the state the parent observed is still consistent at the *new* logical time (in which the child will be retried) because operations within a child transaction ought to be seen as if they were executed as part of

the parent. To this end, we revalidate the parent’s read-set against the new VC (line 10). This is done without locking its write-set. Note that if this validation fails then the parent is deemed to abort in any case, and the early abort improves performance. If the revalidation is successful, we restart only the child (line 13).

Recall that retrying the child is only done for performance reasons and it is always safe to abort the parent. Specific implementations may thus choose to limit the number of times a child is retried.

### 3.2 Queue and Skiplist

We extend TDSL’s queue with nested transactional operations in Algorithm 3. The original queue’s local state includes a list of nodes to enqueue and a reference to the last node to have been dequeued (together, they replace the read- and write-sets). We refer to these components as the parent’s *local queue*, or *parent queue* for short. Nested transactions hold an additional *child queue* in the same format.

■ **Algorithm 3** Nested operations on queues.

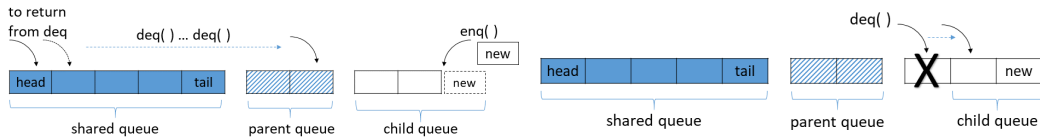
---

```

1: Queue
2:   sharedQ      ▷ Shared among all threads
3:   parentQ, childQ    ▷ Thread local
4: procedure NENQ(val)
5:   childQ.APPEND(val)
6: procedure MIGRATE
7:   PARENTQ.APPENDALL(childQ)
8: procedure VALIDATE
9:   return true
10: procedure NDEQ()
11:   NTRYLOCK()
12:   val ← next node in sharedQ
13:   if val = ⊥
14:     val ← next node in parentQ
15:   if val = ⊥
16:     val ← childQ.DEQ()
17:   return val

```

---



■ **Figure 1** Nested queue operations: dequeue returns objects from the shared, and then parent states without dequeuing them, and when they are exhausted, dequeues from the child’s queue; enqueue always enqueues to the child’s queue.

The nested enqueue operation remains simple: it appends the new node to the tail of the child queue (line 5). The nested dequeue first locks the shared queue. Then, the next node to return from dequeue is determined in lines 12 – 16, as illustrated in Figure 1. As long as there are nodes in the shared queue that have not been dequeued, dequeue returns the value of the next such node but does not yet remove it from the queue (line 12). Whenever the shared queue has been exploited, we proceed to traverse the parent transaction’s local queue (line 14), and upon exploiting it, perform the actual dequeue from the nested transaction’s local queue (line 16). A commit appends (migrates) the entire local queue of the child to the tail of the parent’s local queue. The queue’s validation always returns true: if it never invoked dequeue, its read set is empty, and otherwise, it had locked the queue.

We note that acquiring locks within nested transactions may result in deadlock. Consider the following scenario: Transaction  $T_1$  dequeues from  $Q_1$  and  $T_2$  dequeues from  $Q_2$ , and then both of them initiate nested transactions that dequeue from the other queue ( $T_2$  from  $Q_1$  and vice versa). In this scenario, both child transactions will inevitably fail no matter

how many times they are tried. To avoid this, we retry the child transaction only a bounded number of times, and if it exceeds this limit, the parent aborts as well and releases the locks acquired by it. Livelock at the parent level can be addressed using standard mechanisms (backoff, etc.).

To extend TDSL's skiplist with nesting we preserve its optimistic design. A child transaction maintains read- and write-sets of its own, and upon commit, merges them into its parent's sets. As in the queue, read operations of child transactions can read values written by the parent. Validation of the child's read-set verifies that the versions of the read objects have not changed. The skiplist's implementation is straightforward, and its pseudo-code is presented in the full version.

### 3.3 To Nest, or Not to Nest

Nesting limits the scope of abort and thus reduces the overall abort rate. On the other hand, nesting introduces additional overhead. We now investigate this tradeoff using a synthetic microbenchmark and further provide guidelines for nesting in transactional data structure libraries.

**Experiment setup.** We run our experiments and measure throughput on an AWS m5.24xlarge instance with 2 sockets with 24 cores each, for a total of 48 physical cores. We disable hyperthreading.

We use a synthetic workload, where every thread runs 50,000 transactions, each consisting of 10 random operations on a shared skiplist followed by 2 random operations on a shared queue. Operations are chosen uniformly at random, and so are the keys for the skiplist operations. We examine three different nesting policies: (1) flat transactions (no nesting); (2) nesting skiplist operations and queue operations; and (3) nesting only queue operations.

We examine two scenarios in terms of contention on the skiplist. In the low contention scenario, the skiplist's key range is from 0 to 50,000. In the second scenario, it is from 0 to 50, so there is high contention. Every experiment is repeated 10 times.

**Compared systems.** We use the Synchrobench [15] framework in order to compare our TDSL to existing data structures optimized for running within transactions. Specifically, we run  $\varepsilon$ -STM (Elastic STM [13]) with the three transactional skiplists available as part of Synchrobench – transactional friendly skiplist set, transactional friendly optimized skiplist set, and transactional Pugh skiplist set – and to the (single) available transactional queue therein. In all experiments we ran, the friendly optimized skiplist performed better than the other two, and so we present only the results of this data structure. This skiplist requires a dedicated maintenance thread in addition to the worker threads. To provide an upper bound on the performance of  $\varepsilon$ -STM, we allow it to use the same number of worker threads as TDSL plus an additional maintenance thread, e.g., we compare TDSL with eight threads to  $\varepsilon$ -STM with a total of nine. We note that  $\varepsilon$ -STM requires one maintenance thread per skip list; again, to favor  $\varepsilon$ -STM, we use a single skiplist in the benchmarks.

Synchrobench supports elastic transactions in addition to regular (opaque) ones, and also optionally supports multi-version concurrency control (MVCC) [16, 17], which reduces abort rates on read-only transactions. We experiment with these two modes as well.

We also ran our experiments on TL2 with the transactional friendly skiplist, but it was markedly slower than the alternatives, and in many experiments failed to commit transactions within the internally defined maximum number of attempts. We therefore omit these results.

**Results.** Figure 2 shows the average throughput obtained.

In the low contention scenario (Figure 2a), nesting both queue and skiplist operations yields the best performance in the vast majority of data points. It improves throughput by 1.6x on average compared to flat transactions on 48 threads. It is worth noting that nesting provides the highest throughput without relaxing opacity like elastic transactions, and without keeping track of multiple versions of memory objects like MVCC. This is due to less work being wasted upon abort. Nesting queue operations seems to be the main reason for the performance gain compared to flat transactions, as nesting only queue operations yields comparable performance. In fact, nesting only the operations on the contended object may be preferable, as it provides the best of both worlds: low abort rates, as discussed later in this section, and less overhead around skiplist sub-transactions. The overhead difference is seen clearly when examining the performance of the two nesting variants of TDSL with a single thread, when nesting induces overhead and offers no benefits.

We further investigate the effect of nesting via the abort rate, shown in Figure 2c (for the low contention scenario). We see the dramatic impact of nesting on the abort rate. This sheds light on the throughput results. Nesting both skiplist and queue operations indeed minimizes the abort rate. However, the gap in abort rate does not directly translate to throughput, as it is offset by the increased overhead.

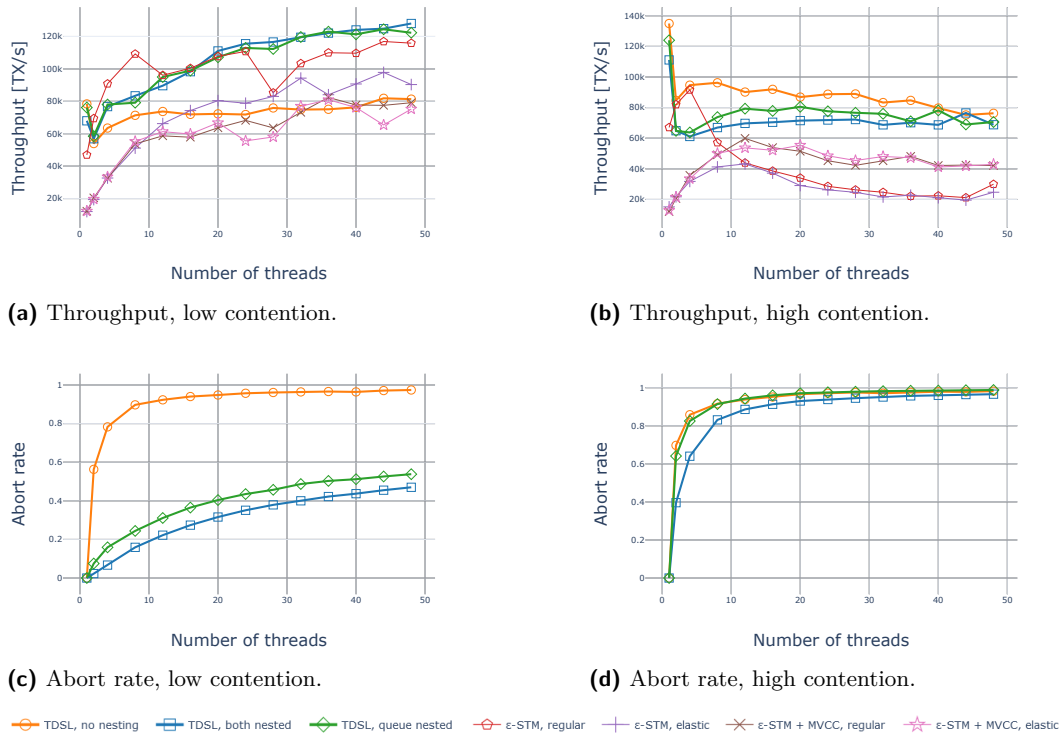
In the high contention scenario (Figure 2b), both DSs are highly contended, and nesting is harmful. The high contention prevents the throughput from scaling with the number of threads, and we observe degradation in performance starting from as little as 2 concurrent threads for TDSL, and between 4-12 concurrent threads for the other variants. From the abort rate point of view (Figure 2d), the majority of transactions abort with as little as 4 threads regardless of nesting, and 80-90% abort with 8 threads. Despite exhibiting the lowest abort rate, nesting all operations performs worse than other TDSL variants. In this scenario, too, nested TDSL performs better than the  $\varepsilon$ -STM variants despite being unfruitful compared to flat transactions.

Aborts on queue operations occur due to failures of *nTryLock*, which has a good chance of succeeding if retried. On the other hand, aborts on nested skiplist operations are due to reading a higher version than the parent’s VC. In such scenarios, the parent is likely to abort as well since multiple threads modify a narrow range of skiplist elements, hence an aborted child is not very likely to commit even if given another chance. Overall, we find that nesting the highly contended queue operations is more useful than nesting map operations – even when contended. Thus, contention alone is not a sufficient predictor for the utility of nesting. Rather, the key is the likelihood of the failed operation to succeed if retried.

### 3.4 Additional Data Structures

Transactions may span multiple objects of different types. Every data structure implements the methods defined by its type (e.g., dequeue for queue), as well as methods for validation, migrating a child transaction’s state to its parent, and committing changes to shared memory. We extend our Java TDSL with three widely used data structures – a producer-consumer pool (supports *produce* and *consume*), a log (*read*, *append*), and a stack (*push*, *pop*). We briefly describe their modus operandi next. We refer the reader to the full version for the specifics of the nesting support transactional implementation and its correctness.

Both the log and the stack resemble with the queue, as both have single points of contention: the log’s tail and the stacks head. Accessing elements preceding the log’s tail may always succeed and does not generate contention, but appending to its tail requires acquiring the log’s lock upon first append, in a similar manner to the queue’s dequeue, as competing



**Figure 2** The impact of nesting in TDSL, compared to transactional friendly data structures on  $\epsilon$ -STM.

transactions will surely abort if the appending transaction commits. For the stack, reading (i.e., popping) is similar in nature to the dequeue operation, and is performed pessimistically. However, a stack may employ its semantics to create some degree of optimism: Since push and pop operations cancel out with each other, a transaction is not required to operate pessimistically and acquire a lock until it had popped more elements than it had pushed. If at any time during the transaction there was at least as many pushed items as popped ones, the stack's lock will only be acquired at the end of the transaction to append the local stack to its top. In both data structures, child transactions maintain local logs and stacks with elements to be added to the parent's local structures, and eventually to the shared structures upon commit.

The producer-consumer pool is unlike any other data structure implemented in TDSL so far: its sequential specification does not require contained elements be ordered. It has multiple potential points of contention, and has no read-only operations. Transactions (both parent and nested) mark slots in the shared pool as that are accessed within the transaction, so other threads do not access them. The transactional implementation includes a cancellation mechanism that releases nodes that were produced and then consumed in the same transaction, as well a migration mechanism to apply operations from a nested transaction to the parent's state.

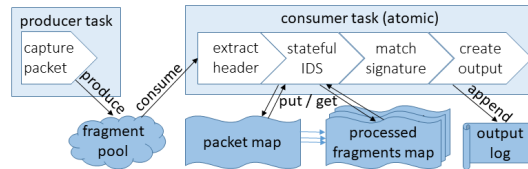
## 4 NIDS Case Study

We conduct a case study of parallelizing a full-fledged network intrusion detection system using memory transactions. In this section we provide essential background for multi-threaded IDS systems, describe our NIDS software and point out candidates for nesting.

## 30:10 Using Nesting to Push the Limits of Transactional Data Structure Libraries

Intrusion detection is a basic security feature in modern networks, implemented by popular systems such as Snort [38], Suricata [14], and Zeek [36]. As network speeds increase and bandwidth grows, NIDS performance becomes paramount, and multi-threading becomes instrumental [19].

**Multi-threaded NIDS.** We develop a multi-threaded NIDS benchmark. The processing steps executed by the benchmark follow the description in [19]. As illustrated in Figure 3, our design employs two types of threads. First, *producers* simulate the *packet capture* process of reading packet fragments off a network interface. In our benchmark, we do not use an actual network, and so the producers generate the packets and push packet fragments into a shared producer-consumer pool called the *fragments pool*. The rationale for using dedicated threads for packet capture is that – in a real system – the amount of work these threads have scales with network resources rather than compute and DRAM resources. In our implementation, the producers simply drive the benchmark and do not do any actual work.



■ **Figure 3** Our NIDS benchmark: tasks and data structures.

Packet processing is done exclusively by the *consumer* threads, each of which consumes and processes a single packet fragment from the shared pool. Algorithm 4 describes the consumer’s code. To ensure consistency, each consumer executes as a single atomic transaction. It begins by performing *header extraction*, namely, extracting information from the link layer header. The next step is called *stateful IDS*; it consists of packet reassembly and detecting violations of protocol rules. Reassembly uses a shared *packet map* associating each packet with its own shared *processed fragment map*. The first thread to process a fragment pertaining to a particular packet creates the packet’s fragment map whereas other threads append fragments to it. Similarly, only the thread that processes a packet’s last fragment continues to process the packet, while the remaining threads move on to process other fragments from the pool. By using atomic transactions, we guarantee that indeed there are unique “first” and “last” threads and so consistency is preserved.

■ **Algorithm 4** Consumer code.

---

```

1:  $f \leftarrow \text{fragmentPool.consume}()$ 
2: process headers of  $f$ 
3:  $\text{fragmentMap} \leftarrow \text{packetMap.get}(f)$ 
    $\triangleright$  Start nested TX
4: if  $\text{fragmentMap} = \perp$ 
5:    $\text{fragmentMap} \leftarrow \text{new map}$ 
6:    $\text{packetMap.put}(f, \text{fragmentMap})$ 
    $\triangleright$  End nested TX
7:  $\text{fragmentMap.put}(f.id, f)$ 
8: if  $f$  is the last fragment in packet
9:   reassemble and inspect packet
    $\triangleright$  Long computation
10: log the result
    $\triangleright$  Nested TX

```

---



The thread that puts together the packet proceeds to the *signature matching* phase, whence the reassembled packet's content is tested against a set of logical predicates; if all are satisfied, the signature matches. This is the most computationally expensive stage [19]. Finally, the thread generates a packet trace and writes it to a shared log.

As an aside, we note that our benchmark performs five of the six processing steps detailed in [19]; the only step we skip is *content normalization*, which unifies the representations of packets that use different application-layer protocols. This phase is redundant in our solution since we use a unified packet representation to begin with. In contrast, the *intruder* benchmark in STAMP [32] implements a more limited functionality, consisting of packet reassembly and naïve signature matching: threads obtain fragments from their local states (rather than a shared pool), signature matching is lightweight, and no packet traces are logged. This results in significantly shorter transactions than in our solution.

**Nesting.** We identify two candidates for nesting. The first is the logging operation given that logs are prone to be highly contended. Because in this application the logs are write-only, transactions abort only when they contend to write at the tail and not because of consistency issues. Therefore, retrying the nested transaction amounts to retrying to acquire a lock on the tail, which is much more efficient than restarting the transaction.

Second, when a packet consists of multiple fragments, its entry in the packet map is contended. In particular, for every fragment, a transaction checks whether an entry for its packet exists in the map, and creates it if it is absent. Nesting lines 3 - 6 of Algorithm 4 may thus prevent aborts.

## 5 NIDS Evaluation

We now experiment with nesting in the NIDS benchmark. We detail our evaluation methodology in Section 5.1 and present quantitative results in Section 5.2.

### 5.1 Experiment Setup

Our baseline is TDSL without nesting, which is the starting point of this research. We also compare to the open source Java STM implementation of TL2 by Korland et al. [28], as well as  $\epsilon$ -STM [13] and PSTM [16, 17]. The results we obtained for  $\epsilon$ -STM and PSTM were very similar to those of TL2 and are omitted from the figures to avoid clutter. Note that the open-source implementations of  $\epsilon$ -STM and PSTM optimize only data structures that contain integers; they use bare-STM implementations for data structures holding general objects, as the data structures in our benchmark do. This explains why their performance is sub-optimal in this benchmark.

We experiment with nesting each of the candidates identified in Section 4 (put-if-absent to the packetMap and updating the log), and also with nesting both. Our baseline executes *flat transactions*, i.e., with no nesting. In TDSL, the packet pool is a producer-consumer pool, the map of processed packets is a skiplist of skiplists, and the output block is a set of logs. For TL2, the packet pool is implemented with a fixed-size queue, the packet map is an RB-tree of RB-trees, and the output log is a set of vectors. We use the implementations provided in [27] without modification.

The experiment environment is the same as for the microbenchmark described in Section 3.3. We repeated the experiment on an in-house 32-core Xeon machine and observed similar trends; these results are omitted. We run each experiment 5 times and plot all data points, connecting the median values with a curve.

## 30:12 Using Nesting to Push the Limits of Transactional Data Structure Libraries

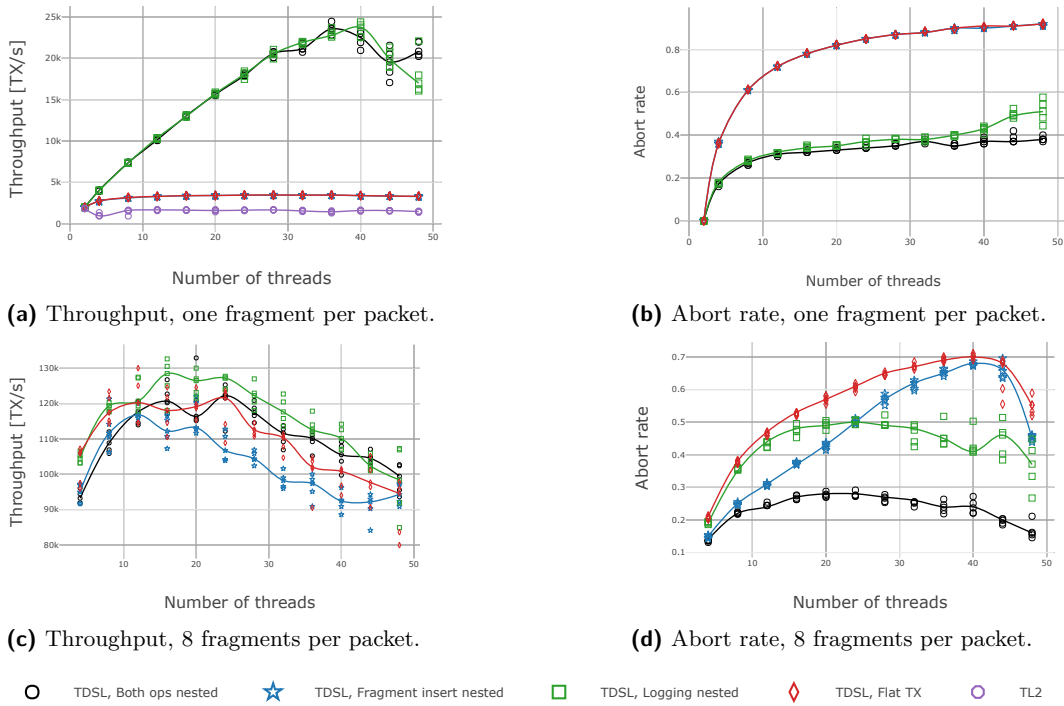


Figure 4 NIDS experiments results.

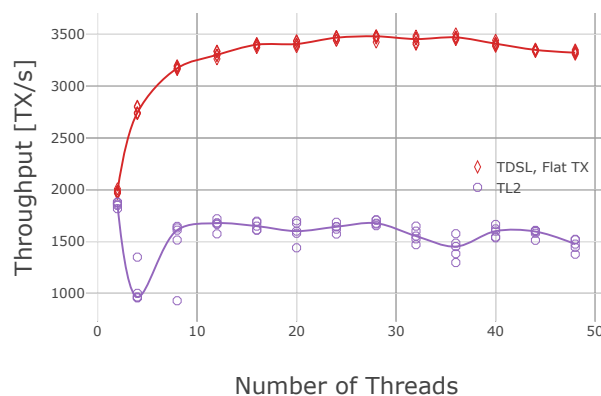
We conduct two experiments. In the first, each packet consists of a single fragment, there is one producer thread, and we scale the number of consumers. In the second experiment, there are 8 fragments per packet and as we scale the number of threads, we designate half the threads as producers. We experimented also with different ratios of producers to consumers, but this did not seem to have a significant effect on performance or abort rates, so we stick to one configuration in each experiment. The number of fragments per packet governs contention: If there are fewer fragments then more threads try to write to logs simultaneously. More fragments, on the other hand, induce more put-if-absent attempts to create maps.

## 5.2 Results

**Performance.** Figures 4a and 4b show the throughput and abort rate in a run with 1 fragment per packet and a single producer. Whereas the performance of all solutions is similar when we run a single consumer, performance differences become apparent as the number of threads increases. For flat transactions (red diamonds), TDSL's throughput is consistently double that of TL2 (purple octagons), as can be observed in Figure 5, which zooms in on these two curves in the same experiment. We note that the TDSL work [41] reported better performance improvements over TL2, but they ran shorter transactions that did not write to a contended log at the end, where TDSL's abort rate remained low. In contrast, our benchmark's long transactions result in high abort rates in the absence of nesting. Nesting the log writes (green squares) improves throughput by an additional factor of up to 6, which is in line with the improvement of TDSL over TL2 reported in [41], and also reduces the abort rate by a factor of 2. The packet map is not contended in this experiment, and so transactions with nested insertion to the map behave similarly to flat ones (in terms of both throughput and abort rate).

■ **Table 1** Scalability: peak performance (tx/sec) / number of threads where it is achieved.

Algorithm	1 Fragment	8 Fragments
TL2	1.6K / 8	24K / 4
TDSL flat	3.5K / 28	122K / 24
TDSL nesting log	23.5K / 40	127K / 24
TDSL nesting put-if-absent	3.5K / 28	113K / 20
TDSL nesting both	23.5K / 36	122K / 24



■ **Figure 5** Throughput of TL2 and flat transactions in TDSL, a single producer and one fragment per packet.

Figure 4c shows the results in experiments with 8 fragments per packet. For clarity, we omit TL2 from this graph because it performs 6 times worse than the lowest alternative. Here, too, the best approach is to nest only log updates, but the impact of such nesting is less significant in this scenario, improving throughput only by about 20%. This is because with one fragment per packet, every transaction tries to write to the log, whereas with 8, only the last fragment's transaction does, reducing contention on the log. Nevertheless, the effect of nesting log updates is more significant as it reduces the number of aborts by a factor of 3, and thus saves work.

Unlike in the 1-thread scenario, with 8 threads, there is contention on the put-if-absent to the fragment map, and so nesting this operation reduces aborts. At first, it might be surprising that flat transactions perform better than ones that nest the put-if-absent despite their higher abort rate. However, the abort reduction has a fairly low impact since this operation is performed early in the transaction. Thus, the overhead induced by nesting exceeds the benefit of not repeating the earlier part of the computation. The effect of this overhead is demonstrated in the difference in performance between nesting both candidates (black circles) and nesting only the log writes (green squares).

**Scaling.** Not only does nesting have a positive effect on performance, it improves scalability as well. For instance, Figure 4a shows that throughput increases linearly all the way up to 40 threads when nesting the logging operation, whereas flat nesting, as can be seen in Figure 5, peaks at 28 threads but saturates already at 16. Table 1 summarizes the scaling factor in both experiments.

## 6 Related Work

**Transactional data structures.** Since the introduction of TDSL [41] and STO [26], transactional libraries got a fair bit of attention [29, 9, 46, 31]. Other works have focused on wait-free [29] and lock-free [9, 46] implementations (as opposed to TDSL and STO’s lock-based approach). Such algorithms are interesting from a theoretical point of view, but provide very little performance benefits, and in some cases can even yield worse results than lock-based solutions [10, 7]. Lebanoff et al. [31] introduce a trade-off between low abort rate and high computational overhead. By restricting their attention to static transactions, they are able to perform scheduling analyses in order to reduce the overall system abort rate. We, in contrast, support dynamic transactions.

Transactional boosting and its follow-ups [20, 23] offer generic approaches for making concurrent data structures transactional. However, they do not exploit the structure of the transformed data structure, and instead rely on semantic abstractions like compensating actions and abstract locks.

Some full-fledged STMs incorporate optimization for specific data structures. For instance,  $\epsilon$ -STM [13] and PSTM [16, 17] support elastic transactions on search data structures. Note, however, that unlike closed nesting, elastic transactions relax transactional semantics. PSTM allows programmers to select the concurrency control mechanism (MVCC or single-version) and the required semantics (elastic or regular) for each transaction. While this offers a potential for performance gains, our results in Section 3.3 have shown that nesting outperforms all of the approaches.

PSTM improves on SwissTM [8], which has featured other optimizations in order to support longer transactions, like a contention manager and mixed concurrency control, and showed 2-3x better performance compared to TL2 [6] and TinySTM [12] and good scalability up to 8 threads. These optimizations are orthogonal to nesting.

**Chopping and nesting.** Recent works introduced the concept of *chopping* [43, 11, 45], which splits up transactions in order to reduce abort rates. Chopping and the similar concept of elastic transactions [13] were recently adopted in transactional memory [34, 44, 31]. The high-level idea of chopping is to divide a transaction into a sequence of smaller ones and commit them one at a time. While atomicity is eventually satisfied (provided that all transactions eventually commit), this approach forgoes isolation, which nesting preserves.

While some previous work on supporting nesting in generic STMs was done in the past [4, 42, 35, 3], we are not aware of any previous work implementing *closed nesting* in a non-distributed sequential STM. This might be due to the fact that the benefit of closed nesting is in allowing longer transactions whereas STMs are not particularly suitable for long transactions in any case, and the extra overhead associated with nesting might be excessive when read- and write-sets are large as in general purpose STMs. Our solution is also the first to introduce nesting into transactional data structure libraries, and thus the first to exploit the specific structure and semantics of data structures for efficient nesting. Because our data-structures use diverse concurrency control approaches, we had to develop nesting support for each of them. An STM using any of these approaches (e.g., fine-grain commit-time locking with read-/write-sets) can mimic our relevant technique (e.g., closed-nesting can be supported in TL2 using a similar scheme to the one we use in maps).

## 7 Conclusion

The TDSL approach enables high-performance software transactions by restricting transactional access to a well-defined set of data structure operations. Yet in order to be usable in practice, a TDSL needs to be able to sustain long transactions, and to offer a variety of data structures. In this work, we took a step towards boosting the performance and usability of TDSLs, allowing them to support complex applications. A key enabler for long transactions is nesting, which limits the scope of aborts without changing the transactional semantics.

We have implemented a Java TDSL with built-in support for nesting in a number of data structures. We conducted a case study of a complex network intrusion detection system running long transactions. We found that nesting improves performance by up to 8x, and the nested TDSL approach outperforms the general-purpose STM by up to 16x. We plan to make our code (both the library and the benchmark) available in open-source.

---

### References

- 1 David J Angel, James R Kumorek, Farokh Morshed, and David A Seidel. Byte code instrumentation, November 6 2001. US Patent 6,314,558.
- 2 Gal Assa, Hagar Meir, Guy Golan-Gueta, Idit Keidar, and Alexander Spiegelman. Using nesting to push the limits of transactional data structure libraries. *arXiv preprint arXiv:2001.00363*, 2021.
- 3 Joao Barreto, Aleksandar Dragojević, Paulo Ferreira, Rachid Guerraoui, and Michal Kapalka. Leveraging parallel nesting in transactional memory. In *ACM Sigplan Notices*, volume 45 (5), 2010.
- 4 Martin Bättig and Thomas R Gross. Encapsulated open nesting for STM: fine-grained higher-level conflict detection. In *PPoPP*, 2019.
- 5 Nathan G Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. In *ACM Sigplan Notices*, volume 45 (5). ACM, 2010.
- 6 Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *DISC*. Springer, 2006.
- 7 Dave Dice and Nir Shavit. Understanding tradeoffs in software transactional memory. In *CGO*. IEEE, 2007.
- 8 Aleksandar Dragojević, Rachid Guerraoui, and Michal Kapalka. Stretching transactional memory. *ACM sigplan notices*, 44, 2009.
- 9 Avner Elizarov and Erez Petrank. Loft: lock-free transactional data structures. Master's thesis, Computer Science Department, Technion, 2019.
- 10 Robert Ennals. Software transactional memory should not be obstruction-free. Technical report, IRC-TR-06-052, Intel Research Cambridge, 2006.
- 11 Jose M Faleiro, Daniel J Abadi, and Joseph M Hellerstein. High performance transactions via early write visibility. *VLDB*, 10(5), 2017.
- 12 Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *PPoPP*, 2008.
- 13 Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. Elastic transactions. In Idit Keidar, editor, *Distributed Computing*, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
- 14 OIS Foundation. Suricata. URL: <https://suricata-ids.org/>.
- 15 Vincent Gramoli. More than you ever wanted to know about synchronization: synchronbench, measuring the impact of the synchronization on concurrent algorithms. In *PPoPP*, 2015.
- 16 Vincent Gramoli and Rachid Guerraoui. Democratizing transactional programming. *Commun. ACM*, 57(1), January 2014.
- 17 Vincent Gramoli and Rachid Guerraoui. Reusable concurrent data types. In *ECOOP 2014 – Object-Oriented Programming*. Springer, 2014.
- 18 Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *PPoPP*. ACM, 2008.

## 30:16 Using Nesting to Push the Limits of Transactional Data Structure Libraries

- 19 Bart Haagdorens, Tim Vermeiren, and Marnix Goossens. Improving the performance of signature-based network intrusion detection sensors by multi-threading. In *WISA*. Springer, 2004.
- 20 Ahmed Hassan, Roberto Palmieri, and Binoy Ravindran. Optimistic transactional boosting. In *PPoPP*, 2014.
- 21 Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N Scherer, and Nir Shavit. A lazy concurrent list-based set algorithm. In *OPODIS*. Springer, 2005.
- 22 Maurice Herlihy. The transactional manifesto: software engineering and non-blocking synchronization. In *PLDI 2005*. ACM, 2005.
- 23 Maurice Herlihy and Eric Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *PPoPP*, 2008.
- 24 Maurice Herlihy, Victor Luchangco, Mark Moir, and William N Scherer III. Software transactional memory for dynamic-sized data structures. In *PODC*. ACM, 2003.
- 25 Maurice Herlihy and J Eliot B Moss. *Transactional memory: Architectural support for lock-free data structures*, volume 21 (2). ACM, 1993.
- 26 Nathaniel Herman, Jeevana Priya Inala, Yihe Huang, Lillian Tsai, Eddie Kohler, Barbara Liskov, and Liuba Shrira. Type-aware transactions for faster concurrent code. In *Eurosys*, 2016.
- 27 Guy Korland. Jstamp, 2014. URL: <https://github.com/DeuceSTM/DeuceSTM/tree/master/src/test/jstamp>.
- 28 Guy Korland, Nir Shavit, and Pascal Felber. Noninvasive concurrency with java STM. In *MULTIPROG*, 2010.
- 29 Pierre LaBorde, Lance Lebanoff, Christina Peterson, Deli Zhang, and Damian Dechev. Wait-free dynamic transactions for linked data structures. In *PMAM*, 2019.
- 30 Douglas Lea. *Concurrent programming in Java: design principles and patterns*. Addison-Wesley Professional, 2000.
- 31 Lance Lebanoff, Christina Peterson, and Damian Dechev. Check-wait-pounce: Increasing transactional data structure throughput by delaying transactions. In *IFIP International Conference on Distributed Applications and Interoperable Systems*. Springer, 2019.
- 32 Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *2008 IEEE International Symposium on Workload Characterization*, 2008.
- 33 John Eliot Blakeslee Moss. Nested transactions: An approach to reliable distributed computing. Technical report, MIT Cambridge lab, 1981.
- 34 Shuai Mu, Sebastian Angel, and Dennis Shasha. Deferred runtime pipelining for contentious multicore software transactions. In *EuroSys*. ACM, 2019.
- 35 Yang Ni, Vijay S Menon, Ali-Reza Adl-Tabatabai, Antony L Hosking, Richard L Hudson, J Eliot B Moss, Bratin Saha, and Tatiana Shpeisman. Open nesting in software transactional memory. In *PPoPP*, 2007.
- 36 Vern Paxson. Bro: a System for Detecting Network Intruders in Real-Time. *Computer Networks*, 31(23-24):2435–2463, 1999. URL: <http://www.icir.org/vern/papers/bro-CN99.pdf>.
- 37 Dmitri Perelman, Anton Byshevsky, Oleg Litmanovich, and Idit Keidar. Smv: selective multi-versioning stm. In *DISC*. Springer, 2011.
- 38 Martin Roesch et al. Snort: Lightweight intrusion detection for networks. In *Lisa*, volume 99, 1999.
- 39 Michael Scott. Transactional memory today. *SIGACT News*, 46(2), June 2015.
- 40 Nir Shavit and Itay Lotan. Skiplist-based concurrent priority queues. In *IPDPS 2000*. IEEE, 2000.
- 41 Alexander Spiegelman, Guy Golan-Gueta, and Idit Keidar. Transactional data structure libraries. In *PLDI 2016*. ACM, 2016.

- 42 Alexandru Turcu, Binoy Ravindran, and Mohamed M Saad. On closed nesting in distributed transactional memory. In *Seventh ACM SIGPLAN workshop on Transactional Computing*, 2012.
- 43 Zhaoguo Wang, Shuai Mu, Yang Cui, Han Yi, Haibo Chen, and Jinyang Li. Scaling multicore databases via constrained parallel execution. In *SIGMOD*, 2016.
- 44 Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using RDMA and HTM. In *SOSP*, 2015.
- 45 Chao Xie, Chunzhi Su, Cody Littlely, Lorenzo Alvisi, Manos Kapritsos, and Yang Wang. High-performance ACID via modular concurrency control. In *SOSP*, 2015.
- 46 Deli Zhang, Pierre Laborde, Lance Lebanoff, and Damian Dechev. Lock-free transactional transformation for linked data structures. *TOPC*, 5(1), 2018.







# Asynchronous Rumor Spreading in Dynamic Graphs

Bernard Mans  

Macquarie University, Sydney, Australia

Ali Pourmiri  

UNSW, Sydney, Australia

---

## Abstract

We study asynchronous rumor spreading algorithm in dynamic and static graphs. In the asynchronous rumor spreading, for a given underlying graph, each node is equipped with an exponential time clock of rate 1. When a node's clock ticks, the node calls a random neighbor in order to exchange a rumor, if at least one of them knows it. Assuming a single node knows a rumor, we apply a differential equation-based technique to obtain an upper bound for the *spread time* of the algorithm in general dynamic graphs, which is the first time when all nodes get informed with high probability. In particular, we derive an upper bound for the spread time of the algorithm in a discrete version of a geometric mobile network, introduced by Clementi et al. [7]. In this model, a set of  $n$  agents independently performs random walks on a  $\sqrt{n} \times \sqrt{n}$  plane and every two agents are able to communicate if they are within Euclidean distance at most  $R$ , where  $f(n)\sqrt{\log n} \leq R \leq \sqrt{n}$  and  $f(n)$  is a slowly growing function in  $n$ . Here, we show that the algorithm spreads a rumor through the network in  $\mathcal{O}(\log n + \sqrt{n}/R)$  time, with high probability. Although we only show an upper bound the spread time of the algorithm in a 2 dimensional space, the framework can be also applied for geometric mobile networks defined over higher dimensional space and other random dynamic evolving networks such as stationary edge-Markovian model. Besides these synchronous and discrete dynamic models, we also consider the spreading time in dynamical Erdős-Rényi graphs.

**2012 ACM Subject Classification** Mathematics of computing  $\rightarrow$  Probabilistic algorithms; Theory of computation  $\rightarrow$  Network flows

**Keywords and phrases** randomized rumor spreading, push/pull, asynchronous rumor spreading

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2021.31

**Funding** *Bernard Mans*: This work was supported by the Australian Research Council Grant DP170102794.

*Ali Pourmiri*: This work was supported by the Australian Research Council Grant DP170102794.

## 1 Introduction

Randomized rumor spreading algorithms are important primitives for information dissemination through a network. The standard randomized rumor spreading proceeds in succeeding rounds. In each round, every node in the network calls a random neighbor and they possibly exchange the rumor, if at least one of them knows it. Demers et al. [10] first introduced the algorithm to consistently distribute an update in a network of databases. Feige et al. [12] observed that the algorithm is scalable in terms of network size, and robust against the node/link failure and thus it has been applied in a wide range of distributed settings (e.g., see [3, 16]). The *spread time* is a well-studied parameter associated with the rumor spreading algorithms which is the first time when all nodes have been informed with high probability. The spread time of the algorithm has been studied on various network topologies [2, 11]. Moreover, it has been shown that the spread time of the algorithm in any static  $n$ -node network is at most  $\mathcal{O}(\log n/\Phi)$ , where  $\Phi$  denotes the graph conductance [5]. In many distributed networks such as peer-to-peer, social and ad hoc networks, agents may not act in a synchronized manner. Therefore, Boyd et al. [3] proposed the asynchronous randomized rumor spreading algorithm, where each node has its own clock and contacts a



© Bernard Mans and Ali Pourmiri;

licensed under Creative Commons License CC-BY 4.0

25th International Conference on Principles of Distributed Systems (OPODIS 2021).

Editors: Quentin Bramas, Vincent Gramoli, and Alessia Milani; Article No. 31; pp. 31:1–31:20

Leibniz International Proceedings in Informatics



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

random neighbor in order to exchange the rumor according to arrival times of its Poisson process with rate 1. The algorithm and its variations have been further studied in static networks [1, 13, 20, 23].

Information spreading in dynamic graphs has been a fundamental question and the subject of a large body of works (e.g., see [8] and references therein). For instance, the spread time of various algorithms has been studied in popular dynamic evolving graphs whose evolution is governed by a stochastic process such as *geometric mobile* [18, 21], *edge-Markovian* [6], and *node-Markovian* dynamic graphs [8]. Deterministic and adversarial settings have also been considered in [14, 22]. A dynamic evolving graph, denoted by  $\mathcal{G} = \{G^{(t)}\}_{t=0}^{\infty}$ , is usually referred to as a sequence of graphs with same set of nodes, but the edge set may change over discrete time  $t = 0, 1, \dots$ . It has gained popularity as it models a wide range of real-world networks including the wireless communication, mobile, and peer-to-peer networks.

## 1.1 Related Works

Kowalski and Caro [17] considered the asynchronous rumour spreading on general graphs and introduced a graphical quantity based on degree distribution of nodes that are incident to edges in any cut set. By applying the quantity they derive upper bounds for the spread time. Also, Panagiotou and Spiedel [20] rigorously analyzed the spread time of the asynchronous algorithm in Erdős-Rényi graphs  $G(n, p)$  with  $p = \omega(\log n/n)$ . In order to show their results, they presented a large deviation inequality for the sum of a particular set of exponential random variables, which cannot be generalized for every graph.

Giakkoupis et al. [13] applied coupling techniques and established an interesting relation between synchronous and asynchronous rumor spreading algorithms. Let  $G$  be a given  $n$ -node static network and assume that  $T_s(G)$  and  $T_a(G)$  are the spread time of synchronous and the standard asynchronous rumor spreading algorithms on  $G$ , respectively. They showed that  $T_a(G) = \mathcal{O}(T_s(G) + \log n)$ . Moreover, they derived an upper bound for  $\frac{T_s(G)}{T_a(G)}$ , which is  $n^{1/2}(\log n)^{\mathcal{O}(1)}$ . Giakkoupis et al. [14] considered the spread time of the synchronous rumor spreading in dynamic evolving graphs. They showed that the rumor propagates through the graph whenever  $\sum_t \{\Phi(G^{(t)}) \cdot D\} = \Omega(\log n)$ , where  $D = \max_u \delta_u / \Delta_u$ ,  $\Delta_u$  and  $\delta_u$  are the upper and lower bounds for degree of node  $u$  over time, respectively, and the maximum is taken over all nodes.

Pourmiri and Mans [22] established a similar upper bound for the spread time of asynchronous rumor spreading in dynamic graphs that is the first time when  $\sum_t \{\Phi(G^{(t)}) \cdot \rho(G^{(t)})\} = \Omega(\log n)$ , where  $\rho(G^{(t)})$  is called the *graph diligence*. The graph diligence presents a more refined version of parameter  $D$  and  $\rho(G^{(t)}) \geq \delta(t) / \Delta(t)$ , where  $\Delta(t)$  and  $\delta(t)$  denote the maximum and minimum degree of  $G^{(t)}$ , respectively. Moreover, they present a family of dynamic graphs for which the upper bounds is tight up to a  $o((\log n)^2)$  factor.

The aforementioned results have shown that besides the graph conductance, variation of degree sequence in a dynamic graph directly affect the spread time.

## 1.2 Our Main Results

We focus on asynchronous rumor spreading in dynamic and static graphs and present a general technique to obtain an upper bound for the spread time. The upper bounds are based on a differential equation taking into account the expansion properties of various subset of nodes, the maximum and the minimum degree of nodes. The methods have two advantages; (i) In contrast to existing method, this technique can be extended to settings where graphical parameters continuously or discretely change over time. (ii) It provides an alternative way

to compute the spread time by defining a Poisson random variable that is stochastically dominated by the size of informed nodes. The latter allows us to apply a concentration result for Poisson random variables and obtain a lower bound for the size of informed nodes at any time.

For every  $n$ -vertex graph  $G = (V, E)$  and  $1 \leq x < n$ , *conductance function*, denoted by  $\Phi(x)$ , measures the expansion property of any subset of nodes of size at most  $x$  in  $G$ , which is defined as follows

$$\Phi(x) = \min_{\substack{S \subset V(G) \\ 1 \leq |S| \leq x}} \frac{|E(S, \bar{S})|}{\min\{\text{vol}(S), \text{vol}(\bar{S})\}},$$

where  $\text{vol}(S) = \sum_{u \in S} d_u$  and  $E(S, \bar{S})$  denotes the set of edges crossing  $S$  and its complement  $\bar{S}$  (i.e.,  $V \setminus S$ ). It is easy to see that the standard graph conductance can be rewritten as  $\Phi(G) = \Phi(n/2)$ . Lovász and Kannan [19] introduced the concept of conductance function and showed that a lazy random walk on a connected graph with  $n$  nodes converges to its stationary distribution within at most  $\int_{1/n}^{1/2} dt / (t\Phi(nt)^2)$  time. Somewhat analogous to this result we estimate the number of informed nodes up to time  $t$  by a Poisson-distributed random variable with rate  $\Lambda(t)$ , where  $d\Lambda(t)/dt$  satisfies a differential equation (see Lemma 2.9).

Using the differential equation presented at Lemma 2.9, we derive upper bounds for the spread time of the asynchronous rumor spreading in a general dynamic evolving graph, geometric mobile, and dynamical Erdős-Rényi graphs. A dynamic evolving graph is a sequence of  $n$ -node graphs,  $G^{(1)}, G^{(2)}, \dots$ , where they all have the same set of nodes but set of edges changes over time  $t = 1, \dots$

► **Theorem 1.1.** *Suppose that  $\mathcal{G} = \{G^{(t)}\}_{t=1}^{\infty}$  denote a dynamic evolving graph whose nodes' degrees range over interval  $[\delta, \Delta]$ . Also, assume that graph exposed at any time  $t$ ,  $G^{(t)}$ , has conductance at least  $\Phi$ . Then, with high probability,*

$$T(\mathcal{G}) = \mathcal{O}(\Delta \log n / (\delta\Phi)),$$

where  $T(\mathcal{G})$  denote the first time when all nodes get informed.

► **Remark.** It turns out that the upper bound tight up to a  $o((\log n)^2)$ . In fact, there exists a dynamic evolving graph with  $\Delta/\delta = \Theta(\sqrt{n})$  and  $\Phi = \Theta(\log \log n / \log n)$  for which the rumor spreads in  $\Omega(\sqrt{n} / \log n)$  time. For more details see [22, Theorem 1.2].

### Geometric Mobile Network

The *geometric mobile stationary network*, introduced by Clementi et al. [7], is a discrete version of *random walk mobility* model, where nodes represent radio stations in a wireless communication system [4]. For some small number  $\epsilon > 0$ , initially,  $n$  agents are randomly distributed on nodes of a  $\sqrt{n}/\epsilon \times \sqrt{n}/\epsilon$  2-dimensional grid, embedded on a  $\sqrt{n} \times \sqrt{n}$  square plane. For a given parameter  $r > 0$ , in each time step  $t = 1, \dots$ , each agent independently and uniformly at randomly moves to a node whose Euclidean distance from its current location is at most  $r$ . Given this random process, in each time step  $t$ , we define network  $G^{(t)}$  whose vertex set is the set of all agents and there is an edge between any two agents in  $G^{(t)}$  if their Euclidean distance is at most  $R$ , where  $f(n)\sqrt{\log n} \leq R \leq \sqrt{n}$  in the plane, and  $f(n)$  is a slowly growing function in  $n$ . The model is denoted by  $\mathcal{M}(n, R) = \{G^{(t)}\}_{t=0}^{\infty}$  and it is assumed that the agents are initially distributed according to the stationary distribution of the random walk on the grid.

► **Theorem 1.2.** *Suppose that  $\mathcal{M}(n, R) = \{G^{(t)}\}_{t=0}^{\infty}$  is a geometric mobile network with  $f(n)\sqrt{\log n} \leq R \leq \sqrt{n}/2$ , where  $f(n)$  is a slowly growing function. Also, assume that, initially, a node of  $G^{(0)}$  is aware of a rumor. Then, with high probability, the asynchronous rumor spreading algorithm propagates the rumor in  $\mathcal{O}(\sqrt{n}/R + \log n)$  time.*

Interestingly, the upper bound has the same magnitude as the spread time of *flooding* in the network [7]. The flooding is a simple variant of the synchronous rumor spreading algorithm where each informed node pushes the rumor to all of its neighbors. For sufficiently large  $R = \Theta(\sqrt{n})$ ,  $\mathcal{M}(n, R)$  is almost fully connected and the theorem gives an upper bound of  $\mathcal{O}(\log n)$  for the algorithm, which is tight for the asynchronous rumor spreading in any fully connected network [3]. The proof technique of the theorem can be also applied for geometric mobile network defined over higher dimensional space.

### Dynamical Erdős-Rényi Graph

Häggström, Peres and Steif [15] introduced dynamical percolation graph by adding a time dynamics to the well-known percolation model. The model, initially, starts with a fixed underlying graph  $G$  whose edges have been associated with a Poisson clock of rate  $\mu$ . When an edge's clock ticks, then the edge is activated (opened) with probability  $p$  and deactivated (closed) with probability  $1 - p$ . Later, Sousi and Thomas [24] studied a setting where the underlying graph is an  $n$ -node complete graph,  $\mu = o((\log n)^{-6}/n)$  and  $p = c/n$ , where  $c > 1$  is a constant. The dynamic graph is called dynamical Erdős-Rényi graph  $ER(n, p, \mu)$  modeling a sparse dynamic graph whose edges get updated, slowly. They studied the mixing properties of a random walk on the graph and show that the random walk mixes in  $\frac{\log n}{\mu}(1 + o(1))$  time.

► **Theorem 1.3.** *Suppose that for some constant  $c > 1$ ,  $p = c/n$  and  $\mu = o((\log n)^{-6}/n)$ . Also, assume that initially a rumor is injected to a node of  $ER(n, \mu, p)$ . Then, with probability  $1 - o(1)$ , the rumor propagates through the  $ER(n, \mu, p)$  within  $\mathcal{O}((\log n)^2/\mu)$  time.*

A natural question would be to investigate the relation between the mixing and spread time in dynamic percolation graphs.

### Outline

In Section 2 we present useful definitions and some preliminaries. We prove Theorems 1.1, 1.2, in Sections 3 and 4, respectively. Also, we give a proof sketch for Theorem 1.3 in Section 5.

## 2 Notations and Preliminaries

In this section we first define notations and some useful preliminaries. Throughout this paper,  $n$  denotes the number of nodes in the dynamic or static graph and  $\log$  stands for the logarithm to the base of  $e$ . We say an event, say  $\mathcal{E}_n$ , holds with high probability, if  $\Pr[\mathcal{E}_n] \geq 1 - n^{-c}$ , for some constant  $c > 1$ . For the sake of brevity we use *w.h.p.* to denote with high probability. Now, let us formally present some definitions.

► **Definition 2.1** (Conductance function). *Let  $G = (V, E)$  be an  $n$ -vertex simple graph. Then, for every  $1 \leq x < n$ , conductance function is defined as*

$$\Phi(x) = \min_{\substack{S \subset V(G) \\ 1 \leq |S| \leq x}} \frac{|E(S, \bar{S})|}{\min\{\text{vol}(S), \text{vol}(\bar{S})\}}, \quad (1)$$

where  $E(S, \bar{S})$  is the set of edges crossing  $S$  and its complement and  $\text{vol}(S) = \sum_{u \in S} d_u$ .

► **Definition 2.2** (Asynchronous rumor spreading). *Suppose  $G$  is a graph whose nodes are associated with an exponential time clock of rate 1. Also, assume that initially a rumor is injected to a node of  $G$ . Each node contacts a random neighbor according to the arrival times of its Poisson process with rate 1. When they contact each other, they may learn the rumor, if at least one of them knows it. Also, we define the spread time as the first time when all nodes get informed with high probability and we use  $T(G)$  to denote the spread time.*

► **Definition 2.3** (Non-homogeneous Poisson process). *Suppose that for every  $\tau \geq 0$  there is a Poisson process with rate  $\lambda(\tau) \geq 0$ . Then,  $\mathbf{P} = \{\lambda(\tau) : \tau \geq 0\}$  is called a non-homogeneous Poisson or counting process. Also, let  $N(\tau)$  denote the number of occurrences made by process during  $[0, \tau]$ .*

► **Definition 2.4** (Stochastic Dominance). *We say random variable  $X$  stochastically dominates random variable  $Y$ , if for any arbitrary number  $a$ , we have that*

$$\Pr[X \leq a] \leq \Pr[Y \leq a].$$

We will now present a well-known theorem regarding non-homogeneous Poisson process.

► **Theorem 2.5** ([9, Chapter 2]). *Suppose that  $\mathbf{P} = \{\lambda(\tau) : \tau \geq 0\}$  is a non-homogeneous Poisson process. Also assume that  $\lambda(\tau) : [0, \infty) \rightarrow [0, \infty)$  is an integrable function. Then, for every  $0 \leq a \leq b$ ,  $N(b) - N(a)$  has a Poisson distribution with rate*

$$\Lambda = \int_a^b \lambda(\tau) d\tau.$$

For more information about non-homogeneous Poisson processes we refer the interested reader to [9]. We now present a large deviation bound for Poisson random variables, whose proof is based on the moment generating function of the Poisson random variables.

► **Theorem 2.6.** *Suppose that  $X$  denote a Poisson random variable with rate  $\Lambda$ . Then we have that*

$$\Pr[|X - \Lambda| \geq \eta] \leq 2 \cdot e^{\frac{-\eta^2}{2(\Lambda + \eta)}}.$$

Towards studying distribution of  $T(G)$ , we divide the asynchronous algorithm in  $n$  states where each state  $1 \leq j \leq n$  stands for the situation where we have  $j$  informed nodes. For every  $j = 1, \dots, n-1$ , define  $t_j(G)$  to be the waiting time for the algorithm to jump from the  $j$ -th state to the  $(j+1)$ -st one. Clearly, we have that

$$T(G) = \sum_{j=1}^{n-1} t_j(G).$$

► **Lemma 2.7.** *Suppose that  $G = (V, E)$  denote an  $n$ -node graph. Also, assume that, initially, rumor is injected to a node of  $G$ . Then, for every  $2 \leq j \leq n-1$ , conditional on the first  $j$  informed nodes, say  $I_j$ ,  $t_j(G)$  is an exponential random variable with rate*

$$\beta_j(G) = \sum_{\{u,v\} \in E(I_j, U_j)} \left\{ \frac{1}{d(u)} + \frac{1}{d(v)} \right\},$$

where  $E(I_j, U_j)$  is the set of edges crossing  $I_j$  and its complement  $U_j$  (set of non-informed nodes). Moreover  $t_j(G)$  is independent of  $t_{j-1}, \dots, t_1$ .

**Proof.** Notice that for every pair of vertices  $\{u, v\} \in E$ ,  $u$  and  $v$  contact each other with Poisson rate  $1/d(u) + 1/d(v)$ , where  $d(u)$  and  $d(v)$  are degrees of  $u$  and  $v$ , respectively. Let  $I_j$  denotes the set of first  $j$  informed nodes. Then the  $(j + 1)$ -st node gets informed with Poisson rate  $\beta_j(G)$ , which is

$$\beta_j(G) = \sum_{\{u,v\} \in E(I_j, U_j)} \left\{ \frac{1}{d(u)} + \frac{1}{d(v)} \right\},$$

As soon as the  $j$ -th node gets informed and  $I_j$  gets determined, by memory-less property of exponential distribution,  $t_j(G)$  is an exponentially distributed random variable which is independent of  $t_1(G), \dots, t_{j-1}(G)$ .  $\blacktriangleleft$

► **Lemma 2.8.** *Suppose that, for some  $n$ ,  $A = (\alpha_1, \dots, \alpha_n), B = (\beta_1, \dots, \beta_n) \in \mathbb{R}_+^n$  be two arbitrary vectors where for every  $1 \leq j \leq n$ ,  $\alpha_j \leq \beta_j$ . Also, let  $\mathbf{P}(A)$  and  $\mathbf{P}(B)$  denote non-homogeneous Poisson process for which,  $j = 1, \dots, n$ , the  $j$ -th event happens with rate  $\alpha_j$  and  $\beta_j$ , respectively. Then, during a given time interval, the number of events generated by  $\mathbf{P}(B)$  stochastically dominates the number of events generated by  $\mathbf{P}(A)$ .*

The proof is given in Appendix A.

## 2.1 Some Useful Lemmas

► **Lemma 2.9.** *Suppose that  $\mathcal{G} = \{G^{(t)}\}_{t=1}^{\infty}$  denotes an evolving dynamic graph whose nodes' degree range over  $[\delta, \Delta]$ . Also, let  $\Phi(x)$ ,  $1 \leq x \leq n$ , be a lower bound for the conductance function of any graph  $G^{(t)} \in \mathcal{G}$ . Now, assume that initially a rumor is injected to an arbitrary node and the asynchronous algorithm starts propagating the rumor. Then, the number of informed nodes up to time  $t$  stochastically dominates a Poisson distribution with rate  $\Lambda(t)$  satisfying at*

$$\Lambda'(t) = 2 \cdot (\delta/\Delta) \cdot \Phi(\min\{n - C(t), C(t)\}) \min\{C(t), n - C(t)\},$$

where  $C(t)$  counts the number of events happened by a Poisson distribution with rate  $\Lambda(t)$ . In particular,  $\Phi(x)$  can be replaced with any function  $F(x) \leq \Phi(x)$  where  $1 \leq x \leq n/2$ .

**Proof.** Let  $\beta_j$  denotes the Poisson rate at which the  $(j + 1)$ -st node gets informed. Also let  $I_j$  and  $U_j$  denote the set of first  $j$  informed and  $n - j$  uninformed nodes, respectively. At any time  $t$ , by Lemma 2.7, for every  $1 \leq j \leq n - 1$ , we get that

$$\beta_j = \sum_{\{u,v\} \in E_t(I_j, U_j)} \left\{ \frac{1}{d_t(u)} + \frac{1}{d_t(v)} \right\} \geq \frac{2|E_t(I_j, U_j)|}{\Delta},$$

where  $d_t(u)$  and  $d_t(v)$  denote the degree of  $u$  and  $v$  at time  $t$ . Also, the last inequality follows from  $1/d(u) + 1/d(v) \geq 2/\Delta$ . Note that  $d_t(u)$  and  $d_t(v)$  are not zero as they are incident to edge  $\{u, v\}$  crossing cut  $E_t(I_j, U_j)$ . By the lemma statement and the definition of conductance function, we have that

$$\begin{aligned} |E_t(I_j, U_j)| &\geq \Phi(\min\{j, n - j\})(\min\{\text{vol}(I_j), \text{vol}(U_j)\}) \\ &\geq \Phi(\min\{j, n - j\}) \cdot (\min\{j, n - j\}) \cdot \delta, \end{aligned}$$

where  $\text{vol}(I_j)$  and  $\text{vol}(U_j)$  denote the volume of sets  $I_j$  and  $U_j$  in  $G^{(t)}$  and hence lower bounded by  $I_j \cdot \delta$  and  $U_j \cdot \delta$ , respectively. Notice that one can replace  $\Phi(x)$  by any  $F(x) \leq \Phi(x)$  and the lower bound still holds. Therefore, for every  $1 \leq j \leq n - 1$ ,

$$\beta_j \geq 2(\delta/\Delta)\Phi(\min\{j, n - j\}) \cdot \min\{j, n - j\} \quad (2)$$

Define a non-homogeneous Poisson process  $\mathbf{P} = \{\lambda(t) : \tau \in [0, \infty)\}$ , where we have

$$\lambda(t) = \begin{cases} (2\delta/\Delta)\Phi(C(t))C(t) & \text{if } C(t) \leq n/2, \\ (2\delta/\Delta)\Phi(n - C(t))[n - C(t)] & n/2 < C(t) < n, \\ 0 & \text{otherwise,} \end{cases} \quad (3)$$

where  $C(t)$  counts the number of informed nodes during  $[0, t]$ . Lemma 2.8 and Inequality (2) together show that the number of informed nodes stochastically dominates the number of events happened by  $\mathbf{P}$ . Moreover, by Theorem 2.5 during any interval  $[0, t]$ ,  $C(t)$  has a Poisson distribution with rate  $\Lambda(t) = \int_0^t \lambda(z)dz$ . Applying the fundamental theorem of the calculus yields that, for  $1 \leq C(t) \leq n$ ,

$$\Lambda'(t) = (2\delta/\Delta)\Phi(\min\{n - C(t), C(t)\}) \min\{n - C(t), C(t)\}. \quad (4)$$

◀

The following useful lemma helps to approximate  $C(t)$  and apply Lemma 2.9.

► **Lemma 2.10.** *Suppose that  $(\log n)^{1.3} \leq \Lambda(t) \leq n - \sqrt{n(\log n)^{1.3}}$ . Then for any constant  $0 < \varepsilon < 1$ , we have that*

$$\Lambda'(t) \geq (1 - \varepsilon)(2\delta/\Delta) \cdot \Phi(\min\{\Lambda(t), n - \Lambda(t)\}(1 + \varepsilon)) \cdot \min\{\Lambda(t), n - \Lambda(t)\}.$$

*In particular,  $\Phi(x)$  can be replaced by any function  $F(x) \leq \Phi(x)$ ,  $1 \leq x < n$ .*

**Proof.** Recall that for every  $t > 0$ ,  $C(t)$  counts the number of events made by a Poisson distribution with rate  $\Lambda(t)$  during time interval  $[0, t]$ . Let us set  $\eta = \sqrt{\Lambda(t)} \cdot (\log n)^{1.1}$  and apply a concentration result (e.g., Theorem 2.6) for  $C(t)$  and obtain an estimation for  $C(t)$  as follows.

$$\Pr [|C(t) - \Lambda(t)| \geq \eta] \leq 2 \cdot e^{\frac{-\eta^2}{2(\Lambda(t) + \eta)}} = 2e^{\frac{-\Lambda(t)(\log n)^{1.1}}{4\Lambda(t)}} \leq 2e^{-(\log n)^{1.1}/4} = n^{-\omega(1)}. \quad (5)$$

By (5) we have that with probability  $1 - n^{-\omega(1)}$ ,

$$\Lambda(t) - \eta \leq C(t) \leq \Lambda(t) + \eta$$

and hence,

$$n - \Lambda(t) - \eta \leq n - C(t) \leq n - \Lambda(t) + \eta.$$

Combing the both inequalities implies that with high probability

$$\min\{n - \Lambda(t), \Lambda(t)\} - \eta \leq \min\{n - C(t), C(t)\} \leq \min\{n - \Lambda(t), \Lambda(t)\} + \eta.$$

Note that if  $(\log n)^{1.3} \leq \Lambda(t) \leq n/2$ , then we have that

$$\eta = \sqrt{\Lambda(t)(\log n)^{1.1}} \leq \frac{\Lambda(t)}{(\log n)^{.1}}.$$

Moreover, if  $n/2 \leq \Lambda(t) \leq n - \sqrt{n(\log n)^{1.3}}$ , then  $n - \Lambda(t) \geq \sqrt{n(\log n)^{1.3}}$  and

$$\eta = \sqrt{\Lambda(t)(\log n)^{1.1}} \leq \frac{\sqrt{n(\log n)^{1.3}}}{(\log n)^{.1}} \leq \frac{n - \Lambda(t)}{(\log n)^{.1}}$$

### 31:8 Rumor Spreading in Dynamic Graphs

Using the above two inequalities implies that  $\eta \leq \min\{\Lambda(t), n - \Lambda(t)\}/(\log n)^{.1}$ . Thus,

$$\begin{aligned} \min\{n - \Lambda(t), \Lambda(t)\} \left(1 - \frac{1}{(\log n)^{.1}}\right) &\leq \min\{n - C(t), C(t)\} \\ &\leq \min\{n - \Lambda(t), \Lambda(t)\} \left(1 + \frac{1}{(\log n)^{.1}}\right) \end{aligned} \quad (6)$$

Let us apply (6) in the differential equation presented at Lemma 2.9 implies that

$$\Lambda'(t) \geq 2 \cdot (\delta/\Delta) \cdot \Phi \left( \min\{n - C(t), C(t)\} \left(1 + \frac{1}{(\log n)^{.1}}\right) \right) \min\{C(t), n - C(t)\} \left(1 - \frac{1}{(\log n)^{.1}}\right).$$

We replace  $1/(\log n)^{.1}$  by any constant  $0 < \varepsilon < 1$  and the statement follows.  $\blacktriangleleft$

### 3 Spread Time and Graph Conductance

This section is devoted to the proof of Theorem 1.1 presenting an upper bound for the spread time in terms of graph conductance.

**Proof of Theorem 1.1.** By Lemma 2.9 we have that the number of informed nodes stochastically dominates a Poisson distribution with rate  $\Lambda(t)$  satisfying

$$\Lambda'(t) = (2\delta/\Delta) \cdot \Phi \cdot \min\{C(t), n - C(t)\}, \quad (7)$$

where  $C(t)$  denotes the number of events happened by a non-homogenous process  $\{\lambda(t) : t \geq 0\}$  and  $\lambda(t) = (2\delta/\Delta) \cdot \Phi \cdot \min\{C(t), n - C(t)\}$ . Therefore, the number of informed nodes during any interval  $[0, \tau]$  stochastically dominates a Poisson-distributed random variable with rate  $\int_0^\tau \lambda(t) dt$ . Moreover, using a large deviation result for Poisson-distributed random variables (e.g., Theorem 2.6), the number of events is concentrated around  $\int_0^\tau \lambda(t) dt$ . Therefore, by computing  $\int_0^\tau \lambda(t) dt$  one can obtain a lower bound for the number of informed nodes during time interval  $[0, \tau]$ , with high probability, and an upper bound for the spread time, consequently. To do so according to  $C(t)$ , we study the process in three consecutive phases.

■ **Initial phase.** This phase starts with  $C(t) = 1$  and ends when  $C(t)$  exceeds  $\log n$ . Let  $T_{init}$  be the time when the phase ends. By Theorem 2.5,  $C(t)$  is a Poisson random variable with rate  $\Lambda(t) = \int_0^t \lambda(z) dz$ .

$$\Lambda(t) = \int_0^t \lambda(z) dz \geq \int_0^t (2\delta/\Delta) \Phi dz = (2\delta/\Delta) \cdot \Phi \cdot t \quad (8)$$

Then, by setting  $t = 4\Delta \log n / (\delta\Phi)$ , we conclude that  $\lambda(t) \geq 8 \log n$ . Using a large deviation bound (see e.g., Theorem 2.6) we get that

$$\Pr \left[ C(t) \leq \Lambda(t) - \sqrt{\Lambda(t) 8 \log n} \right] \leq \exp\{-8 \log n / 4\} = n^{-2}.$$

Therefore, with high probability,  $T_{init} \leq \frac{4\Delta \log n}{\delta\Phi}$ .

■ **Middle Phase.** This phase starts with  $C(t) = \log n$  and ends when  $C(t)$  exceeds  $n/2$ . Let  $T_{mid}$  be the first time when the phase ends. Also, define  $t_0, t_1, t_2$  to be the first times that we have  $\Lambda(t_0) = (\log n)^{1.3}$ ,  $\Lambda(t_1) = n/2$  and  $\Lambda(t_2) = 2n/3$ , respectively. In this phase for every  $t \in [T_{init}, T_{mid}]$ , we have that

$$\Lambda(t) = \int_0^t \lambda(z) dz \geq \int_{T_{init}}^t (2\delta/\Delta) \cdot \Phi \cdot (\log n) dz = (2\delta/\Delta) \cdot \Phi \cdot (\log n) \cdot (t - T_{init}).$$



This implies that there exists

$$t_0 \leq T_{init} + \frac{\Delta(\log n)^3}{2\delta\Phi} \leq \mathcal{O}\left(\frac{\Delta \log n}{\delta\Phi}\right) \quad (9)$$

for which we have  $\Lambda(t_0) = (\log n)^{1.3}$ . Applying Lemma 2.10 and using the fact that  $\Phi \leq \Phi(x)$ ,  $1 \leq x < n$ , we have that for every  $(\log n)^{1.3} \leq \Lambda(t) \leq n - \sqrt{(\log n)^{1.3}n}$ ,

$$\frac{\Lambda'(t)}{\min\{\Lambda(\tau), n - \Lambda(\tau)\}} \geq (2(1 - \varepsilon)\delta/\Delta)\Phi,$$

where  $0 < \varepsilon < 1$  is an arbitrary constant. Taking the integral from both sides, on interval  $[t_0, t]$ , and setting appropriate integral constants we get that

$$\Lambda(t) \geq \begin{cases} \exp\left\{\frac{2(1-\varepsilon)\delta\Phi}{\Delta}(t - t_0) + (1.3) \log \log n\right\} & (\log)^{1.3} \leq \Lambda(t) \leq n/2 \\ n - \exp\left\{\frac{-2\delta(1-\varepsilon)\Phi}{\Delta}(t - t_1) + \log(n/2)\right\} & n/2 \leq \Lambda(t) \leq n - \sqrt{n(\log n)^{1.3}} \end{cases}$$

where  $t_1$  is the first time when  $\Lambda(t_1) = n/2$ . From the first row in the above piecewise function we conclude that there exist

$$t_1 \leq \frac{\Delta \log(n/2)}{2(1 - \varepsilon)\delta\Phi} + t_0 = \mathcal{O}\left(\frac{\Delta \log n}{\delta\Phi}\right),$$

where the last equality follows from (9). Considering the second row and the previous equality we deduce that there exists

$$t_2 = \mathcal{O}\left(\frac{\Delta \log n}{2(1 - \varepsilon)\delta\Phi}\right) + t_1 = \mathcal{O}\left(\frac{\Delta \log n}{\delta\Phi}\right)$$

with  $\Lambda(t_2) \geq 2n/3$ . Since  $C(t)$  has Poisson rate  $\Lambda(t)$ , by using a large deviation inequality we get that

$$\Pr[C(t_2) \leq n/2] \leq \Pr\left[C(t_2) \leq \Lambda(t_2) - \log n \sqrt{\Lambda(t_2)}\right] \leq n^{-\omega(1)}.$$

Therefore, w.h.p.,  $T_{mid} \leq t_2 = \mathcal{O}\left(\frac{\Delta \log n}{\delta\Phi}\right)$ .

- **Final Phase.** This phase starts with  $C(t) = n/2$  and ends when  $C(t) = n$ . Let  $T_{final}$  denote the time when the phase ends. Notice that by definition of  $\mathbf{P}$ , the process is symmetric in  $C(t)$ , as  $\lambda(t)$  is proportional to  $1 \leq \min\{C(t), n - C(t)\} \leq n/2$ . Considering the time interval for which the process starts at  $C(t) = \lceil n/2 \rceil$  and ends at  $C(t) = n$ . The length of this interval has the same distribution as the time that  $\mathbf{P}$  requires to start from  $C(t) = 1$  and reach to the  $C(t) = \lfloor n/2 \rfloor$ . Therefore, from the previous phases we have that, with high probability,

$$T_{final} - T_{mid} = \mathcal{O}\left(\frac{\Delta \log n}{\delta\Phi}\right).$$

The number of informed nodes up to time  $t$ ,  $I(t)$ , stochastically dominates  $C(t)$ . Thus

$$\Pr[I(T_{final}) < n] \leq \Pr[C(T_{final}) < n] = n^{-\omega(1)}$$

Therefore, with high probability  $T(G) = \mathcal{O}\left(\frac{\Delta \log n}{\delta\Phi}\right)$ . ◀

## 4 Geometric Mobile Networks

In this section, we prove Theorem 1.2 presenting an upper bound for the spread time of the asynchronous rumor spreading in a geometric mobile network introduced by Clementi et al. [7]. Let us first present the following lemma regarding some useful properties of the dynamic graph whose proof is deferred to Appendix B.

► **Lemma 4.1.** *Suppose that  $\mathcal{M}(n, R) = \{G^{(t)}\}_{t=0}^{\infty}$ , is a geometric mobile network with  $f(n)\sqrt{\log n} \leq R < \sqrt{n}$ , where  $f(n)$  is a slowly growing function in  $n$ . Then, with probability  $1 - n^{-\omega(1)}$ , for every  $1 \leq t \leq n^3$ , the followings hold:*

1. *For every node  $u$ ,  $d_u(t) = \Theta(R^2)$ , where  $d_u(t)$  is the degree of node  $u$  in  $G^{(t)}$ .*
2. *There exists a constant  $a > 0$  such that conductance function  $G^{(t)}$  satisfies*

$$\Phi(x) \geq \begin{cases} a & 1 \leq x \leq R^2, \\ a \frac{R}{\sqrt{\min\{x, n-x\}}} & R^2 < x \leq n-1. \end{cases}$$

**Proof of Theorem 1.2.** Similar to the proof of Theorem 1.1, applying Lemma 2.9 results that the number of informed nodes stochastically dominates a Poisson distribution with rate  $\Lambda(t)$  satisfying

$$\Lambda'(t) = (2\delta/\Delta) \cdot \Phi(\min\{C(t), n - C(t)\}) \cdot \min\{C(t), n - C(t)\},$$

where  $C(t)$  denotes the number of events happened by a non-homogenous process  $\{\lambda(t) : t \geq 0\}$  and  $\lambda(t) = (2\delta/\Delta) \cdot \Phi(\min\{C(t), n - C(t)\}) \cdot \min\{C(t), n - C(t)\}$ . Therefore the number of informed nodes during any interval  $[0, \tau]$  stochastically dominates a Poisson-distributed random variable with rate  $\int_0^\tau \lambda(t) dt$ . On the other hand, a large deviation bound (e.g., see Theorem 2.6) for the Poisson-distributed random variable shows that the number of events is concentrated around  $\int_0^\tau \lambda(t) dt$ , with high probability. Therefore, one can obtain a lower bound for the number informed nodes during time interval  $[0, \tau]$  by approximating  $\int_0^\tau \lambda(t) \cdot dt$ . In what follows, by a case analysis according to  $C(t)$  we estimate  $\int_0^\tau \lambda(t) \cdot dt$  and apply the large deviation bound to obtain an upper bound for the spread time, with high probability.

By Lemma 4.1, we observe that, w.h.p., at any time step  $t$ ,  $1 \leq t \leq n^3$ ,  $G^{(t)}$  is almost regular. Thus the minimum degree of  $G^{(t)}$  over its maximum degree is a constant and we have  $\delta/\Delta = b = \Theta(1)$ . The lemma also gives a lower bound for the conductance function  $\Phi(x)$ . Conditioning on the mentioned properties about  $G^{(t)}$ 's,  $1 \leq t \leq n^3$  that hold with probability  $1 - n^{-\omega(1)}$ , one can apply Lemma 2.9 and conclude that

$$\Lambda'(t) = \begin{cases} c_1 C(t) & 1 \leq C(t) \leq \min\{R^2, n/2\} \\ c_1 \sqrt{\min\{C(t), n - C(t)\}} & \min\{R^2, n/2\} < C(t) \leq n-1, \end{cases} \quad (10)$$

where  $R^2 < n/2$ ,  $c_1 = 2 \cdot a \cdot b$  is a constant and  $a$  appeared in Lemma 4.1. Moreover, applying Lemma 2.10 implies that for every constant  $0 < \varepsilon_0 < 1$  we have that

$$\Lambda'(t) \geq \frac{1 - \varepsilon_0}{\sqrt{1 + \varepsilon_0}} \begin{cases} c_1 \Lambda(t) & (\log n)^{1.1} \leq \Lambda(t) \leq \max\{R^2, (\log n)^{1.1}\}, \\ c_1 R \sqrt{\min\{\Lambda(t), n - \Lambda(t)\}} & \max\{R^2, (\log n)^{1.1}\} < \Lambda(t) \leq n-1, \end{cases}$$

In order to have a simpler form we set  $1 - \varepsilon = \frac{1 - \varepsilon_0}{\sqrt{1 + \varepsilon_0}}$ . Therefore, if  $\max\{R^2, (\log n)^{1.1}\} < \Lambda(t) \leq n-1$ , then we have that

$$\frac{\Lambda'(t)}{\sqrt{\min\{\Lambda(t), n - \Lambda(t)\}}} \geq (1 - \varepsilon) c_1 R. \quad (11)$$

Note that the number of informed nodes up to time  $t$  stochastically dominates  $C(t)$ . Therefore, in what follows we analyze  $C(t)$  in three consecutive phases.

- **Initial phase.** This phase starts with  $C(t) = 1$  and finishes when  $C(t)$  exceed  $R^2$ . Let  $T_{init}$  be the first time when this phase ends. By (10) we get that  $\Lambda'(t) = c_1 C(t)$ . Therefore, if  $C(t) = j$ , then the  $(j + 1)$ -st event happens with Poisson rate at least  $c_1 \cdot j$ . Let  $s_j$  denote the waiting time for moving from  $C(t) = j$  to  $C(t) = j + 1$ . Also, let  $m = \lfloor R^2 \rfloor$  and define  $S_m = \sum_{j=1}^{m-1} s_j$ . Since  $s_j$  has an exponential distribution with rate  $c_1 j$ , we conclude that

$$\mathbf{E}[s_j] = \mathbf{E}[\mathbf{E}[s_j | C(t) = j]] \leq \mathbf{E}\left[\frac{1}{c_1 j}\right] = \frac{1}{c_1 j}$$

By linearity of expectation we get that

$$\mathbf{E}[S_m] = \sum_{j=1}^m \mathbf{E}[s_j] \leq \sum_{j=1}^m \frac{1}{c_1 j} = H_m / c_1 = (\log m) / c_1 + \mathcal{O}(1)$$

where  $H_m$  is the  $m$ -th harmonic number and we have that  $H_m = \log m + \mathcal{O}(1)$ . By a large deviation inequality for this particular set of exponential random variables (e.g, see Lemma A.1) we can show that, for every  $c \geq 0$ ,  $\Pr[S_m \geq \log m + c \log n] \leq n^{-c}$ . Therefore, with high probability,

$$T_{init} \leq (\log m) / c_1 + c \log n = \mathcal{O}(\log n). \quad (12)$$

- **Middle phase.** This phase starts with  $C(t) = R^2$  and ends when  $C(t)$  exceeds  $n/2$ . Let  $T_{mid}$  be the first time when this phase ends. Define  $t_0, t_1, t_2$  to be the first times that we have  $\Lambda(t_0) = \max\{R^2, (\log n)^{1.3}\}$ ,  $\Lambda(t_1) = n/2$  and  $\lambda(t_2) = 2n/3$ . By (10) we have that for every  $R^2 \leq C(t) \leq n/2$ ,

$$\Lambda(t_0) = \int_0^{t_0} c_1 R \sqrt{C(t)} dt \geq c_1 R^2 (t_0 - T_{init}),$$

where the lower bound follows from the fact that  $C(t) \geq R^2$ . The presented lower bound implies that there exists  $t_0$  such that  $\Lambda(t_0) = \max\{R^2, (\log n)^{1.3}\}$ . Moreover, we have that  $R^2 \geq (\log n)$  and hence

$$t_0 \leq T_{init} + (\log n)^{1.3} = \mathcal{O}(\log n). \quad (13)$$

Using the fact that for every  $t \geq t_0$ ,  $\Lambda(t) \geq \max\{R^2, (\log n)^{1.3}\}$ , we integrate from both sides of (11) and we have that if  $t_0 \leq t \leq t_1$  we have that  $\max\{R^2, (\log n)^{1.3}\} \leq \Lambda(t) \leq n/2$ . Thus,

$$\int_{t_0}^{t_1} \frac{\Lambda'(t) dt}{\sqrt{\Lambda(t)}} = 2\sqrt{\Lambda(t_1)} - 2\sqrt{\Lambda(t_0)} \geq \int_{t_0}^{t_1} (1 - \varepsilon) c_1 R dt = (1 - \varepsilon) c_1 R (t_1 - t_0).$$

If  $t_1 \leq t \leq t_2$ , then we have that  $n/2 \leq \Lambda(t) \leq 2n/3$  and hence

$$\int_{t_1}^{t_2} \frac{\Lambda'(t) dt}{\sqrt{n - \Lambda(t)}} = 2\sqrt{n - \Lambda(t_1)} - 2\sqrt{n - \Lambda(t_2)} \geq \int_{t_1}^{t_2} (1 - \varepsilon) c_1 R dt = (1 - \varepsilon) c_1 R (t_2 - t_1)$$

Recall that we have defined  $\Lambda(t_0) = \max\{R^2, (\log n)^{1.3}\}$ ,  $\Lambda(t_1) = n/2$  and  $\lambda(t_2) = 2n/3$ . Considering the above lower bounds, one can observe that  $t_1 - t_0 = \mathcal{O}(\sqrt{n}/R)$  and  $t_2 - t_1 = \mathcal{O}(\sqrt{n}/R)$ . Therefore,

$$t_2 = (t_2 - t_1) + (t_1 - t_0) + t_0 = \mathcal{O}(\sqrt{n}/R) + t_0 = \mathcal{O}(\sqrt{n}/R) + \mathcal{O}(\log n),$$

## 31:12 Rumor Spreading in Dynamic Graphs

where the equality follows from (13). Since  $C(t_2)$  is distributed as a Poisson random variable with rate  $\Lambda(t_2) = 2n/3$ , by using Theorem 2.6 we have that

$$\Pr[C(t_2) < n/2] \leq \Pr[|C(t_2) - 2n/3| \geq n/6] = n^{-\omega(1)}.$$

Then, w.h.p.  $T_{mid} \leq t_2 = \mathcal{O}(\sqrt{n}/R + \log n)$ .

- **Final phase.** This phase starts with  $C(t) = n/2$  and ends when  $C(t) = n$ . Let  $T_{final}$  be the time when the phase ends. The analysis makes use of the fact that Poisson process with rate  $\Lambda(t)$  is symmetric in  $C(t)$ . Therefore, similar to the the final phase in the proof of Theorem 1.1, we have that w.h.p  $T_{final} - T_{mid} = \mathcal{O}(\sqrt{n}/R) + \log n$  and hence  $T_{final} = \mathcal{O}(\sqrt{n}/R + \log n)$ .

Using the fact that the number of informed node up to time  $t$ , say  $I(t)$ , dominates  $C(t)$  we have that

$$\Pr[C(T_{final}) < n] \leq \Pr[I(T_{final}) < n]$$

Therefore, w.h.p. the spread time in  $\mathcal{M}(n, R)$  is bounded by  $T_{final} = \mathcal{O}(\sqrt{n}/R + \log n)$ . ◀

## 5 Dynamical Erdős-Rényi Graphs

In this section we provide a proof sketch for Theorem 1.3 which presents an upper bound for the spread time in a dynamical Erdős-Rényi graph  $ER(n, p, \mu)$ . Before that we need some properties of the graph that have been shown in [24]. Recall  $ER(n, p, \mu)$  is a continuous Markov chain whose stationary distribution is a random graph distributed as Erdős-Rényi graph  $G(n, p)$ .

► **Definition 5.1** ([24, Definition 2.2]). *For a specified constant  $c$ , we say that graph  $G = (V, E)$  is good and we write  $G \in \mathcal{H}(c)$ , if  $G$  has a unique connected component  $\mathcal{C}$  with  $|\mathcal{C}| \geq c \cdot \log n$ , and we call it the giant component and satisfies the following properties.*

- *Size.* We have  $|\mathcal{C}| \geq c \cdot n$ .
- *Maximum degree.* The maximum degree of  $\mathcal{C}$  is at most  $c \log n$ .
- *Number of edges.* There are at most  $cn$  edges in  $\mathcal{C}$
- *Expansion properties.* We have that  $\Phi_{\mathcal{C}} \geq c(\log n)^{-2}$ , where  $\Phi_{\mathcal{C}}$  is the conductance of the giant component.

► **Proposition 5.2** ([24, Proposition 2.3]). *For any graph  $G$  sampled from the stationary distribution of dynamical Erdős-Rényi process. Then, we have that  $\Pr[G \notin \mathcal{H}(c)] = \mathcal{O}(n^{-9})$ .*

**Proof of Theorem 1.3.** We analyze the algorithm in three consecutive phases. For every  $\tau > 0$ , let  $\mathcal{G}(\tau)$  be the dynamical Erdős-Rényi graph at time  $\tau$ . Also, let  $T_{mix}$  denote the mixing time of the dynamical Erdős-Rényi graph, which is  $T_{mix} = (2 + o(1)) \frac{\log n}{\mu}$  (e.g., see [24]).

- **Informing a node of the giant component.** This phase starts with one single informed node and ends when a node of giant component knows the rumor. Let  $T_1$  be the first time when this phase ends. For each  $k = 1, \dots$ , define  $\tau_k = k \cdot T_{mix}$ . By Proposition 5.2,  $\mathcal{G}(\tau_k) \in \mathcal{H}(c)$  and hence it has a unique giant component of size at least  $c \cdot n$ . Therefore, at time  $\tau_k$ , the informed node is not included in the giant component with probability at least  $(1 - c)^k$ . Hence, we deduce that with probability at least  $1 - 1/n$  after at most  $(\log n/c)T_{mix} = \frac{(\log n)^2}{c\mu}(2 + o(1))$  time this phase ends.

- **Informing all nodes within the giant component.** This phase starts with a single informed node that belongs to the giant component and ends when all nodes in the giant component get informed. Let  $T_2$  denotes the time this phase requires to finish. By Proposition 5.2, the giant has at most  $cn$  edges. Thus, the first edge in the component gets updated with Poisson rate  $cn\mu = \mathcal{O}((\log n)^{-6})$ . This implies that with probability  $1 - o(1)$ , during time  $[T_1, T_1 + (\log n)^5]$ , the component remains connected and will be the same during the time interval. Applying Theorem 1.1 implies that rumor spreads through the giant component within  $\mathcal{O}\left(\frac{\Delta \log n}{\delta \Phi}\right)$  time. By Proposition 5.2, with probability  $1 - \mathcal{O}(n^{-9})$ , the giant component is a good graph and hence we have that  $\delta = 1$ ,  $\Delta = \mathcal{O}(\log n)$  and  $\Phi_C = \Omega((\log n)^{-2})$ . Thus, with probability  $1 - o(1)$ , the rumor spreads through the giant component within  $T_2 - T_1 = \mathcal{O}((\log n)^4)$  time.
- **Informing rest of the nodes.** This phase starts with at least  $cn$  informed nodes, which is the size of giant component. Let  $\mathbf{U}(\tau)$  and  $\mathbf{I}(\tau)$  be the set of informed and non-informed nodes up to time  $\tau$ . Let  $T_3$  denote the time when this phase ends. For each  $k = 1, \dots$ , define  $\tau_k = T_2 + k \cdot T_{mix}$  and suppose that  $u \in \mathbf{U}(\tau_k)$ . It is worth mentioning that definition of  $\tau_k$  allows us to have a new sample of  $G(n, p)$  and for each  $\tau_k$ ,  $\mathcal{G}(\tau_k)$  is distributed as  $G(n, p)$ . For every  $k = 1, \dots$ , define random variables  $X_u(k)$  to be the number of informed nodes that are adjacent to  $u$  at time  $\tau_k$ . Also  $|\mathbf{I}(\tau_k)| \geq c \cdot n$  is non-decreasing function in time. Therefore,  $X_u(k)$  dominates binomial random variable  $X \sim \text{Bin}(cn, p)$ . Then, for every  $0 < \theta < 1$ , by Zygmund-Paley inequality we have that

$$\begin{aligned} \Pr[X_u(k) > \theta cnp] &\geq \Pr[X_u(k) > \theta cnp] \\ &\geq (1 - \theta)^2 \frac{(cnp)^2}{cnp(1 - p) + (cnp)^2} \\ &\geq (1 - \theta)^2 \frac{cnp}{1 + cnp}. \end{aligned}$$

Setting  $\theta = 1/2$  and using the fact that  $p = \Theta(1/n)$ , we conclude that

$$\Pr[X_u(k) > 0] \geq \frac{cnp}{4(1 + cnp)} = c_1,$$

where  $c_1$  is a constant. Therefore, with probability at least  $c_1$ , for every  $k = 1, \dots$ ,  $u \in \mathbf{U}(\tau_k)$  is connected to some informed node. Fixing arbitrary  $u \in \mathbf{U}(\tau_1)$ , the probability that at time  $\tau_k$ ,  $u$  is not connected to some informed node is at least  $(1 - c_1)^k$ . Thus, by setting  $k = 2 \log n / c_1$ , with probability at most  $n^{-2}$ ,  $u$  is not connected. By union bound over all non-informed nodes, we conclude that, with probability  $1 - 1/n$ , every non-informed node is being connected to some informed one before time  $\tau_k$ . Provided  $u \in \mathbf{U}(\tau)$  has an informed neighbor, say  $v$ , they share the rumor with rate  $1/d(u) + 1/d(v)$ . Notice that for every  $k$ ,  $\mathcal{G}(\tau_k) \sim G(n, p)$  and hence its maximum degree is at most  $\mathcal{O}(\log n)$ . Therefore,  $u$  and  $v$  communicates with rate at least  $\Omega(1/\log n)$ . So with high probability during a period of length at most  $(\log n)^3$ ,  $u$  gets informed from its neighbor  $v$ , which follows from a concentration result for a Poisson random variable of rate  $\Omega(1/\log n) \cdot (\log n)^3 \geq (\log n)^{3/2}$ .

Note that edge  $\{u, v\}$  may disappear with rate  $\mu$ , however, during a time interval of length  $(\log n)^3$ , the edge disappear with probability  $\mu(\log n)^3 = o(1)$ . Therefore, with probability  $1 - o(1)$  after at most  $T_2 + (2 \log n / c_1)T_{mix} + \mathcal{O}((\log n)^3)$  time every node gets informed. From the first and second phases  $T_2 = T_1 + \mathcal{O}((\log n)^4) = \mathcal{O}(\log n T_{mix})$  and hence, with probability  $1 - o(1)$ , the rumor spreads in  $\mathcal{O}((\log n)^2 / \mu)$  time. ◀

## References

- 1 Hüseyin Acan, Andrea Collecchio, Abbas Mehrabian, and Nick Wormald. On the push&pull protocol for rumour spreading: [extended abstract]. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastián, Spain, July 21 - 23, 2015*, pages 405–412, 2015. doi:10.1145/2767386.2767416.
- 2 Petra Berenbrink, Robert Elsässer, and Tom Friedetzky. Efficient randomised broadcasting in random regular networks with applications in peer-to-peer systems. In *Proc. 27th Symp. Principles of Distributed Computing (PODC)*, pages 155–164, 2008.
- 3 Stephen Boyd, Arpita Ghosh, Balaji Prabhakar, and Devavrat Shah. Randomized gossip algorithms. *IEEE Transactions on Information Theory*, 52(6):2508–2530, 2006.
- 4 Tracy Camp, Jeff Boleng, and Vanessa Davies. A survey of mobility models for ad hoc network research. *Wireless Communications and Mobile Computing*, 2(5):483–502, 2002. doi:10.1002/wcm.72.
- 5 Flavio Chierichetti, George Giakkoupis, Silvio Lattanzi, and Alessandro Panconesi. Rumor spreading and conductance. *J. ACM*, 65(4):17:1–17:21, 2018. doi:10.1145/3173043.
- 6 Andrea E. F. Clementi, Pierluigi Crescenzi, Carola Doerr, Pierre Fraigniaud, Marco Isopi, Alessandro Panconesi, Francesco Pasquale, and Riccardo Silvestri. Rumor spreading in random evolving graphs. In *Algorithms - ESA 2013 - 21st Annual European Symposium, Sophia Antipolis, France, September 2-4, 2013. Proceedings*, pages 325–336, 2013. doi:10.1007/978-3-642-40450-4\_28.
- 7 Andrea E. F. Clementi, Angelo Monti, Francesco Pasquale, and Riccardo Silvestri. Information spreading in stationary markovian evolving graphs. *IEEE Trans. Parallel Distrib. Syst.*, 22(9):1425–1432, 2011. doi:10.1109/TPDS.2011.33.
- 8 Andrea E. F. Clementi, Riccardo Silvestri, and Luca Trevisan. Information spreading in dynamic graphs. In *ACM Symposium on Principles of Distributed Computing, PODC '12, Funchal, Madeira, Portugal, July 16-18, 2012*, pages 37–46, 2012. doi:10.1145/2332432.2332439.
- 9 D. J. Daley and D. Vere-Jones. *An introduction to the theory of point processes. Vol. I. Probability and its Applications* (New York). Springer-Verlag, New York, second edition, 2003. Elementary theory and methods.
- 10 Alan Demers, Mark Gealy, Dan Greene, Carl Hauser, Wes Irish, John Larson, Sue Manning, Scott Shenker, Howard Sturgis, Dan Swinehart, Doug Terry, and Don Woods. Epidemic algorithms for replicated database maintenance. In *Proc. 6th Symp. Principles of Distributed Computing (PODC)*, pages 1–12, 1987.
- 11 Benjamin Doerr, Mahmoud Fouz, and Tobias Friedrich. Social networks spread rumors in sublogarithmic time. In *Proc. 43th Symp. Theory of Computing (STOC)*, pages 21–30, 2011.
- 12 Uriel Feige, David Peleg, Prabhakar Raghavan, and Eli Upfal. Randomized broadcast in networks. *Random Struct. Algorithms*, 1(4):447–460, 1990.
- 13 George Giakkoupis, Yasamin Nazari, and Philipp Woelfel. How asynchrony affects rumor spreading time. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, July 25-28, 2016*, pages 185–194, 2016. doi:10.1145/2933057.2933117.
- 14 George Giakkoupis, Thomas Sauerwald, and Alexandre Stauffer. Randomized rumor spreading in dynamic graphs. In *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part II*, pages 495–507, 2014. doi:10.1007/978-3-662-43951-7\_42.
- 15 Olle Häggström, Yuval Peres, and Jeffrey E. Steif. Dynamical percolation. *Ann. Inst. H. Poincaré Probab. Statist.*, 33(4):497–528, 1997. doi:10.1016/S0246-0203(97)80103-3.
- 16 Mor Harchol-Balter, Frank Thomson Leighton, and Daniel Lewin. Resource discovery in distributed networks. In *Proc. 18th Symp. Principles of Distributed Computing (PODC)*, pages 229–237, 1999.

- 17 Dariusz R. Kowalski and Christopher Thraves Caro. Estimating time complexity of rumor spreading in ad-hoc networks. In Jacek Cichon, Maciej Gebala, and Marek Klonowski, editors, *Ad-hoc, Mobile, and Wireless Network - 12th International Conference, ADHOC-NOW 2013, Wroclaw, Poland, July 8-10, 2013. Proceedings*, volume 7960 of *Lecture Notes in Computer Science*, pages 245–256. Springer, 2013.
- 18 Henry Lam, Zhenming Liu, Michael Mitzenmacher, Xiaorui Sun, and Yajun Wang. Information dissemination via random walks in  $d$ -dimensional space. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, Kyoto, Japan, January 17-19, 2012*, pages 1612–1622, 2012. doi:10.1137/1.9781611973099.128.
- 19 László Lovász and Ravi Kannan. Faster mixing via average conductance. In Jeffrey Scott Vitter, Lawrence L. Larmore, and Frank Thomson Leighton, editors, *Proceedings of the Thirty-First Annual ACM Symposium on Theory of Computing, May 1-4, 1999, Atlanta, Georgia, USA*, pages 282–287. ACM, 1999. doi:10.1145/301250.301317.
- 20 Konstantinos Panagiotou and Leo Speidel. Asynchronous rumor spreading on random graphs. In *Algorithms and Computation - 24th International Symposium, ISAAC 2013, Hong Kong, China, December 16-18, 2013, Proceedings*, pages 424–434, 2013. doi:10.1007/978-3-642-45030-3\_40.
- 21 Alberto Pettarin, Andrea Pietracaprina, Geppino Pucci, and Eli Upfal. Tight bounds on information dissemination in sparse mobile networks. In *Proceedings of the 30th Annual ACM Symposium on Principles of Distributed Computing, PODC 2011, San Jose, CA, USA, June 6-8, 2011*, pages 355–362, 2011. doi:10.1145/1993806.1993882.
- 22 Ali Pourmiri and Bernard Mans. Tight analysis of asynchronous rumor spreading in dynamic networks. In *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2020, Salerno, Italy, 3-7 August 2020*, pages 263–272, 2020.
- 23 Ali Pourmiri and Fahimeh Ramezani. Ultra-fast asynchronous randomized rumor spreading (brief announcement). In *The 31st ACM on Symposium on Parallelism in Algorithms and Architectures, SPAA 2019, Phoenix, AZ, USA, June 22-24, 2019.*, pages 81–83, 2019. doi:10.1145/3323165.3323167.
- 24 Perla Sousi and Sam Thomas. Cutoff for random walk on dynamical erdős-rényi graph. *arXiv*, 2018. arXiv:1807.04719.

## A Missing Proofs of Section 2

**Proof of Lemma 2.8.** For very  $j = 1, \dots, n - 1$ , define  $s_j$  and  $t_j$  to be exponentially distributed random variables with rates  $\alpha_j$  and  $\beta_j$ , respectively. Define  $X = (s_1, \dots, s_n)$  and  $Y = (t_1, \dots, t_n)$ . Toward proving the stochastic dominance, we couple  $X$  and  $Y$ , first by revealing  $Y$ , and then, for every  $1 \leq j \leq n - 1$ , set  $s_j = \beta_j t_j / \alpha_j$ . Now, for every  $j = 1, \dots, n$ , and any positive number  $x$  we have that

$$\begin{aligned} \Pr[s_j \geq x] &= \Pr\left[\frac{\alpha_j}{\beta_j} s_j \geq \frac{\alpha_j}{\beta_j} x\right] = \Pr\left[t_j > \frac{\alpha_j}{\beta_j} x\right] \\ &= \exp\{-\beta_j(\alpha_j/\beta_j)x\} = \exp\{-\alpha_j x\}. \end{aligned}$$

Therefore,  $X = (s_1, \dots, s_n)$  are exponentially distributed according to  $A$ . Also,  $\beta_j/\alpha_j \geq 1$  and hence, for every  $1 \leq m \leq n - 1$ ,  $t_j \leq s_j$  and we get that

$$\sum_{j=1}^m t_j \leq \sum_{j=1}^m s_j.$$

This implies that, Poisson process  $\mathbf{P}(A)$  requires at least as much time as  $\mathbf{P}(B)$  requires to generate  $m$  events. For any given  $t$ , let  $\mathbf{N}_A(t)$  and  $\mathbf{N}_B(t)$  denote the number of events

## 31:16 Rumor Spreading in Dynamic Graphs

happened by  $\mathbf{P}(A)$  and  $\mathbf{P}(B)$  during  $[0, t]$ , respectively. Then we have that  $\mathbf{N}_A(t) \leq \mathbf{N}_B(t)$ . Thus, for every positive integer  $k$ , we get that

$$\Pr[\mathbf{N}_B(t) \leq k] \leq \Pr[\mathbf{N}_A(t) \leq k],$$

which proves the lemma.  $\blacktriangleleft$

We will combine the Lemmas 9 and 10 from [20] and get the following concentration bound for a particular set of exponential random variables.

► **Lemma A.1** (Lemma 9 and 10. [20]). *Let  $f(m, j)$  be a deterministic sequence such that for  $1 \leq j < m$ ,  $\mathbf{E}[s_j | I_j]^{-1} \geq f(m, j)$  and  $f(m, j) = \Theta(\min\{j, m - j\})$ . Moreover, let  $\{t_j^+\}_{j=1}^{m-1}$  be a sequences of independent random variables, where  $t_j^+$  is exponentially distributed with parameter  $f(m, j)$ . Also let  $T^+ = \sum_{j=1}^{m-1} t_j^+$ , and  $S_m = \sum_{j=1}^{m-1} s_j$ . Then, we have*

$$\text{for } 0 < \lambda < \min_{j \in [m-1]} f(m, j), \quad \mathbf{E}[e^{\lambda S_m}] \leq \exp\{\lambda \mathbf{E}[T^+] + O(1)\}.$$

This implies that, for every  $z > 0$ ,

$$\Pr[S_{m-1} \geq z] \leq \exp\{\lambda \mathbf{E}[T^+] + O(1) - \lambda z\}$$

## B Properties of Geometric Mobile Networks

This section is devoted to the proof of Lemma 4.1. Geometric mobile model is a dynamic evolving network  $\mathcal{M}(n, R) = \{G^{(t)}\}_{t=0}^{\infty}$  contains a set of  $n$  agents, denoted by  $[n]$ . Each agent independently performs nearest neighbor random walks on  $H = (L_{n,\epsilon}, E)$  and there is an edge between two agents if their Euclidean distance is at most  $R$ . Recall that

$$L_{n,\epsilon} = \{(k \cdot \epsilon, l \cdot \epsilon) : k, l \in \mathbb{N}, k, l \leq \sqrt{n}/\epsilon\},$$

and

$$E = \{\{x, y\} : x, y \in L_{n,\epsilon}, d(x, y) \leq r\}.$$

It is easy to see that for every  $x \in L_{n,\epsilon}$  there are  $\Theta(r^2)$  locations at distance at most  $r$  from  $x$  and hence  $H$  is almost regular (i.e., the ratio of the maximum and minimum degree is at most a constant). Since  $H$  is connected and almost regular, a Markov chain defined by the random walk is ergodic and converges to an almost uniform stationary distribution over  $L_{n,\epsilon}$ , say  $\pi$ . Notice that by an almost uniform we mean that for every location  $x, y \in L_{n,\epsilon}$ ,  $\pi(x)/\pi(y) = \Theta(1)$ . Since  $n$  agents perform random walks on  $H$ , we will have an Markov chain with state space

$$\underbrace{L_{n,\epsilon} \times L_{n,\epsilon} \times \dots \times L_{n,\epsilon}}_{n \text{ times}}.$$

Then, by a basic property of ergodic and finite Markov chains, at any time  $t \geq 1$ , each agent is located at location  $x \in L_{n,\epsilon}$  with probability  $\pi(x)$ . Before proving Lemma 4.1, we first present some useful lemmas. Note that we use node or agent but they have the same meaning.

► **Lemma B.1.** *Let  $A$  denote a arbitrary 2-dimensional grid with  $m'$  nodes and  $S$  be an arbitrary set of nodes in  $A$ , with size at most  $m'/2$ . Then, there exists a constant  $c > 0$  such that  $N(S) \geq c' \sqrt{|S|}$ , where  $N(S)$  is the number of nodes that have at least one neighbor in  $S$ .*



The proof can be found in [7, Theorem 4.1].

► **Lemma B.2.** *Let  $M$  be an  $m \times m$  grid embedded on  $\sqrt{n} \times \sqrt{n}$  square plane, where  $m = \sqrt{5n}/R$ . Then, with high probability, each cell of  $M$  contains  $\Theta(R^2)$  agents.*

**Proof.** Each cell in  $M$  is an  $R/\sqrt{5} \times R/\sqrt{5}$  square and contains  $R^2/5\varepsilon^2$  nodes from  $L_{n,\varepsilon}$ . Let us fix some  $1 \leq t \leq n^3$  and arbitrary cell  $C$ . Then, the location of each agent at time  $t$  has an almost uniform distribution  $\pi$  over  $L_{n,\varepsilon}$ . For every agent  $u \in [n]$ , define indicator random variable  $I_{u,C}$  as follows:

$$I_{u,C} = \begin{cases} 1 & \text{if } u \text{ is located in cell } C, \\ 0 & \text{otherwise.} \end{cases}$$

Thus,

$$\Pr[I_{u,C} = 1] = \sum_{x \in C \cap L_{n,\varepsilon}} \pi(x) = (R^2/5\varepsilon^2) \times \Theta(\varepsilon^2/n) = \Theta(R^2/5n).$$

Also let  $Y = \sum_{u \in [n]} I_{u,C}$  to denote the number of agents at time  $t$  in cell  $C$ . By the linearity of expectation we have that  $\mathbf{E}[Y] = \Theta(R^2)$ . Applying a Chernoff bound, we conclude that

$$\Pr[|Y - \mathbf{E}[Y]| \geq \mathbf{E}[Y]/2] \leq e^{-\mathbf{E}[Y]/12} = n^{-\omega(1)}.$$

Therefore, with probability  $n^{-\omega(1)}$ , cell  $C$  does not contain  $\Theta(R^2)$  agents at time  $t$ . An application of union bound over all time steps and cells implies that with probability  $1 - n^{-\omega(1)}$ , for every  $1 \leq t \leq n^3$ , each cell of  $M$  contains  $\Theta(R^2)$  agents which completes the proof. ◀

► **Lemma B.3** (Restatement of Lemma 4.1). *Suppose that  $\mathcal{M}(n, R) = \{G^{(t)}\}_{t=0}^{\infty}$ , is a geometric mobile network with  $f(n)\sqrt{\log n} \leq R \leq \sqrt{n}$ , where  $f(n)$  is a slowly growing function in  $n$ . Then, with probability  $1 - n^{-\omega(1)}$ , for every  $1 \leq t \leq n^3$ , the followings hold:*

1. *For every node  $u$  (agent),  $d_u(t) = \Theta(R^2)$ , where  $d_u(t)$  is the degree of node  $u$  in  $G^{(t)}$ .*
2. *There exists constant  $a > 0$  such that conductance function  $G^{(t)}$  satisfies*

$$\Phi(x) \geq \begin{cases} a & 1 \leq x \leq R^2, \\ a \frac{R}{\sqrt{\min\{x, n-x\}}} & R^2 < x \leq n-1. \end{cases}$$

**Proof of (1).** Let us fix an arbitrary time step  $1 \leq t \leq n^3$  and an arbitrary agent, say  $u$ , that is located at some  $x \in L_{n,\varepsilon}$ . Define

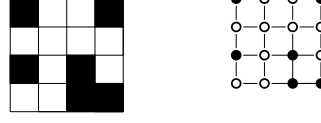
$$B(x) = \{y : y \in L_{n,\varepsilon}, d(x, y) \leq R\}.$$

It is not hard to see that for every  $x$ ,  $|B(x)| = \Theta((R/\varepsilon)^2)$ . For every  $y \in B(x)$  and  $u \in [n]$ , let us define the indicator random variable  $I_{u,y}$  as follows:

$$I_{u,y} = \begin{cases} 1 & \text{if } u \text{ is located at } y, \\ 0 & \text{otherwise.} \end{cases}$$

Clearly,  $Y = \sum_{v \in [n] \setminus u} \sum_{y \in B(x)} I_{v,y}$  is the degree of agent  $u$  in  $G^{(t)}$ . Since every agent  $v$  has almost uniform distribution  $\pi$  over  $L_{n,\varepsilon}$  and agents are independent from each other, we get that

$$\Pr[I_{v,y} = 1] = \pi(y) = \Theta(\varepsilon^2/n).$$



■ **Figure 1** The right grid is the dual of colored left grid, where colored faces are translated to colored vertices.

Thus, by linearity of expectation we have

$$\begin{aligned} \mathbf{E}[Y] &= \sum_{v \in [n] \setminus u} \sum_{y \in B(x)} \mathbf{E}[I_{v,y}] = \sum_{v \in [n] \setminus u} \sum_{y \in B(x)} \pi(y) \\ &= (n-1) \sum_{y \in B(x)} \pi(y) = (n-1)|B(x)|\Theta(\varepsilon^2/n) = \Theta(R^2) = \omega(\log n). \end{aligned}$$

$I_{v,y}$ 's are mutually independent so we apply a Chernoff bound and conclude that

$$\Pr[|Y - \mathbf{E}[Y]| \geq \mathbf{E}[Y]/2] \leq e^{-\mathbf{E}[Y]/12} = n^{-\omega(1)}.$$

Note that the above inequality holds only for an arbitrary and fixed time step  $t$  and node  $u$ . By union bound over all  $n$  agents and  $n^3$  time steps, we conclude that with probability  $n^3 \times n \times n^{-\omega(1)}$  there is a time step  $s$  and agent  $w$  such that  $d_w(s) \notin [\mathbf{E}[Y]/2, 3\mathbf{E}[Y]/2]$ . Therefore, with probability  $1 - n^{-\omega(1)}$ , for every agent  $u$  and time step  $t$ ,  $d_u(t) = \Theta(R^2)$ , which completes proof of (1).

**Proof of (2).** Suppose that  $m = \sqrt{5n}/R$  and consider an  $m \times m$  grid  $M$  embedded in a plane square of  $\sqrt{n} \times \sqrt{n}$ , whose cells are  $\sqrt{R/5} \times \sqrt{R/5}$  squares. By Lemma B.2, with high probability, for every  $t$ , each cell of  $M$  contains  $\Theta(R^2)$  agents. Fix an arbitrary set of agents (nodes), say  $S$ , with size  $1 \leq s \leq n/2$ . Also fix some time step  $t$ . Then, with respect to  $S$  and time  $t$ , we color cells of  $M$  as follows. Cell  $C$  becomes white, if at most  $3/4$  agents in  $C$  are contained in  $S$  and black otherwise. As a result, each cell of  $M$  gets colored by either black or white at time step  $t$ . Let  $B$  and  $W$  denote the set of black and white cells in  $M$ . Now, let us consider the dual of the grid  $M$ , which is again a  $(m-1) \times (m-1)$  grid. Notice that the vertex set of the dual graph is the interior faces of the primal and two vertices in the dual are connected if their corresponding faces (cells) are side by side (e.g. see Figure 1). By definition of  $\mathcal{M}(n, R) = \{G^{(t)}\}_{t=0}^{\infty}$ , every two agents located at any two side-by-side cells are connected by an edge, because their Euclidean distance is at most  $\sqrt{4R^2/5 + R^2/5} = R$ . According to the size of  $B$  we consider two cases:

■  $|B| < m^2/2$ :

Let  $D$  be the set of vertices corresponding to cells of  $B$  in the dual graph (i.e. set of black nodes in the right grid of Fig. 1). Thus,  $|D| < m^2/2$ . By Lemma B.1, we have that  $N(D) \geq c'\sqrt{|D|}$ , for some constant  $c'$ . This implies that there are at least  $c'\sqrt{|B|}$  white cells that are connected to cells of  $B$ . By the coloring rule, we deduce that if a black cell and a white cell are side by side, then at least  $3/4$  agents from the black cell contained in  $S$  are connected to at least  $1/4$  agents of the white cell contained in  $\bar{S}$ . Moreover, every agent of  $S$  contained in a white cell is connected to at least  $1/4$  agents from the same cell, which are contained in  $\bar{S}$ . Remember that by Lemma B.2 each cell contains  $\Theta(R^2)$  agents, w.h.p. Thus, we get that in  $G^{(t)}$ ,

$$|E(S, \bar{S})| \geq c\sqrt{|B|}\Theta(R^2) + \sum_{C \in W} x(C)\Theta(R^2),$$

where  $x(C)$  is the number of agents in  $S$  that are contained in white cell  $C$ . Moreover,  $G^{(t)}$  is almost regular with degree  $\Theta(R^2)$  and each cell contains at most  $\Theta(R^2)$  agents. So we have

$$\text{vol}(S) \leq |B|\Theta(R^4) + \sum_{C \in W} x(C)\Theta(R^2).$$

Now, we may consider two cases  $|B| = 0$  and  $|B| > 0$ . In first case, by two above inequalities we get

$$\frac{|\mathbb{E}(S, \bar{S})|}{\text{vol}(S)} = \Theta(1). \quad (14)$$

For the second case we get

$$\begin{aligned} \frac{|\mathbb{E}(S, \bar{S})|}{\text{vol}(S)} &\geq \frac{\sqrt{|B|}\Theta(R^4) + \sum_{C \in W} x(C)\Theta(R^2)}{|B|\Theta(R^4) + \sum_{C \in W} x(C)\Theta(R^2)} \geq \Theta\left(\frac{\sqrt{|B|}R^4}{|B|R^4}\right) \\ &= \Theta\left(\frac{1}{\sqrt{|B|}}\right) \geq \Theta\left(\sqrt{\frac{3R^2}{20|S|}}\right) = \Theta\left(\frac{R}{\sqrt{|S|}}\right), \end{aligned} \quad (15)$$

where the second inequality comes from the fact for every number  $z > x > 0$  and arbitrary  $z > 0$ , we have that  $\frac{x+y}{z+y} \geq \frac{x}{z}$ . Also, the third one follows from  $|B|\Theta(R^2)3/4 \leq |S|$ , as  $3/4$  agents in each black cell contained in  $S$ .

- $|B| \geq m^2/2$ : In this case, we first observe that  $|W| = \Theta(m^2)$ . Toward a contradiction, we assume that

$$|W| = o(m^2) = o(n/R^2)$$

and hence white cells can have at most  $|W|\Theta(R^2) = o(n)$  agents, by lemma B.2, each cell contains  $\Theta(R^2)$  agents. On the other hand, by definition, black cells can accommodate at most  $n/4$  agents from  $\bar{S}$ , which contradicts assumption that  $|\bar{S}| \geq n/2$ . So we have that  $|W| = \Theta(m^2) = \Theta(|B|)$ . Since  $|W| + |B| = m^2$  we conclude that  $|W| < m^2/2$ . Again similar to the previous case, there are at least  $c\sqrt{|W|}$  black cells, which are adjacent to white cells and we have

$$|\mathbb{E}(S, \bar{S})| \geq c\sqrt{|W|}\Theta(R^2) + \sum_{C \in W} x(C)\Theta(R^2).$$

Moreover,

$$\text{vol}(S) \leq |B|\Theta(R^4) + \sum_{C \in W} x(C)\Theta(R^2) = |W|\Theta(R^4) + \sum_{C \in W} x(C)\Theta(R^2),$$

where it follows from the fact that  $|W| = \Theta(|B|)$ . Similar to the previous case we will have,

$$\begin{aligned} \frac{|\mathbb{E}(S, S^c)|}{\text{vol}(S)} &\geq \frac{\sqrt{|W|}\Theta(R^4) + \sum_{C \in W} x(C)\Theta(R^2)}{|W|\Theta(R^4) + \sum_{C \in W} x(C)\Theta(R^2)} \\ &\geq \Theta\left(\frac{1}{\sqrt{|W|}}\right) = \Theta\left(\frac{1}{\sqrt{|B|}}\right) = \Theta\left(\sqrt{\frac{3R^2}{20|S|}}\right) = \Theta\left(\frac{R}{\sqrt{|S|}}\right). \end{aligned} \quad (16)$$

### 31:20 Rumor Spreading in Dynamic Graphs

From Inequalities (14), (15), and (16) we conclude that for every time step  $1 \leq t \leq n^3$  and a set of agents of size at most  $n/2$  in  $G^{(t)}$ , there exists constant  $a > 0$  such that

$$\frac{|\mathbb{E}(S, \bar{S})|}{\text{vol}(S)} \geq \min \left\{ a \frac{R}{\sqrt{|S|}}, a \right\}. \quad (17)$$

For every subset of agents, say  $S$ , define

$$g(S) = \begin{cases} S & \text{if } |S| \leq n/2, \\ \bar{S} & \text{otherwise.} \end{cases}$$

Clearly, we have that  $|g(S)| = \min\{|S|, n - |S|\} \leq n/2$  and  $|\mathbb{E}(S, \bar{S})| = |\mathbb{E}(g(S), \overline{g(S)})|$  completing the proof.  $\blacktriangleleft$